EPTCS 368

Proceedings of the

Combined 29th International Workshop on Expressiveness in Concurrency

and 19th Workshop on Structural Operational Semantics

Warsaw, Poland, 12th September 2022

Edited by: Valentina Castiglioni and Claudio A. Mezzina

Published: 6th September 2022
DOI: 10.4204/EPTCS.368
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Preface

This volume contains the proceedings of EXPRESS/SOS 2022, the Combined 29th International Workshop on Expressiveness in Concurrency (EXPRESS) and the 19th Workshop on Structural Operational Semantics (SOS).

The first edition of EXPRESS/SOS was held in 2012, when the EXPRESS and SOS communities decided to organise an annual combined workshop bringing together researchers interested in the formal semantics of systems and programming concepts, and in the expressiveness of computational models. Since then, EXPRESS/SOS was held as one of the affiliated workshops of the International Conference on Concurrency Theory (CONCUR). Following this tradition, EXPRESS/SOS 2022 was held affiliated to CONCUR 2022, as part of CONFEST 2022, in Warsaw, Poland.

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

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

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

- *From Legal Contracts to Legal Calculi: the code-driven normativity*, by Silvia Crafa (University of Padova, Italy);
- *How advances in open bisimilarity help certify the privacy of contactless payments and ePassports*, by Ross Horne (University of Luxembourg, Luxembourg).

We would like to thank the authors of the submitted papers, the invited speakers, the members of the program committee, and their subreviewers for their contribution to both the meeting and this volume. We also thank the CONCUR 2022 and the CONFEST 2022 organizing committees for hosting the workshop. Finally, we would like to thank our EPTCS editor Rob van Glabbeek for publishing these proceedings and his help during the preparation.

Valentina Castiglioni and Claudio Antares Mezzina,
August 2022

Program Committee

- Gerogiana Caltais , University of Twente, The Netherlands
- Valentina Castiglioni (co-chair), Reykjavik University, Iceland
- Wan Fokkink , Vrije University Amsterdam, The Netherlands

- Paola Giannini , University of Piemonte Orientale, Italy
- Daniel Hirschkoff , ENS Lyon, France
- Mathias Jakobsen , University of Glasgow, UK
- Tobias Kappé ,University of Amsterdam, The Netherlands
- Vasileios Koutavas , Trinity College Dublin, Ireland
- Ondrej Lengal , Brno University of Technology, Czech Republic
- Gerald Lüttgen , University of Bamberg, Germany
- Claudio Antares Mezzina (co-chair) University of Urbino, Italy
- Max Tschaikowski , Aalborg University, Denmark

Additional reviewers

- Vojtěch Havlena

Bisimulations Respecting Duration and Causality for the Non-interleaving Applied π-Calculus

Clément Aubert
Augusta University, USA
caubert@augusta.edu

Ross Horne
University of Luxembourg, Luxembourg
ross.horne@uni.lu

Christian Johansen
NTNU, Norway
christian.johansen@ntnu.no

This paper shows how we can make use of an asynchronous transition system, whose transitions are labelled with events and which is equipped with a notion of independence of events, to define non-interleaving semantics for the applied π-calculus. The most important notions we define are: Start-Termination or ST-bisimilarity, preserving duration of events; and History-Preserving or HP-bisimilarity, preserving causality. We point out that corresponding similarity preorders expose clearly distinctions between these semantics. We draw particular attention to the distinguishing power of HP failure similarity, and discuss how it affects the attacker threat model against which we verify security and privacy properties. We also compare existing notions of located bisimilarity to the definitions we introduce.

1 Introduction

Non-interleaving semantics is sometimes referred to as true concurrency. This reflects the idea that parallel composition has a semantically distinct status from its interleavings obtained by allowing each parallel process to preform actions one-by-one in any order. In this work, we explore a spectrum of non-interleaving semantics for the applied π-calculus, which is motivated by some recent works on modelling and verifying security and privacy properties of cryptographic protocols [9, 21]. The definitions introduced are operational in style, bypassing denotations such as event structures.

We build on our recent work [4] that introduced a non-interleaving Structural Operational Semantics (SOS) for the applied π-calculus that generates Labelled Asynchronous Transition Systems (LATS). Compared with standard transition systems, whose transitions are labelled with actions, a LATS labels its transitions with richer *events*, and is equipped with a notion of *independence* over adjacent events (concurrently enabled or enabled one after another). A LATS allows independent events to be permuted and hence techniques such as partial-order reduction to be applied. This work is part of a research agenda where we wish to lay a foundation for exploring questions such as whether verification techniques are enabled by adopting a semantics that is naturally compatible with an independence relation used for partial-order reduction. Another research question is whether adopting a non-interleaving semantics impacts the attacker model for certain problems. In particular, armed with our definitions, we may ask whether our non-interleaving semantics may detect attacks that may be missed if we employ an interleaving semantics.

The contribution of this paper towards addressing the questions above is the introduction of non-interleaving equivalences and similarities that can be defined for the applied π-calculus equipped with a LATS [4]. A well understood starting point is how to generate "located" equivalences [6] for CCS [7, 24] and the π-calculus [27]. The former approach makes direct use of the LATS for CCS, while the latter uses a cut down located transition system for the π-calculus which accounts for locations but does not satisfy all properties of a LATS. We go further since, given our LATS, we can generate in an operational

V. Castiglioni and C. A. Mezzina (Eds): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics 2022 (EXPRESS/SOS 2022).
EPTCS 368, 2022, pp. 3–22, doi:10.4204/EPTCS.368.1

Terminology	*Remarks*	*Def.*
i-similarity	"Interleaving"-similarity is the notion of similarity most commonly explored in the literature.	Def. 8
ST-similarity	"Start-Terminate"-similarity accounts for the fact that events have duration. It uses events to distinguish between actions with the same label, and to ensure that two "terminate" events correspond to the same "start" event.	Def. 11
HP-similarity	"History-Preserving"-similarity preserves the causal dependencies between events.	Def. 12
I-similarity	"Independence"-similarity are parametrised by some notion of independence I. We obtain "located bisimilarities" using the structural independence relation I_ℓ that considers only if two events are in different locations.	Def. 16

Table 1: Strategies in the interleaving/non-interleaving spectrum explored for the applied π-calculus.

style other notions of non-interleaving semantics, particularly those that preserve duration of events (Start-Termination or ST semantics) [15] and those that preserve causality (History-Preserving or HP semantics) [12, 26]. Since we cover the applied π-calculus, of course, we encompass the π-calculus, where the later surprisingly benefits from adopting a modern applied π-calculus style when handling *link causality* – the causal relationship between outputs and inputs that depend upon them. Our operational approach also avoids the need to unfold to event structures [10, 30] or configuration structures [11] that would track entire histories of causal dependencies; instead, we consider only what is happening or enabled at a particular point in time.

We include in Tables 1 and 2 a glossary, including key standard and non-standard terminology employed in this paper. We emphasise similarity rather than bisimilarity for two reasons. Firstly, similarity exposes more clearly than bisimilarity the differences between non-interleaving semantics as it allows clearer separating examples. Secondly, similarity is known to have compelling attacker models in terms of probabilistic may testing [13], and it is standard in computational security to consider probabilistic attackers [8]. Table 1 presents the notions of similarity that we discuss in the interleaving/non-interleaving spectrum we explore. Along this spectrum the attacker has different powers for observing concurrency.

While we draw attention to similarity, we are also interested in non-interleaving *bi*similarity and other notions in the linear-time/branching-time spectrum [16]. Indeed, all the notions in Table 1 also exist in their other variants in the linear-time/branching time spectrum listed in Table 2, such as failure similarity. Along this spectrum the observer has more or less power to observe and make choices. We also use the term *mutual*, e.g., mutual ST-similarity, when some notion of similarity holds in both directions.

There are further spectra that could be explored: for the π-calculus there is the open/early spectrum, including notions such as early, late, quasi-open [29], and open [28] variants of equivalences. This work considers only *early* and *strong* semantics: early semantics means that the message input is chosen at the moment the event starts, whereas the other variants allow different degrees of laziness in learning what message was input retrospectively. This choice is made since the majority of equivalences for the applied π-calculus in the literature are early, and early bisimilarity coincides with notions of testing via concurrent processes [1]. Since our semantics are strong, every τ-transition is matched by exactly one τ-transition in all our strategies. Many security and privacy problems that motivate us can be reduced to a strong equivalence problem. However, the main reason for these choices is simply to focus on the interleaving/non-interleaving spectrum. For example, it would be easy to define quasi-open variants of our non-interleaving semantics, which coincide with a testing semantics making use of all contexts [22].

Terminology	*Remarks*	*Symb.*
X-bisimilarity	An equivalence ranging over all strategies of a particular type X.	$\sim_X$
X-similarity	The preorder arising when we assume one player leads throughout a strategy (except when testing equations, as explained around Def. 8).	$\preceq_X$
X-presimilarity	A notion of similarity we introduce in this paper (Def. 7) to emphasise the testing power of inequalities in the applied π-calculus.	$\sqsubseteq_X$
Xf-similarity	X "failure" similarity is one of many variants of similarity in the linear-time/branching-time spectrum, and is chosen due to its testing model allowing us to test if something is not enabled. In particular, we look at STf-similarity (Def. 13) and HPf-similarity (Def. 14).	$\preceq_{Xf}$

Table 2: Notions in the linear-time/branching-time spectrum explored for the applied π-calculus.

After briefly recalling our non-interleaving SOS generating a LATS (Sect. 2), we use interleaving semantics to illustrate and motivate the genericity of *static* equivalences (Sect. 3). Sect. 4 is the core of our proposal: it starts by introducing and stressing the importance of the independence relation (Sect. 4.1), which is used throughout the rest of the article. ST and HP-similarities are then defined in Sect. 4.2 and 4.3 and compared in the context of privacy in Sect. 4.4. Sect. 4.5 discusses failure semantics for HP- and ST-similarities. Some design decisions are justified in light of located bisimulations in Sect. 5.

2 Background: A Non-interleaving SOS for the Applied π-Calculus

This section recalls a non-interleaving structural operational semantics for the applied π-calculus. The design decisions are discussed extensively in a companion paper [4]. What we present below is intended only as a condensed summary of that operational semantics for ease of reference.

All variables x,y,z are the same syntactic category, but are distinct from *aliases*. Aliases range over α,β,γ and consist of an alias variable, say λ, prefixed with a string $s \in \{0,1\}^*$, i.e., $\alpha = s\lambda$. *Messages* range over M,N,K, built from a signature of function symbols Σ. As standard, a *substitution* σ,θ or ρ is a function with a domain ($\mathrm{dom}(\sigma) = \{\alpha\colon \alpha \neq \alpha\sigma\}$) and a range ($\mathrm{ran}(\sigma) = \{\alpha\sigma\colon \alpha \in \mathrm{dom}(\sigma)\}$) that are applied in suffix form. The *identity substitution* is denoted id and composition $\sigma \circ \theta$.

Processes are denoted by P,Q,R, and in $\nu x.P$ and $a(x).P$ occurrences of x in P are bound. Sequences of names $\nu\vec{x}.P$ abbreviate multiple name binders defined inductively such that $\nu\varepsilon.P = P$ and $\nu x,\vec{y}.P = \nu x.\nu\vec{y}.P$, where ε is the empty sequence. *Active substitutions*, denoted σ, θ, map aliases in their finite domain to messages containing no aliases, and appear in *extended processes*, ranging over A,B,C. We assume a *normal form*, where aliases do not appear in processes, and an *equational theory* E containing equalities on messages, e.g., $\mathtt{dec}(\{M\}_K,K) =_E M$. Figs. 1 and 2 give the syntax and semantics.

Definition 1 (freshness, α-equivalence, etc.)**.** *A variable x (resp. an alias α) is* free in a message M *if* $x \in \mathrm{fv}(M)$ *(resp.* $\alpha \in \mathrm{fa}(M)$*) for*

$$\mathrm{fv}(f(M_1,\ldots M_n)) = \cup_{i=1}^n \mathrm{fv}(M_i) \qquad \mathrm{fv}(x) = \{x\} \qquad \mathrm{fv}(\alpha) = \emptyset$$
$$\mathrm{fa}(f(M_1,\ldots M_n)) = \cup_{i=1}^n \mathrm{fa}(M_i) \qquad \mathrm{fa}(x) = \emptyset \qquad \mathrm{fa}(\alpha) = \{\alpha\}\,.$$

The fv *function extends in the standard way to (extended) processes, letting* $\mathrm{fv}(\nu x.P) = \mathrm{fv}(P) \setminus \{x\}$ *and* $\mathrm{fv}(M(x).P) = \mathrm{fv}(M) \cup (\mathrm{fv}(P) \setminus \{x\})$*, and similarly for* $\mathrm{fv}(A)$*. The functions for free variables and free*

PROCESSES:

$P,Q,R ::=$	0		deadlock
	$\mid$	$\nu x.P$	new
	$\mid$	$P \mid Q$	parallel
	$\mid$	G	guarded process
	$\mid$	$!P$	replication

GUARDED PROCESSES:

$G,H ::=$	$M(x).P$		input prefix
	$\mid$	$\overline{M}\langle N\rangle.P$	output prefix
	$\mid$	$[M=N]G$	match
	$\mid$	$[M\neq N]G$	mismatch
	$\mid$	$G+H$	choice

EXTENDED PROCESSES:

$A,B ::=$	$\sigma \mid P$		active process
	$\mid$	$\nu x.A$	new

MESSAGES:

$M,N ::=$	x		variable
	$\mid$	α	alias
	$\mid$	$f(M_1,\ldots,M_n)$	function

EARLY ACTION LABELS:

$\pi ::=$	MN		free input
	$\mid$	$\overline{M}(\alpha)$	output
	$\mid$	τ	interaction

Figure 1: Syntax of extended processes with guarded choices, where $f \in \Sigma$.

aliases extend to labels as follows.

$$\mathrm{fv}(\pi) = \begin{cases} \mathrm{fv}(M)\cup\mathrm{fv}(N) & \text{if } \pi = MN \\ \mathrm{fv}(M) & \text{if } \pi = \overline{M}(\alpha) \\ \emptyset & \text{if } \pi = \tau \end{cases} \qquad \mathrm{fa}(\pi) = \begin{cases} \mathrm{fa}(M)\cup\mathrm{fa}(N) & \text{if } \pi = MN \\ \mathrm{fa}(M) & \text{if } \pi = \overline{M}(\alpha) \\ \emptyset & \text{if } \pi = \tau \end{cases}$$

We say a variable x is fresh for a message M (resp. process P, extended process A), *written $x \# M$ (resp. $x \# P$, $x \# A$) whenever $x \notin \mathrm{fv}(M)$ (resp. $x \notin \mathrm{fv}(P)$, $x \notin \mathrm{fv}(A)$), and similarly for aliases. Freshness extends point-wise to lists of entities, i.e., $x_1,x_2,\ldots x_m \# M_1,M_2,\ldots,M_n$, denotes the conjunction of all $x_i \# M_j$ for all $1 \le i \le m$ and $1 \le j \le n$.*

We define α-equivalence *(denoted $\equiv_\alpha$) for variables only (not aliases which are fixed "addresses") as the least congruence (a reflexive, transitive, and symmetric relation preserved in all contexts) such that, whenever $z \# \nu x.P$, we have $\nu x.P \equiv_\alpha \nu z.(P\{^z/_x\})$ and $M(x).P \equiv_\alpha M(z).(P\{^z/_x\})$. Similarly, for extended processes, we have the least congruence such that, whenever $z \# \nu x.A$, we have $\nu x.A \equiv_\alpha \nu z.(A\{^z/_x\})$. Restriction is such that $\theta\restriction_{\vec{\alpha}}(x) = \theta(x)$ if $x \in \vec{\alpha}$ and x otherwise.*

Capture-avoiding substitutions *are defined for processes such that $(M(x).P)\sigma \equiv_\alpha M\sigma(z).P\{^z/_x\}\sigma$ and $(\nu x.P)\sigma \equiv_\alpha \nu z.P\{^z/_x\}\sigma$ for $z \# \mathrm{dom}(\sigma),\mathrm{ran}(\sigma),\nu x.P$. For extended processes, it is defined such that $(\nu x.A)\rho \equiv_\alpha \nu z.(A(\{^z/_x\}\circ\rho))$ and $(\sigma \mid P)\rho = (\sigma\circ\rho\restriction_{\mathrm{dom}(\sigma)} \mid P\rho)$, for $z \# \mathrm{dom}(\rho),\mathrm{ran}(\rho),\nu x.A$.*

Definition 2 (structural congruence)**.** *Our minimal* structural congruence *(denoted $\equiv$) is the least equivalence relation on extended processes extending α-equivalence such that whenever $\sigma = \theta$, $P \equiv_\alpha Q$ and $A \equiv B$, we have: $\sigma \mid P \equiv \theta \mid Q$, $\nu x.A \equiv \nu x.B$ and $\nu x.\nu z.A \equiv \nu z.\nu x.A$.*

Definition 3 (location labels)**.** *A* location *ℓ is of the form $s[t]$, where $s \in \{0,1\}^*$ and $t \in \{0,1\}^*$. If s or t is empty, we omit it (hence, we write $\varepsilon[\varepsilon]$ as $[]$). A* location label *u is either a location ℓ or a pair of locations (ℓ_0,ℓ_1), and we let $c(\ell_0,\ell_1) = (c\ell_0,c\ell_1)$ for $c \in \{0,1\}$.*

3 Handling located aliases, explained using interleaving similarities

Although the objective of this paper is to explore non-interleaving semantics, we begin by defining an interleaving semantics. The reason is that we wish to expose clearly which parts of our definitions are generic to any type of semantics, and which are specific to non-interleaving semantics.

$$\frac{M =_E K}{K(x).P \xrightarrow[[]]{MN} \mathrm{id} \mid P\{^N/_x\}}\ \textsc{Inp} \qquad \frac{M =_E K}{\overline{K}\langle N\rangle.P \xrightarrow[[]]{\overline{M}(\lambda)} \{^N/_\lambda\} \mid P}\ \textsc{Out}$$

$$\frac{P \xrightarrow[u]{\pi} \nu\vec{x}.(\sigma \mid R) \quad \vec{x}\#Q}{P \mid Q \xrightarrow[0u]{\pi} \nu\vec{x}.(\sigma \mid R \mid Q)}\ \textsc{Par-L} \qquad \frac{Q \xrightarrow[u]{\pi} \nu\vec{x}.(\sigma \mid R) \quad \vec{x}\#P}{P \mid Q \xrightarrow[1u]{\pi} \nu\vec{x}.(\sigma \mid P \mid R)}\ \textsc{Par-R}$$

$$\frac{P\{^z/_x\} \xrightarrow[u]{\pi} A \qquad z\#\mathrm{fv}(\pi), \nu x.P}{\nu x.P \xrightarrow[u]{\pi} \nu z.A}\ \textsc{Extrude} \qquad \frac{A \xrightarrow[u]{\pi} B \qquad x\#\mathrm{fv}(\pi)}{\nu x.A \xrightarrow[u]{\pi} \nu x.B}\ \textsc{Res}$$

$$\frac{P \xrightarrow[s[s']]{\overline{M\sigma}(\lambda)} \nu\vec{x}.\left(\{^N/_\lambda\} \mid Q\right) \quad \vec{x}\#\mathrm{ran}(\sigma) \quad \mathrm{fa}(M) \subseteq \mathrm{dom}(\sigma) \quad s\lambda\#\mathrm{dom}(\sigma)}{\sigma \mid P \xrightarrow[s[s']]{\overline{M}(s\lambda)} \nu\vec{x}.\left(\sigma \circ \{^N/_{s\lambda}\} \mid Q\right)}\ \textsc{Alias-out}$$

$$\frac{P \xrightarrow[u]{\pi\sigma} \nu\vec{x}.(\mathrm{id} \mid Q) \quad \vec{x}\#\mathrm{ran}(\sigma) \quad \mathrm{fa}(\pi) \subseteq \mathrm{dom}(\sigma)}{\sigma \mid P \xrightarrow[u]{\pi} \nu\vec{x}.(\sigma \mid Q)}\ \textsc{Alias-free}$$

$$\frac{G \xrightarrow[[t]]{\pi} A}{G+H \xrightarrow[[0t]]{\pi} A}\ \textsc{Sum-L} \qquad \frac{H \xrightarrow[[t]]{\pi} A}{G+H \xrightarrow[[1t]]{\pi} A}\ \textsc{Sum-R} \qquad \frac{P \mid !P \xrightarrow[u]{\pi} A}{!P \xrightarrow[u]{\pi} A}\ \textsc{Bang}$$

$$\frac{P \xrightarrow[u]{\pi} A \qquad M =_E N}{[M=N]P \xrightarrow[u]{\pi} A}\ \textsc{Mat} \qquad \frac{P \xrightarrow[u]{\pi} A \qquad M \neq_E N}{[M\neq N]P \xrightarrow[u]{\pi} A}\ \textsc{Mismat}$$

$$\frac{P \xrightarrow[\ell_0]{\overline{M}(\lambda)} \nu\vec{y}.\left(\{^N/_\lambda\} \mid P'\right) \qquad Q \xrightarrow[\ell_1]{MN} \nu\vec{w}.\left(\mathrm{id} \mid Q'\right) \qquad \vec{y}\#Q \qquad \vec{w}\#P,\vec{y}}{P \mid Q \xrightarrow[(0\ell_0,1\ell_1)]{\tau} \nu\vec{y},\vec{w}.\left(\mathrm{id} \mid P' \mid Q'\right)}\ \textsc{Close-L}$$

$$\frac{P \xrightarrow[\ell_0]{MN} \nu\vec{y}.\left(\mathrm{id} \mid P'\right) \qquad Q \xrightarrow[\ell_1]{\overline{M}(\lambda)} \nu\vec{w}.\left(\{^N/_\lambda\} \mid Q'\right) \qquad \vec{w}\#P \qquad \vec{y}\#Q,\vec{w}}{P \mid Q \xrightarrow[(0\ell_0,1\ell_1)]{\tau} \nu\vec{y},\vec{w}.\left(\mathrm{id} \mid P' \mid Q'\right)}\ \textsc{Close-R}$$

Figure 2: An early non-interleaving structural operational semantics.

The first shared trait by all equivalences for the applied π-calculus is that they make use of a *static equivalence*. Its role is to prevent the attacker from using the data they know to form a test for one process that does not hold for another process. In an extended process, one can think of the active substitution as a record of the information available to an attacker observing messages communicated on public channels. The attacker can then combine that information in various ways to try to pass a test, e.g., hashing the first message and checking whether it is equal to the second message. We find it insightful to break down static equivalence into simpler definitions, that we will employ to achieve the same effect. In particular, we start with the following satisfaction relation.

Definition 4 (satisfaction)**.** *Satisfaction* $\vDash$ *is defined inductively as:*

- $\nu x.A \vDash M = N$ *whenever, for* $y\#\nu x.A,M,N$*, we have* $A\{^y/_x\} \vDash M = N$*, and also*
- $\theta \mid P \vDash M = N$ *whenever* $M\theta =_E N\theta$*.*

The above ensures that the private names in an extended process do not appear directly in M or N, leaving only the possibility of using aliases in the domain of the active substitution in M and N to indirectly refer to private names. That is, M and N are recipes that must produce the same message, up to the equational theory E, given the information recorded in the active substitution of the extended process. As a simple example, we have $\nu x.\left(\{^{x}/_{0\lambda}\} \circ \{^{h(x)}/_{1\lambda}\} \mid P\right) \vDash h(0\lambda) = 1\lambda$.

Now we can make a generic point about all reasonable notions of equivalence based on our structural operational semantics. As explained in related work [4], each alias has a location prefix, allowing each location to have its unique pool of aliases, thus ensuring that the choice of alias is localised and not impacted by choices of aliases made by concurrent threads. For example, the following process has two transitions, labelled with $(\overline{a}(0\lambda), 0[])$ and $(\overline{b}(1\lambda), 1[])$ (cf. Def. 9 for a formal definition of those *events*):

$$\nu x.\left(\{^{x}/_{0\lambda}\} \mid 0 \mid \overline{b}\langle h(x)\rangle\right) \xleftarrow[0[]]{\overline{a}(0\lambda)} \mathrm{id} \mid \nu x.\left(\overline{a}\langle x\rangle \mid \overline{b}\langle h(x)\rangle\right) \xrightarrow[1[]]{\overline{b}(1\lambda)} \nu x.\left(\left\{^{h(x)}/_{1\lambda}\right\} \mid \overline{a}\langle x\rangle \mid 0\right)$$

Clearly, any reasonable semantics should equate the above process with the one below, where the only difference is that the parallel processes $\overline{a}\langle x\rangle$ and $\overline{b}\langle h(x)\rangle$ have been permuted (e.g., exchanged their locations).

$$\nu x.\left(\{^{x}/_{1\lambda}\} \mid \overline{b}\langle h(x)\rangle \mid 0\right) \xleftarrow[1[]]{\overline{a}(1\lambda)} \mathrm{id} \mid \nu x.\left(\overline{b}\langle h(x)\rangle \mid \overline{a}\langle x\rangle\right) \xrightarrow[0[]]{\overline{b}(0\lambda)} \nu x.\left(\left\{^{h(x)}/_{0\lambda}\right\} \mid 0 \mid \overline{a}\langle x\rangle\right)$$

Notice that the events labelling the transitions differ only in the prefix string 0 or 1, but that this change impacts the domain of the active substitutions. Therefore, when defining any notion of equivalence using this operational semantics, we must keep track of a substitution between aliases (which should be a bijection), thereby allowing for differences in prefixes and making the particular choice of alias irrelevant when performing equivalence checking.

Definition 5 (alias substitution)**.** *Alias substitutions ρ extend to labels such that $(MN)\rho = M\rho N\rho$ and $(M(\alpha))\rho = M\rho(\alpha\rho)$, and $\tau\rho = \tau$.*

The following function is just a convenience to pick out the domain of an active substitution. This is useful since the domain remembers the set of aliases that have already been extruded.

Definition 6. *We extend the domain function to extended processes such that* $\mathrm{dom}(\nu\vec{x}.(\theta \mid A)) = \mathrm{dom}(\theta)$.

We make use of aliases substitution even for interleaving equivalences and similarities. For example, the following[1] defines a notion of interleaving "presimilarity" (a term coined here to distinguish it from "similarity", introduced in Def. 8) that disregards the locations but requires the aliases to be substituted.

Definition 7 (interleaving presimilarity)**.** *Let $\mathscr{R}$ be a relation between pairs of extended processes and ρ be an alias substitution. We say $\mathscr{R}$ is an i-presimulation whenever if $A\ \mathscr{R}^{\rho}\ B$, then:*

- *If $A \xrightarrow[u]{\pi} A'$ then there exists ρ', B', u', π' s.t. $\rho\restriction_{\mathrm{dom}(A)} = \rho'\restriction_{\mathrm{dom}(A)}$, $B \xrightarrow[u']{\pi'} B'$, $\pi\rho' = \pi'$ and $A'\ \mathscr{R}^{\rho'}\ B'$.*
- *If $A \vDash M = N$, then $B \vDash M\rho = N\rho$.*

We say process P i-presimulates Q, and write $P \sqsubseteq_i Q$, whenever there exists a i-presimulation $\mathscr{R}$ such that $\mathrm{id} \mid P\ \mathscr{R}^{\mathrm{id}}\ \mathrm{id} \mid Q$.

Notice that i-presimilarity $\sqsubseteq_i$ is defined on processes: defining it on extended processes A and B require bijective alias substitutions ρ such that $\mathrm{dom}(A)\rho = \mathrm{dom}(B)$ that complicate later definitions.

Now consider again the processes examined above $\nu x.\left(\overline{a}\langle x\rangle \mid \overline{b}\langle h(x)\rangle\right)$ and $\nu x.\left(\overline{b}\langle h(x)\rangle \mid \overline{a}\langle x\rangle\right)$. They are mutually i-presimilar, i.e., there exist two i-presimulations that relate them in each direction. These

[1] We color what we want to stress or the "diff" with the previous definition or a definition indicated in footnote.

presimulations involve building up a bijection on aliases ρ such that $\rho : 0\lambda \mapsto 1\lambda$ and $\rho : 1\lambda \mapsto 0\lambda$. By applying this bijection to the labels of each of the transitions presented above, indeed the actions of both processes, $\overline{a}(0\lambda)$ and $\overline{a}(1\lambda)$ map to each other. Observe also that the final states these processes reach are $A = \nu x.\left(\{^{x}/_{0\lambda}\} \circ \{^{h(x)}/_{1\lambda}\} \mid 0 \mid 0\right)$ and $B = \nu x.\left(\{^{x}/_{1\lambda}\} \circ \{^{h(x)}/_{0\lambda}\} \mid 0 \mid 0\right)$. Since $A \vDash h(0\lambda) = 1\lambda$, we also want this test to be satisfied by B, modulo the alias substitution ρ that has been built by the presimilarity, i.e., $B \vDash (h(0\lambda))\rho = (1\lambda)\rho$, which indeed holds. Notice that it is necessary to apply ρ to the messages when checking that equality tests are preserved, and that it must be applied before the active substitution.

One may ask whether it is possible to simply have a permutation of location prefixes, keeping alias variables the same. Such an approach would not be sufficiently flexible to capture relations such as

$$\nu x.\left(\overline{b}\langle h(x)\rangle.\overline{a}\langle x\rangle\right) \sqsubseteq_i \nu x.\left(\overline{b}\langle h(x)\rangle \mid \overline{a}\langle x\rangle\right) \qquad \text{and} \qquad \nu x.(\overline{a}\langle x\rangle \mid \overline{x}\langle h(x)\rangle) \sqsubseteq_i \nu x.(\overline{a}\langle x\rangle.\overline{x}\langle h(x)\rangle).$$

In both examples, on one side there are two locations, and on the other there is only one location. This helps explain why we employ a bijection between aliases and not only between locations.

The above definition is an aesthetic preorder in that we always match a positive test on the left with a positive test on right. The clause concerning equality tests effectively defines "static implication" proposed in related work on applied process calculi [25]. However, there is a small gap compared to the standard simulation we expect for the π-calculus. Indeed, the definition of presimilarity lets the following hold:

$$\nu y.(\overline{a}\langle x\rangle + \overline{a}\langle y\rangle) \sqsubseteq_i \overline{a}\langle x\rangle$$

Therefore the above processes are mutually presimilar, since the other direction holds trivially. The reason the above relation holds is that there is no equality that can distinguish the message x from the private name y. That is, both

$$\mathrm{id} \mid \nu y.(\overline{a}\langle x\rangle + \overline{a}\langle y\rangle) \xrightarrow[[0]]{\overline{a}(\lambda)} \nu y.(\{^{x}/_{\lambda}\} \mid 0) \qquad \text{and} \qquad \mathrm{id} \mid \nu y.(\overline{a}\langle x\rangle + \overline{a}\langle y\rangle) \xrightarrow[[1]]{\overline{a}(\lambda)} \nu y.(\{^{y}/_{\lambda}\} \mid 0)$$

can only be matched by $\mathrm{id} \mid \overline{a}\langle x\rangle \xrightarrow[[]]{\overline{a}(\lambda)} \{^{x}/_{\lambda}\} \mid 0$, and there is no M and N such that $\nu y.(\{^{y}/_{\lambda}\} \mid 0) \vDash M = N$ and $\{^{x}/_{\lambda}\} \mid 0 \nvDash M = N$. Notice this is despite the fact that $\{^{x}/_{\lambda}\} \mid 0 \vDash \lambda = x$, but $\nu y.(\{^{y}/_{\lambda}\} \mid 0) \nvDash \lambda = x$, which would amount to $\nu y.(\{^{y}/_{\lambda}\} \mid 0)$ satisfying the inequality $\lambda \neq x$; hence such negative distinguishing tests are not picked up on by presimilarity.

Intuitively, one can think of the above example modelling, with the left process, an "unreliable" channel (i.e., output on channel $\overline{a}$ can either be the intended message x or anything else as y); whereas the right process is a reliable channel where the receiver would always get the intended message x. Since we expect that in a conservative extension of the π-calculus the above processes can be distinguished, we strengthen presimilarity to obtain "similarity". This strengthening amounts to demanding static equivalence, even when considering similarity preorders.

Definition 8 (interleaving similarity)**.** *Let $\mathscr{R}$ be a relation between pairs of extended processes and ρ be an alias substitution. We say $\mathscr{R}$ is a i-simulation whenever if $A\ \mathscr{R}^{\rho}\ B$, then:*

- *If $A \xrightarrow[u]{\pi} A'$ then there exists ρ', B', u', π' s.t. $\rho\restriction_{\mathrm{dom}(A)} = \rho'\restriction_{\mathrm{dom}(A)}$, $B \xrightarrow[u']{\pi'} B'$, $\pi\rho' = \pi'$ and $A'\ \mathscr{R}^{\rho'}\ B'$.*
- *$A \vDash M = N$ iff $B \vDash M\rho = N\rho$.*

We say process P i-simulates Q, and write $P \preceq_i Q$, whenever there exists an i-simulation $\mathscr{R}$ such that $\mathrm{id} \mid P\ \mathscr{R}^{\mathrm{id}}\ \mathrm{id} \mid Q$. If in addition the relation is symmetric, e.g., $A\ \mathscr{R}^{\rho}\ B$ iff $B\ \mathscr{R}^{\rho^{-1}}\ A$, then P and Q are i-bisimilar, written $P \sim_i Q$.

The notions of bisimilarity obtained from presimilarity and similarity concide, hence we see similarity as presimilarity with a little of the power of bisimilarity for equating tests. Note that $\nu y.(\overline{a}\langle x\rangle + \overline{a}\langle y\rangle)$ and $\overline{a}\langle x\rangle$ are not i-similar, since there is a $\overline{a}(\lambda)$-transition after which only the right side satisfies $\lambda = x$.

Definitions in related work on the applied π-calculus do not require an alias substitution, as in the definition above. Those papers [1, 21] allow the alias to be freely chosen, without indicating the location. Notice also the location under the labelled transition is never used in these interleaving semantics. The located aliases and location labels are however important for our non-interleaving equivalences, and for concurrency diamonds required to extend techniques such as POR to the full applied π-calculus.

4 Using LATS to define semantics preserving duration or causality

We now make the transition from interleaving to non-intereaving semantics. The border between interleaving and non-interleaving semantics was heavily debated in the early 1990's. A common argument at the time was that problems concerning non-interleaving semantics could be reduced to a problem in terms of an interleaving semantics, since processes such as $\nu x.(\overline{a}\langle x\rangle \mid \overline{a}\langle x\rangle)$ and $\nu x.(\overline{a}\langle x\rangle.\overline{a}\langle x\rangle)$ could be distinguished by splitting each output actions into a "begin output" and "end output" action and then considering the interleavings. This view was eventually dispelled by van Glabbeek and Vaandrager [18] (based on works, such as [3, 19, 32]), who showed that, no matter how many times actions are split, one cannot obtain an interleaving semantics that preserves desirable properties of a non-interleaving semantics.

Their key example, translated here to the π-calculus, is that there is an interleaving simulation relating the following processes.

$$\nu c,d.\big(\big(\overline{d}\langle d\rangle \mid \nu n.\overline{a}\langle n\rangle.d(z).n(x)\big) \mid (\overline{c}\langle c\rangle \mid c(y))\big) \preceq_i \nu c,d.\big(\big(\overline{d}\langle d\rangle \mid \nu n.\overline{a}\langle n\rangle.d(z)\big) \mid (\overline{c}\langle c\rangle \mid c(y).n(x))\big) \quad (1)$$

Furthermore, even if we were to enhance similarity with the power to split actions, these processes would still be related. What is happening here is that when a τ-transition both starts and terminates while another τ-transition is running, the end of the longer and shorter τ-transition can be swapped, resulting in a behaviour that can be simulated on the right. Such "swapping" semantics were investigated by Vogler [32], when investigating the coarsest language theory robust against splitting.

Although the above example preserves event splitting, allowing it to hold can be considered problematic since we confuse the beginning and end of two distinct events that happen to be labelled in the same way. A notion of similarity allowing the above example to hold, neither preserves the duration of events, nor the causal dependencies between events. To see why, observe that the process on the left above has a τ-transition that can start before any other event and terminate after all events have finished, but there is no τ-transition on the right that can match that timing history. In this section, we lift two truly non-interleaving semantics (ST and HP) to the applied π-calculus that do preserve such properties.

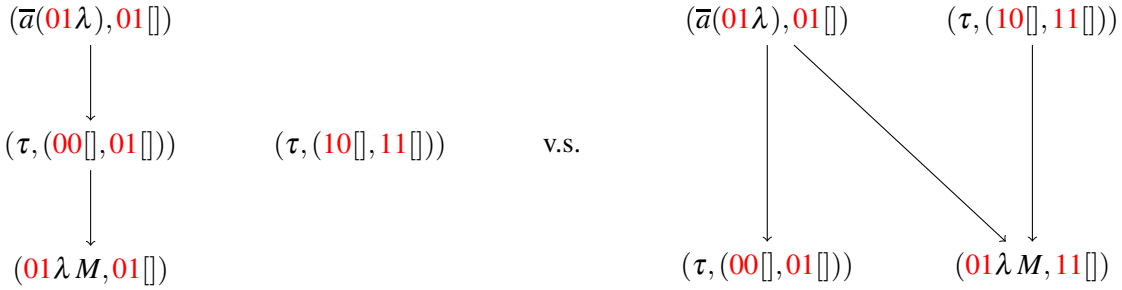
4.1 Independence and permutations of events

To define non-interleaving equivalences we make use of independence relations. Structural independence, that looks only at the locations, is sufficient for calculi such as CCS. However, for the π-calculus and its extensions, in addition, so called *link causality* should be accounted for to determine whether an output must occur first before a subsequent event occurs.

Definition 9 (independence)**.** *Define $\mathscr{L}oc$ a function on location labels (Def. 3) such that $\mathscr{L}oc(\ell) = \{\ell\}$ and $\mathscr{L}oc(\ell_0,\ell_1) = \{\ell_0,\ell_1\}$. The structural independence relation I_ℓ on location labels is the least relation*

defined by $u_0\ I_\ell\ u_1$ *whenever for all locations* $\ell_0 \in \mathscr{L}oc(u_0)$ *and* $\ell_1 \in \mathscr{L}oc(u_1)$*, there exist a string* $s \in \{0,1\}^*$ *and locations* ℓ'_0, ℓ'_1*, such that either:* $\ell_0 = s0\ell'_0$ *and* $\ell_1 = s1\ell'_1$*; or* $\ell_0 = s1\ell'_0$ *and* $\ell_1 = s0\ell'_1$*. Events* (π, u) *are pairs of action labels* π *and location labels* u*. The independence relation* $\frown$ *on events is the least symmetric relation such that* $(\pi_0, u_0) \frown (\pi_1, u_1)$ *whenever* $u_0\ I_\ell\ u_1$ *and if* $\pi_0 = \overline{M}(\alpha)$*, then* $\alpha \# \pi_1$*.*

Consider again Eq. 1, where we present its executions as a graph where the events are nodes and edges represent dependencies (i.e., the absence of independence). Note M is any message such that $\mathrm{fa}(M) \subseteq \{01\lambda\}$, and results from an input.



On the left above, observe that the rightmost τ-transition is independent from all other transitions, while all other events in that diagram are dependent on each other. In contrast, on the right above, both τ-transitions are dependent on only one other event, and independent of the others. In what follows, we make precise what it means for the processes producing these events to be incomparable.

4.2 ST-similarity and ST-bisimilarity, preserving duration

We define now ST semantics that preserve the duration of events, abstractly, without explicit time, by providing mechanisms for modelling the start and termination of events. To avoid confusion about which event terminates at a particular moment, definitions of ST equivalences make use of a device to pair events that started at the same moment, which is done by a relation over events in this work. We define some simple auxiliary functions to work with relations and sets of events.

Definition 10 (auxiliary functions)**.** *Given a relation over events* S*, we write* $\mathrm{dom}(\mathsf{S})$ *and* $\mathrm{ran}(\mathsf{S})$ *the sets of events forming the domain and range of* S*, respectively. Given an event* e *and set of events* E *we write* $e \frown E$ *whenever for all* $e' \in E$ *we have* $e \frown e'$*.*

Our definition of ST-similarity below enhances the definition of interleaving similarity such that we not only preserve the transitions, but also respect the fact that some events may have started already and are running concurrently with the new event. This is captured by ensuring that we only consider a transition labelled with event (π, u) if the condition $(\pi, u) \frown \mathrm{dom}(\mathsf{S})$ holds, which ensures that all events currently running in S are independent of (π, u). We then demand that the corresponding transition, labelled with (π', u'), is also independent of all events currently started, which is ensured by the condition $(\pi', u') \frown \mathrm{ran}(\mathsf{S})$. Notice that the relation on events strongly associate (π, u) and (π', u'), and thus, when we appeal to the second clause below they will be removed from the relation simultaneously.[2] This models the termination of the events. Thus we only record in relation S those events that are concurrently running now, which is suited to our independence relation that is only well-defined on transitions enabled in the same state or subsequent states.

Definition 11 (ST-similarity)**.** *Let* $\mathscr{R}$ *be a relation between pairs of extended processes,* ρ *be an alias substitution, and* S *be a relation over events. We say* $\mathscr{R}$ *is an ST-simulation whenever if* $A\ \mathscr{R}^{\rho,\mathsf{S}}\ B$*, then:*

[2] Using a relation has the same effect as employing a bijection between the labels of events in other formulations of ST-bisimilarity [15, p. 14].

- *If* $A \xrightarrow[u]{\pi} A'$ *and* $(\pi,u) \smile \mathrm{dom}(\mathsf{S})$ *then there exists* ρ', B', u', *and* π' *s.t.* $\rho\restriction_{\mathrm{dom}(A)} = \rho'\restriction_{\mathrm{dom}(A)}$, $B \xrightarrow[u']{\pi'} B'$, $\pi\rho' = \pi'$, $(\pi',u') \smile \mathrm{ran}(\mathsf{S})$, *and* $A' \mathscr{R}^{\rho',\mathsf{S}\cup\{((\pi,u),(\pi',u'))\}} B'$.
- *If* $\mathsf{S}'\subseteq\mathsf{S}$ *then* $A \mathscr{R}^{\rho,\mathsf{S}'} B$.
- $A \vDash M = N$ *iff* $B \vDash M\rho = N\rho$.

We say process P ST-simulates Q, and write $P \preceq_{ST} Q$, whenever there exists a ST-simulation $\mathscr{R}$ s.t. $\mathrm{id} \mid P \mathscr{R}^{\mathrm{id},\emptyset} \mathrm{id} \mid Q$. *If in addition $\mathscr{R}$ is symmetric, e.g., $A \mathscr{R}^{\rho,\mathsf{S}} B$ iff $B \mathscr{R}^{\rho^{-1},\mathsf{S}^{-1}} A$, then P and Q are ST-bisimilar, written $P \sim_{ST} Q$.*

Consider the following, which are i-bisimilar, but can be distinguished by ST-similarity.

$$\nu x.\overline{a}\langle x\rangle \mid \nu x.\overline{a}\langle x\rangle \not\preceq_{ST} \nu x.\overline{a}\langle x\rangle.\nu x.\overline{a}\langle x\rangle$$

To see why the above does not hold, observe that two events can be concurrently started on the left, but the second cannot be matched on the right. That is, when playing the ST-simulation game, we reach the following states, where $\rho : 0\lambda \mapsto \lambda'$ and $(\overline{a}(0\lambda), 0[]) \ \mathsf{S}\ (\overline{a}(\lambda'), [])$.

$$\nu y.(\{^{y}/_{0\lambda}\} \mid 0 \mid \nu x.\overline{a}\langle x\rangle) \ R^{\rho,\mathsf{S}} \ \nu y.(\{^{y}/_{\lambda'}\} \mid \nu x.\overline{a}\langle x\rangle)$$

Now observe that the extended process on the left can perform an event $(\overline{a}(1\lambda), 1[])$ independent of $\mathrm{dom}(\mathsf{S})$, but the process on the right cannot perform any action independent of $\mathrm{ran}(\mathsf{S})$. From this we conclude that the above processes cannot be related by any ST-simulation.

We still however obtain many relations that also hold according to interleaving semantics. For example, observe that the following holds.

$$\nu x,y,z.(\overline{a}\langle x\rangle.(\overline{b}\langle y\rangle \mid \overline{c}\langle z\rangle)) \preceq_{ST} \nu x,y,z.(\overline{a}\langle x\rangle.\overline{b}\langle y\rangle \mid \overline{c}\langle z\rangle). \tag{2}$$

Indeed, the left term's only transition

$$\mathrm{id} \mid \nu x,y,z.(\overline{a}\langle x\rangle.(\overline{b}\langle y\rangle \mid \overline{c}\langle z\rangle)) \xrightarrow[[]]{\overline{a}(\lambda)} \nu x,y,z.(\{^{x}/_{\lambda}\} \mid \overline{b}\langle y\rangle \mid \overline{c}\langle z\rangle)$$

can easily be matched by the right term

$$\mathrm{id} \mid \nu x,y,z.(\overline{a}\langle x\rangle.\overline{b}\langle y\rangle \mid \overline{c}\langle z\rangle) \xrightarrow[0[]]{\overline{a}(0\lambda)} \nu x,y,z.(\{^{x}/_{0\lambda}\} \mid \overline{b}\langle y\rangle \mid \overline{c}\langle z\rangle)$$

and $\rho : \lambda \mapsto 0\lambda$, $\mathsf{S} = \{((\overline{a}(\lambda), []), (\overline{a}(0\lambda), 0[]))\}$ satisfies our definition. Then, one needs to show that the resulting two terms are in $\mathscr{R}^{\rho',\mathsf{S}}$ *and* $\mathscr{R}^{\rho',\emptyset}$. For $\mathscr{R}^{\rho',\mathsf{S}}$, since $\nu x,y,z.(\{^{x}/_{\lambda}\} \mid \overline{b}\langle y\rangle \mid \overline{c}\langle z\rangle)$'s only transitions (with events $(\overline{b}(0\lambda'), 0[])$ and $(\overline{c}(1\lambda'), 1[])$) are *not* independent with $\mathrm{dom}(S) = (\overline{a}(\lambda), [])$, they do not need to be matched by $\nu x,y,z.(\{^{x}/_{0\lambda}\} \mid \overline{b}\langle y\rangle \mid \overline{c}\langle z\rangle)$. For $\mathscr{R}^{\rho',\emptyset}$, it is straightforward to pair $(\overline{b}(0\lambda'), 0[])$ and $(\overline{c}(1\lambda'), 1[])$ with themselves, and to map $0\lambda'$ and $1\lambda'$ to themselves.

Interestingly, two processes that are unrelated by ST-similarity can be in the limit identified even by ST-bisimilarity. Consider for example the following.

$$\nu x.\overline{a}\langle x\rangle \mid \nu x.\overline{a}\langle x\rangle \not\preceq_{ST} \nu x.\overline{a}\langle x\rangle.\nu x.\overline{a}\langle x\rangle \quad \text{and yet} \quad !\nu x.\overline{a}\langle x\rangle \sim_{ST} !(\nu x.\overline{a}\langle x\rangle.\nu x.\overline{a}\langle x\rangle)$$

To establish the equation on the right above, we construct the relation below and prove that it is an ST-bisimulation by checking that each condition holds. Firstly, S is downward closed, since it is not required to be defined for all $i \in \phi \cup \psi$. When the right side leads, it can either start an action in a component that has not fired (in L or greater than n), or it can start a second component that is not blocked (i.e., in ϕ, such

that $(\overline{a}(1^i0\lambda), 1^i0[]) \notin \mathrm{ran}(\mathsf{S})$), either of which can be matched on the left by starting a new independent component. When the left side leads it can only fire a new component, which can be matched by starting a new component on the right. Those transitions are preserved by $\mathscr{R}^{\rho,\mathsf{S}}$; notably, there can never be more concurrently started actions on the left than there are started components on the right. Let $\mathscr{R}$ be the least symmetric relation containing the following (upto $\equiv$).

$$\nu\vec{z}.(\theta \mid Q_0 \mid \ldots (Q_m \mid !\nu x.\overline{a}\langle x\rangle)\ldots) \;\mathscr{R}^{\rho,\mathsf{S}}\; \nu\vec{y}.(\sigma \mid P_0 \mid \ldots (P_n \mid !\nu x.\overline{a}\langle x\rangle.\nu x.\overline{a}\langle x\rangle)\ldots)$$

$\{0,\ldots m\} = \chi \cup J$ with χ and J disjoint and $m \notin J$

$$Q_i = \begin{cases} 0 & \text{if } i \in \chi \\ \nu x.\overline{a}\langle x\rangle & \text{if } i \in J \end{cases} \qquad \vec{z}_i = \begin{cases} z_i & \text{if } i \in \chi \\ \varepsilon & \text{if } i \in J \end{cases} \qquad \vec{z} = \textstyle\bigcup_{i=0}^{m} \vec{z}_i$$

$$\theta_i = \begin{cases} \{^{z_i}/_{1^i0\lambda}\} & \text{if } i \in \chi \\ \mathrm{id} & \text{if } i \in J \end{cases} \qquad \theta = \textstyle\prod_{i=0}^{m} \theta_i$$

$\{0,\ldots n\} = \psi \cup \phi \cup L$ with ψ, ϕ and L disjoint and $n \notin L$

$$P_i = \begin{cases} 0 & \text{if } i \in \psi \\ \nu x.\overline{a}\langle x\rangle & \text{if } i \in \phi \\ \nu x.\overline{a}\langle x\rangle.\nu x.\overline{a}\langle x\rangle & \text{if } i \in L \end{cases} \qquad \vec{y}_i = \begin{cases} x_i, y_i & \text{if } i \in \psi \\ x_i & \text{if } i \in \phi \\ \varepsilon & \text{if } i \in L \end{cases} \qquad \vec{y} = \textstyle\bigcup_{i=0}^{n} \vec{y}_i$$

$$\sigma_i = \begin{cases} \{^{x_i}/_{1^i0\lambda}\} \circ \{^{y_i}/_{1^i0\lambda'}\} & \text{if } i \in \psi \\ \{^{x_i}/_{1^i0\lambda}\} & \text{if } i \in \phi \\ \mathrm{id} & \text{if } i \in L \end{cases} \qquad \sigma = \textstyle\prod_{i=0}^{n} \sigma_i$$

with $\rho : \mathrm{dom}(\theta) \to \mathrm{dom}(\sigma)$ any bijection such that $(1^{f(i)}0\lambda)\rho = \begin{cases} 1^i0\lambda & \text{if } i \in \phi \\ 1^i0\lambda' & \text{if } i \in \psi \end{cases}$, for

$f\colon \phi \cup \psi \to \chi$ any injection and $\begin{cases} (\overline{a}(1^{f(i)}0\lambda), 1^{f(i)}0[]) \;\mathsf{S}\; (\overline{a}(1^i0\lambda), 1^i0[]) & \text{only if } i \in \phi \\ (\overline{a}(1^{f(i)}0\lambda), 1^{f(i)}0[]) \;\mathsf{S}\; (\overline{a}(1^i0\lambda'), 1^i0[]) & \text{only if } i \in \psi \end{cases}$

4.3 History-Preserving similarity: preserving causality

Besides observing the duration of events as in ST semantics, History-Preserving semantics observe also the partial order of causal dependencies between events. We define here HP-similarity as a strengthening of our definition of ST-similarity such that we observe not only independence but also dependence, thereby, step-by-step, ensuring that exactly the same dependencies are satisfied by the events produced by both processes. Technically this is achieved in the definition below, by partitioning the relation representing concurrently started events S according to the firing event (π,u) into: S_1 consisting of events that are independent of the current event (i.e., $(\pi,u) \frown \mathrm{dom}(\mathsf{S}_1)$); S_2 consisting of those events that are not independent (i.e., $(\pi,u) \not\frown \mathrm{dom}(\mathsf{S}_2)$). Thus S_2 is the minimal set of events that must have terminated before the new event can proceed. This partitioning must be reflected by the matching transition on the right, thereby preserving both independence and dependence. Since only the independent events and the new event are retained at the next step, the relation over events always consists of independent events.

Definition 12 (HP-similarity[3])**.** *Let $\mathscr{R}$ be a relation between pairs of extended processes, ρ be an alias substitution, and S be a relation over events. We say $\mathscr{R}$ is an HP-simulation whenever if $A\ \mathscr{R}^{\rho,\mathsf{S}}\ B$, then:*

- *If $A \xrightarrow[u]{\pi} A'$, $\mathsf{S}_1 \cup \mathsf{S}_2 = \mathsf{S}$, $(\pi,u) \frown \mathrm{dom}(\mathsf{S}_1)$ and $(\pi,u) \not\frown \mathrm{dom}(\mathsf{S}_2)$, then there exists ρ', B', u', and π' s.t. $\rho\restriction_{\mathrm{dom}(A)} = \rho'\restriction_{\mathrm{dom}(A)}$, $B \xrightarrow[u']{\pi'} B'$, $\pi\rho' = \pi'$, $(\pi',u') \frown \mathrm{ran}(\mathsf{S}_1)$, $(\pi',u') \not\frown \mathrm{ran}(\mathsf{S}_2)$, and $A'\ \mathscr{R}^{\rho',\mathsf{S}_1 \cup \{((\pi,u),(\pi',u'))\}}\ B'$.*

[3] This definition is "diffed" against Def. 11. The clause "If $\mathsf{S}' \subseteq \mathsf{S}$ then $A\ \mathscr{R}^{\rho,\mathsf{S}'}\ B$." was replaced by the partitioning of events.

- $A \vDash M = N$ *iff* $B \vDash M\rho = N\rho$.

We say process P is HP-simulated by Q, and write $P \preceq_{HP} Q$, whenever there exists an HP-simulation $\mathscr{R}$ s.t. $\mathrm{id} \mid P \; \mathscr{R}^{\mathrm{id},\emptyset} \; \mathrm{id} \mid Q$. *If in addition $\mathscr{R}$ is symmetric, then P and Q are HP-bisimilar, written $P \sim_{HP} Q$.*

When we consider similarity the difference between ST-similarity and HP-similarity is clear. For example, although Eq. 2 proved the ST-similarity of the following, they are not HP-similar.

$$\nu x,y,z.(\overline{a}\langle x\rangle.(\overline{b}\langle y\rangle \mid \overline{c}\langle z\rangle)) \not\preceq_{HP} \nu x,y,z.(\overline{a}\langle x\rangle.\overline{b}\langle y\rangle \mid \overline{c}\langle z\rangle)$$

To see this, observe that when attempting to construct an HP-simulation we can reach the following pair of processes, where $\rho : \lambda \mapsto 0\lambda$ and $(\overline{a}(\lambda),[]) \; \mathsf{S} \; (\overline{a}(0\lambda),0[])$.

$$\nu x,y,z.\left(\{^{x}/_{\lambda}\} \mid \overline{b}\langle y\rangle \mid \overline{c}\langle z\rangle\right) \; \mathscr{R}^{\rho,\mathsf{S}} \; \nu x,y,z.\left(\{^{x}/_{0\lambda}\} \mid \overline{b}\langle y\rangle \mid \overline{c}\langle z\rangle\right)$$

At this moment, the left side can perform a transition on channel c that is *dependent* on $(\overline{a}(\lambda),[])$ in $\mathrm{dom}(\mathsf{S})$. Yet, although the right side can perform a transition on channel c, it cannot match the dependency, since $(\overline{c}(1\lambda),1[])$ and $(\overline{a}(0\lambda),0[])$ are independent.

When we consider bisimilarity, the gap is more subtle for finite processes. An example separating ST-bisimilarity from HP-bisimilarity is the following.

$$\nu a,b.\left((\overline{a}\langle a\rangle \mid (a(x)+b(x))) \mid \overline{c}\langle c\rangle.\overline{b}\langle b\rangle\right) \sim_{ST} \nu a.((\overline{a}\langle a\rangle \mid a(x)) \mid \overline{c}\langle c\rangle) \tag{3}$$

To see that they are unrelated by HP-similarity (hence certainly unrelated by HP-bisimilarity), observe that the two processes can perform the following transitions

$$\mathrm{id} \mid \nu a,b.\left((\overline{a}\langle a\rangle \mid (a(x)+b(x))) \mid \overline{c}\langle c\rangle.\overline{b}\langle b\rangle\right) \xrightarrow[1[]]{\overline{c}(1\lambda)} \nu a,b.\left(\{^{c}/_{1\lambda}\} \mid (\overline{a}\langle a\rangle \mid (a(x)+b(x))) \mid \overline{b}\langle b\rangle\right)$$

and

$$\mathrm{id} \mid \nu a.((\overline{a}\langle a\rangle \mid a(x)) \mid \overline{c}\langle c\rangle) \xrightarrow[1[]]{\overline{c}(1\lambda)} \nu a.(\{^{c}/_{1\lambda}\} \mid (\overline{a}\langle a\rangle \mid a(x)) \mid 0)\,.$$

The relation on events at this moment is such that $(\overline{c}(1\lambda),1[]) \; \mathsf{S} \; (\overline{c}(1\lambda),1[])$ where alises are related by the identity function. Notice now that $\nu a,b.\left(\{^{c}/_{1\lambda}\} \mid (\overline{a}\langle a\rangle \mid (a(x)+b(x))) \mid \overline{b}\langle b\rangle\right)$ can perform a transition labelled with $(\tau,(01[1],1[]))$, which is not independent from $(\overline{c}(1\lambda),1[])$; yet, although the other process can perform a τ-transition, it cannot match the dependency constraints. In contrast, since ST-similarity would not require dependency constraints to be matched, a matching τ-transition can be performed at the corresponding point in any ST-bisimulation game.

The distinction between ST and HP is less subtle when we consider replicated processes. Consider

$$!(\nu x.\overline{a}\langle x\rangle.\nu x.\overline{a}\langle x\rangle) \not\preceq_{HP} !(\nu x.\overline{a}\langle x\rangle) \qquad \text{and yet} \qquad !(\nu x.\overline{a}\langle x\rangle.\nu x.\overline{a}\langle x\rangle) \sim_{ST} !(\nu x.\overline{a}\langle x\rangle).$$

The latter relation above we have already established previously, p. 12. Now we attempt to construct an HP-simulation containing the relation on the left. Observe that a possible first transition can be matched by both processes as follows.

$$\mathrm{id} \mid !(\nu x.\overline{a}\langle x\rangle.\nu x.\overline{a}\langle x\rangle) \xrightarrow[0[]]{\overline{a}(0\lambda)} \nu y.(\{^{y}/_{0\lambda}\} \mid \nu x.\overline{a}\langle x\rangle \mid !(\nu x.\overline{a}\langle x\rangle.\nu x.\overline{a}\langle x\rangle))$$

$$\mathrm{id} \mid !(\nu x.\overline{a}\langle x\rangle) \xrightarrow[1^{n}0[]]{\overline{a}(1^{n}0\lambda)} \nu y.(\{^{y}/_{1^{n}0\lambda}\} \mid 0 \mid (\nu x.\overline{a}\langle x\rangle \ldots (\nu x.\overline{a}\langle x\rangle \mid !\nu x.\overline{a}\langle x\rangle)))$$

At this point we have $(\overline{a}(0\lambda), 0[]) \mathsf{S} (\overline{a}(1^n 0\lambda), 1^n 0[])$ and aliases substitution such that $\rho : 0\lambda \mapsto 1^n 0\lambda$. Then, $\nu y.(\{^y/_{0\lambda}\} \mid \nu x.\overline{a}\langle x\rangle \mid !(\nu x.\overline{a}\langle x\rangle.\nu x.\overline{a}\langle x\rangle))$ can perform an event $(\overline{a}(0\lambda'), 0[])$ that is *not independent* of $(\overline{a}(0\lambda), 0[])$, but the other process can only perform an independent transition, violating the condition of HP-similarity that the transition on the right must have the same dependencies.

Similarly, we have $!\nu x, y.\left(\overline{a}\langle x\rangle.\overline{b}\langle y\rangle + \overline{b}\langle y\rangle.\overline{a}\langle x\rangle\right) \not\sim_{HP} !\nu x.\overline{a}\langle x\rangle \mid !\nu x.\overline{b}\langle x\rangle$ which are equated by the ST similarity. We interpret these kinds of examples as follows. From the perspective of the ST-semantics, executing the processes in an interleaved manner on one server that can be duplicated is the same as executing them on two servers that can be duplicated. This is because the same duration of events can be achieved by both, and in some settings this may be the desirable effect. However, this comes at the cost of a loss of awareness in the number of servers required (seen as resources), and of a sense of partition tolerance, since the right process needs up to half as much servers as the left process requires to complete the same task. This can be problematic if an attacker has the power to partition a system, e.g., by DDoS on a connection link, thereby isolating a small number of servers from the rest. In that situation, the difference picked out by HP-similarity becomes evident, and one can notice moreover that HP-similarity behaves the same in the finite case and in the limit.

There is related work on "causal" bisimilarity for the π-calculus [5], which is strictly finer than HP-bisimilarity. This is because causal bisimilarity only accounts for structural causality and not for link causality. Thus, for example although $\nu n.(\overline{a}\langle n\rangle \mid n(x)) \sim_{HP} \nu n.(\overline{a}\langle n\rangle.n(x))$ holds, these processes are distinguished by causal bisimilarity, because "there is both a subject and an object dependency between the actions [in the former], whereas in [the latter] there is only an object dependency" [5, p. 387].

4.4 Discussion on ST and HP in the context of privacy

We now revisit the essence of a privacy problem in the literature [14, 21]. The following compares two systems containing a process ready to respond to a message sent using a one-time key k, i.e., there is only one input action capable of responding to that key. The left process allows processes in distinct locations to send a message using k, while on the right there is only one location with that capability. Letting $P_{\text{ok}} \triangleq b(x).[\mathtt{snd}(\mathtt{dec}(x,k)) = \text{hi}]\overline{a}\langle\{\text{ok}\}_k\rangle$, we have :

$$\nu k.\Big((\nu r.\overline{a}\langle\{r,\text{hi}\}_k\rangle \mid (\nu m.\overline{a}\langle m\rangle + \nu r.\overline{a}\langle\{r,\text{hi}\}_k\rangle)) \mid P_{\text{ok}}\Big) \not\simeq_i \nu k.\Big((\nu r.\overline{a}\langle\{r,\text{hi}\}_k\rangle \mid \nu m.\overline{a}\langle m\rangle) \mid P_{\text{ok}}\Big)$$

The above processes are trace equivalent, yet these processes are distinguished by interleaving similarity as indicated above. Note that we assume a standard symmetic key Dolev-Yao equational theory E such that $\mathtt{dec}(\{M\}_K, K) =_E M$, $\mathtt{fst}(\langle M,N\rangle) =_E M$ and $\mathtt{snd}(\langle M,N\rangle) =_E N$.

Now compare this example above with the example below, where we essentially replicate some of the processes, and notice that, by doing so, these processes become i-bisimilar—they are even ST-bisimilar.

$$\nu k.\Big((!\nu r.\overline{a}\langle\{r,\text{hi}\}_k\rangle \mid !\nu m.\overline{a}\langle m\rangle) \mid P_{\text{ok}}\Big) \underset{\not\simeq_{HP}}{\overset{\sim_{ST}}{}} \nu k.\Big((\nu r.\overline{a}\langle\{r,\text{hi}\}_k\rangle \mid !\nu m.\overline{a}\langle m\rangle) \mid P_{\text{ok}}\Big) \tag{4}$$

The problem is that there is no way for an observer to tell the difference between the output on channel a after the match and a parallel random output on channel a (in the finite case all such parallel actions can be used up before performing the input, so it becomes clear whether or not $\{\text{ok}\}_k$ is triggered, even without the attacker being able to read the message). Of course, creating a channel for each process can be a solution to this modelling problem [21]. But the question we ask here is different: *is the difference in locations picked up only by non-interleaving semantics?*.

The fact that the processes in Eq. 4 are ST-bisimilar shows that observing differences in the duration of events does not affect the problem. Indeed, while the output $\{\text{ok}\}_k$ can only occur after the input,

there is always another parallel action indistinguishable to the attacker ready to fire for the same duration. Therefore ST-bisimilarity is not distinguishing sufficiently the localities for this problem.

In contrast to the above, HP-similarity can detect the difference in localities. This is because $\{\mathrm{ok}\}_k$ is triggered after the input, and HP-similarity ensures that the same dependencies are preserved on the right hand side of the simulation.

This problem is encapsulated by the following ST-bisimilar, but not mutually HP-similar, processes:

$$!\nu n.\overline{a}\langle n\rangle \mid b(x).\nu n.\overline{a}\langle h(n)\rangle \overset{\sim ST}{\not\sim HP} !\nu n.\overline{a}\langle n\rangle \mid b(x)$$

Hence, HP semantics is better at preserving structure, since we know that there is a success message (represented by $\{\mathrm{ok}\}_k$ here) caused by the input action, while ST semantics confuses this with other indistinguishable messages on channel a.

4.5 Failure semantics

Considering simulations, not only bisimulation, allows to explore more of the linear-time/branching-time spectrum. For example, we can define ST failure similarity [2], which extends ST-similarity such that if an action is enabled by the process on the right, then it should be enabled on the left.

Definition 13 (STf-similarity[4])**.** *Let $\mathscr{R}$ be a relation between pairs of extended processes, ρ be an alias substitution, and S be a relation over events. We say $\mathscr{R}$ is an STf-simulation whenever if $A\ \mathscr{R}^{\rho,\mathsf{S}}\ B$, then:*

- *If $A \xrightarrow[u]{\pi} A'$ and $(\pi,u) \smile \mathrm{dom}(\mathsf{S})$ then there exists ρ', B', u', and π' s.t. $\rho\restriction_{\mathrm{dom}(A)} = \rho'\restriction_{\mathrm{dom}(A)}$, $B \xrightarrow[u']{\pi'} B'$, $\pi\rho' = \pi'$, $(\pi',u') \smile \mathrm{ran}(\mathsf{S})$, and $A'\ \mathscr{R}^{\rho',\mathsf{S}\cup\{((\pi,u),(\pi',u'))\}}\ B'$.*
- *If $B \xrightarrow[u']{\pi'} B'$ and $(\pi',u') \smile \mathrm{ran}(\mathsf{S})$ then there exists ρ', A', u and π s.t. $\rho\restriction_{\mathrm{dom}(A)} = \rho'\restriction_{\mathrm{dom}(A)}$, $A \xrightarrow[u]{\pi} A'$, $\pi\rho' = \pi'$, and $(\pi,u) \smile \mathrm{dom}(\mathsf{S})$.*
- *If $\mathsf{S}'\subseteq\mathsf{S}$ then $A\ \mathscr{R}^{\rho,\mathsf{S}'}\ B$.*
- *$A \vDash M = N$ iff $B \vDash M\rho = N\rho$.*

We say process P is STf-simulated by Q, and write $P \preceq_{STf} Q$, whenever there exists an STf-simulation $\mathscr{R}$ such that $\mathrm{id} \mid P\ \mathscr{R}^{\mathrm{id},\emptyset}\ \mathrm{id} \mid Q$.

Tantalisingly, the above definition appears to preserve more dependencies than ST-similarity. Not only can we detect differences in the branching structure, as expected for interleaving failure similarity, but we can also detect the differences in the independence structure. For instance we have:

$$\nu x,y.(\overline{a}\langle x\rangle.\overline{a}\langle y\rangle) \not\preceq_{STf} \nu x,y.(\overline{a}\langle x\rangle \mid \overline{a}\langle y\rangle)$$

The distinguishing strategy is as follows. Both processes are free to perform the first output on a to reach the following indexed pair.

$$\rho\colon \lambda \mapsto 0\lambda \quad (\overline{a}(\lambda),[])\ \mathsf{S}\ (\overline{a}(0\lambda),0[]), \quad \nu x,y.(\{^{x}/_{\lambda}\} \mid \overline{a}\langle y\rangle)\ \mathscr{R}^{\mathrm{id},\mathsf{S}}\ \nu x,y.(\{^{x}/_{0\lambda}\} \mid 0 \mid \overline{a}\langle y\rangle)$$

At this moment, the right hand side can perform a transition labelled with event $(\overline{a}(1\lambda),1[])$, since that event is independent of $(\overline{a}(0\lambda),0[])$; yet the process on the left cannot match this event. Stated

[4]This definition is "diffed" against Def. 11.

otherwise, the process on the left fails to perform the next output on a while the other output on a is still being performed, but the process on the right can. This represents a failure measurable by observing the concurrency of events. Also, $\nu x,y,z.(\overline{a}\langle x\rangle.(\overline{b}\langle y\rangle|\,\overline{c}\langle z\rangle)) \not\preceq_{STf} \nu x,y,z.(\overline{a}\langle x\rangle.\overline{b}\langle y\rangle|\,\overline{c}\langle z\rangle)$ since an action on channel c is not enabled on the left initially.

Observing failures however does not allow us to distinguish the processes in Eq. 3 nor in Eq. 4, since they are ST-bisimilar, hence mutually STf-similar.

We now adapt our privacy-inspired example of Sect. 4.4 to show the power of failure similarity. The following are mutually ST-similar (and failure interleaving trace equivalent, which we do not define here), yet they are distinguished by STf-similarity. Letting $P_{\text{er}} \triangleq b(x).[\texttt{snd}(\texttt{dec}(x,k)) \neq \text{hi}]\overline{a}\langle\{\text{er}\}_k\rangle$:

$$\nu k.\Big((\nu r.\overline{a}\langle\{r,\text{hi}\}_k\rangle|\,(\nu m.\overline{a}\langle s\rangle + \nu r.\overline{a}\langle\{r,\text{hi}\}_k\rangle))\mid P_{\text{er}}\Big) \underset{\not\preceq_{STf}}{\preceq_{ST}} \nu k.\Big((\nu r.\overline{a}\langle\{r,\text{hi}\}_k\rangle|\,\nu m.\overline{a}\langle m\rangle)\mid P_{\text{er}}\Big)$$

The difference compared to the example of Sect. 4.4 is that we can detect whether the outputs from the two locations are the same by *not seeing an error* (er) after the input. This kind of negative testing is part of the vocabulary of failure semantics. However, similarly to Eq. 4, if we include replication then the processes become ST-bisimilar, and hence cannot be distinguished by STf-similarity.

$$\nu k.\Big((!\nu r.\overline{a}\langle\{r,\text{hi}\}_k\rangle|\,!\nu m.\overline{a}\langle m\rangle)\mid P_{\text{er}}\Big) \underset{\not\preceq_{HPf}}{\sim_{ST}} \nu k.\Big((\nu r.\overline{a}\langle\{r,\text{hi}\}_k\rangle|\,!\nu m.\overline{a}\langle m\rangle)\mid P_{\text{er}}\Big) \tag{5}$$

Despite the above processes being mutually STf-similar, they are distinguished using HPf-similarity:

Definition 14 (HPf-similarity[5])**.** *Let $\mathscr{R}$ be a relation between pairs of extended processes, ρ be an alias substitution, and S be a relation over events. We say $\mathscr{R}$ is an HPf-simulation whenever if $A\ \mathscr{R}^{\rho,\mathsf{S}}\ B$, then:*

- *If $A \xrightarrow[u]{\pi} A'$, $\mathsf{S}_1 \cup \mathsf{S}_2{=}\mathsf{S}$, $(\pi,u) \smile \text{dom}(\mathsf{S}_1)$ and $(\pi,u) \not\smile \text{dom}(\mathsf{S}_2)$ then there exists ρ', B', u', and π' s.t. $\rho\restriction_{\text{dom}(A)} = \rho'\restriction_{\text{dom}(A)}$ $B \xrightarrow[u']{\pi'} B'$, $\pi\rho' = \pi'$, $(\pi',u') \smile \text{ran}(\mathsf{S}_1)$, $(\pi',u') \not\smile \text{ran}(\mathsf{S}_2)$, and $A'\ \mathscr{R}^{\rho',\mathsf{S}_1\cup\{((\pi,u),(\pi',u'))\}}\ B'$.*
- *If $B \xrightarrow[u']{\pi'} B'$, $\mathsf{S}_1 \cup \mathsf{S}_2{=}\mathsf{S}$, $(\pi',u') \smile \text{ran}(\mathsf{S}_1)$ and $(\pi',u') \not\smile \text{ran}(\mathsf{S}_2)$ then there exists ρ', A', u and π s.t. $\rho\restriction_{\text{dom}(A)} = \rho'\restriction_{\text{dom}(A)}$ $A \xrightarrow[u]{\pi} A'$, $\pi\rho' = \pi'$, $(\pi,u) \smile \text{dom}(\mathsf{S}_1)$, and $(\pi,u) \not\smile \text{dom}(\mathsf{S}_2)$.*
- *$A \vDash M = N$ iff $B \vDash M\rho = N\rho$.*

We say process P is HPf-simulated by Q, and write $P \preceq_{HPf} Q$, whenever there exists an HPf-simulation $\mathscr{R}$ such that $\text{id} \mid P\ \mathscr{R}^{\text{id},\emptyset}\ \text{id} \mid Q$.

To see why HPf-similarity can be used to distinguish the processes in Eq. 5, observe that after inputting a message encrypted with k in two possible ways, we can tell that, on the right, in at least one case there will be an output message on channel *a that is dependent on the input*. Yet on the left it is possible, in both cases, that neither can perform such an output. An important part of this is the dependencies of the error message that we do not see, since all messages are indistinguishable to the attacker who does not know k, and hence cannot tell by looking at the message whether it is an error message.

Interestingly, anything coarser than HPf-similarity would not distinguish the processes in Eq. 5, since we use branching-time (so they are pomset failure trace equivalent[6]), failures (so they are HP-similar), *and* causality preservation (so they are ST-bisimilar): we need all the features of HPf-similarity.

[5]This definition is "diffed" against Def. 12.

[6]We do not define failure trace semantics in this paper, however it is easy to see how to obtain it via our approach to located aliases in Sect. 3 combined with classic definitions [2, 31].

5 Comparison to located bisimulations

This section compares our definitions to located equivalences, to help explain some less obvious design decisions. Early work on LATS for CCS defined a notion of bisimilarity preserving independence [24]. A key difference compared to our definition of HP-bisimilarity is that all events are accumulated in a history of events, whereas our definition remembers only those events that are currently active, and need not yet have terminated. Remembering all events may appear to simplify things, but we explain in this section that doing so gives rise to located equivalences that preserve the location of events, but forget about causal dependencies. To see this, consider the following processes, which are equivalent, even with respect to HP-bisimilarity.

$$L_1 \triangleq \nu b.\,\big(\overline{a}\langle a\rangle.\overline{b}\langle b\rangle \mid b(x).\overline{c}\langle c\rangle\big) \qquad \sim_{HP} \qquad \nu b.\,\big(\overline{b}\langle b\rangle \mid \overline{a}\langle a\rangle.b(x).\overline{c}\langle c\rangle\big) \triangleq L_2$$

To see why these processes are HP-bisimilar observe there are only three possible transitions for both processes, and one choice of alias substitution, as follows.

$$\mathrm{id} \mid L_1 \xrightarrow[0[]]{\overline{a}\langle 0\lambda\rangle} \xrightarrow[(0[],1[])]{\tau} \xrightarrow[1[]]{\overline{c}\langle 1\lambda\rangle} \qquad \mathrm{id} \mid L_2 \xrightarrow[1[]]{\overline{a}\langle 1\lambda\rangle} \xrightarrow[(0[],1[])]{\tau} \xrightarrow[1[]]{\overline{c}\langle 1\lambda'\rangle} \qquad \rho : 0\lambda \mapsto 1\lambda \qquad \rho : 1\lambda \mapsto 1\lambda'$$

There are no other transitions (modulo renaming λ, of course), and none of these events can be permuted. Notice that after each step the next transition is not independent of the currently started transitions, hence any started event must be removed from the set of active independent transitions S_1 for the game to continue. Therefore, we can pair the four states of these processes to form an HP-bisimulation.

In contrast, for the established located bisimilarities based on a LATS, the set of all events that have happened is accumulated in $\mathscr{E}$, and the independence of our LATS is preserved over all events. That is, we remember all pairs of events, and preserve independence everywhere, as captured by the following definition.

Definition 15 (I-consistent relation)**.** *For some symmetric relation over events I, an I-consistent relation over a set of events, say $\mathscr{E}$, is such that if $(e_0,d_0)\in\mathscr{E}$ and $(e_1,d_1)\in\mathscr{E}$ then $e_0\,I\,e_1$ iff $d_0\,I\,d_1$.*

The definition above can be instantiated with any notion of independence over events, such as I_ℓ or $\smile$ as in Def. 9, denoted here by I.

Now if we accumulate all pairs of events for our example above we obtain, after three transitions, the relation over events $\mathscr{E}$ defined as follows.

$$(\overline{a}\langle 0\lambda\rangle, 0[])\,\mathscr{E}\,(\overline{a}\langle 1\lambda\rangle, 1[]) \qquad (\tau,(0[],1[]))\,\mathscr{E}\,(\tau,(0[],1[])) \qquad (\overline{c}\langle 1\lambda\rangle, 1[])\,\mathscr{E}\,\big(\overline{c}\langle 1\lambda'\rangle, 1[]\big)$$

Taking the relation I to be $\smile$, we have that the above is not $\smile$-consistent, since $(\overline{a}\langle 0\lambda\rangle, 0[])\;I_\ell\;(\overline{c}\langle 1\lambda\rangle, 1[])$ holds but $(\overline{a}\langle 1\lambda\rangle, 1[])\;I_\ell\;(\overline{c}\langle 1\lambda'\rangle, 1[])$ does not.

An immediate consequence of the above is that the definition of bisimulation based on I-consistency, defined below, preserves the location of events more strongly than HP-bisimilarity, which preserves causal relationships. Indeed when we take I to be I_ℓ, obtaining I_ℓ-bisimilarity, we obtain a located bisimilarity and located bisimilarities and HP-bisimilarities are known to be incomparable.

Definition 16 (I-similarity)**.** *Let $\mathscr{R}$ be a relation between pairs of extended processes and ρ be an alias substitution. We say $\mathscr{R}$ is an I-simulation whenever if $A\,\mathscr{R}^{\rho,\mathscr{E}}\,B$, then:*

- *$\mathscr{E}$ is I-consistent.*

- *If* $A \xrightarrow[u]{\pi} A'$ *then there exists* ρ', B', u', *and* π' *s.t.* $\rho\restriction_{\mathrm{dom}(A)} = \rho'\restriction_{\mathrm{dom}(A)}$, $B \xrightarrow[u']{\pi'} B'$, $\pi\rho' = \pi'$, *and* $A' \mathscr{R}^{\rho',\mathscr{E}\cup\{((\pi,u),(\pi',u'))\}} B'$.
- $A \vDash M = N$ *iff* $B \vDash M\rho = N\rho$.

We say process P I-simulates Q, and write $P \preceq_I Q$, *whenever there exists an I-simulation* $\mathscr{R}$ *s.t.* $\mathrm{id} \mid P \mathscr{R}^{\mathrm{id},\emptyset} \mathrm{id} \mid Q$. *If in addition* $\mathscr{R}$ *is symmetric, then P and Q are I-bisimilar, written* $P \sim_I Q$.

In a sense, it is just a coincidence that for CCS, the above definition exploits nicely the independence relation of CCS, which coincides with I_ℓ since there is no link causality, and hence is strongly linked to the definition of a LATS for CCS. If we try to use $\smile$-bisimilarity, using the full independence relation $\smile$ from Def. 9, that accounts for link causality, we end up with an awkward relation. This has to do with the fact that independence for a LATS for the π-calculus must respect link causality, which means, for example, that the following processes are $\smile$-bisimilar:

$$\nu n.(\overline{a}\langle n\rangle \mid n(x)) \quad \sim_{\smile} \quad \nu n.\overline{a}\langle n\rangle.n(x)$$

This is because for both processes, the two events can only execute in one order, and neither is independent of the other, hence the set of events are $\smile$-consistent. Yet these processes are not I_ℓ-bisimilar, since their pairing S is not I_ℓ-consistent. This is rather troubling when juxtapositioned with the observation that the following are not $\smile$-bisimilar.

$$\nu n.\left(\overline{a}\langle n\rangle \mid n(x).\overline{\mathrm{ok}}\langle \mathrm{ok}\rangle\right) \quad \not\sim_{\smile} \quad \nu n.\overline{a}\langle n\rangle.n(x).\overline{\mathrm{ok}}\langle \mathrm{ok}\rangle$$

Similarly to the above we have that the three events may only be fired in a given order. However, the resulting relation over events is not $\smile$-consistent, since the first and third events are independent for the left process above, but are not independent for the right process above. This seems strange that the first event of the sub-process $n(x).\overline{\mathrm{ok}}\langle \mathrm{ok}\rangle$ is somehow not location-sensitive, yet the second is. To us, this is morally broken, hence $\sim_{\smile}$ is ill-defined. On the other hand $\sim_{I_\ell}$ consistently distinguishes these two examples, where the former involves two locations while the latter involves only one location.

Indeed $\sim_{I_\ell}$ is the notion of bisimilarity that would be obtained from the notion of trace equivalence implemented in the equivalence checking tool DeepSec [9]. They call their equivalence session equivalence and define it for a fragment of the applied π-calculus only. It is clear that a notion of trace equivalence that ensures that the events in compared traces are I_ℓ-consistent is the session equivalence of DeepSec. Intuitively, this is because session equivalence forms a bijection between processes in distinct locations and matches the behaviours in each location, which is exactly what I_ℓ-consistency would demand. Interestingly, that tool employs partial order reduction to improve equivalence checking; which is evidence that POR might be lifted to other notions of equivalence defined in this paper.

Thus, for the π-calculus and its extensions, there seems to be no real connection between $\smile$ and located bisimilarity; effectively we throw away part of the LATS to obtain a located bisimilarity [27]. The above observations help explain two things. Firstly, why we chose to target equivalences related to ST-similarity and HP-similarity rather than located bisimilarities in this work. Secondly, why our definitions are more complicated than those for located bisimilarities for CCS in the literature.

6 Conclusion

Having introduced a LATS for the applied π-calculus [4], we have shown that a world of non-interleaving operational semantics opens up for value passing process calculi. Notably, by using the independence relation (Def. 9) of a LATS, we capture ST-bisimilarity (Def. 11) and HP-bisimilarity (Def. 12) that reflect

correctly link causality, which were not preserved by established located bisimilarities for the π-calculus. Both semantics have their merits: for infinite processes, ST-semantics are very close to interleaving semantics, while being naturally compatible with the independence relation of a LATS; while HP-semantics better preserves the testing of finite subcomponents, even when we consider limits and infinite process. Eq. 4 showed that HP-similarity is able to detect attacks that are detectable using interleaving similarity for finite systems, yet are not detectable even by the strictly more powerful ST-bisimilarity when we take limits. This observation is reinforced in Eq. 5 where we show that HP failure similarity picks up on attacks that would be missed by anything coarser in any dimension (ST-bimilarity, HP-similarity, or even pomset failure traces). Since HP-bisimilarity would equally pick up on the attacks, we suggest HP-bisimilarity may be a good choice for security.

Having these definitions opens up formal and practical questions. It is non-trivial to verify that these definitions are the same as what we would expect if we pass via the more denotational world of event structures, configuration structures, or ST-structures [17, 23]. It is also non-trivial to provide characterisations using tests and modal logics [20]. What is fairly clear is that the relationship between these notions, since we start with the minimal notion of presimilarity and grow from there, providing separating examples at each step. The practical questions are more pressing, in particular, whether we can make use of ST- and HP-semantics in tools for protocol verification.

Acknowledgements The definitions in this paper are introduced to support an invited talk by the second author at EXPRESS/SOS on proving privacy properties using bisimilarity. We thank the organisers Valentina Castiglioni and Claudio Antares Mezzina for this invitation.

References

[1] Martín Abadi, Bruno Blanchet & Cédric Fournet (2018): *The Applied Pi Calculus: Mobile Values, New Names, and Secure Communication*. *J. ACM* 65(1), pp. 1:1–1:41, doi:10.1145/3127586.

[2] Luca Aceto & Uffe Engberg (1991): *Failures semantics for a simple process language with refinement*. In Somenath Biswas & Kesav V. Nori, editors: *Foundations of Software Technology and Theoretical Computer Science*, Springer, pp. 89–108, doi:10.1007/3-540-54967-6_63.

[3] Luca Aceto & Matthew Hennessy (1994): *Adding action refinement to a finite process algebra*. *Inform. and Comput.* 115(2), pp. 179–247, doi:10.1006/inco.1994.1096.

[4] Clément Aubert, Ross Horne & Christian Johansen (2022): *Diamonds for Security: A Non-Interleaving Operational Semantics for the Applied Pi-Calculus*. In Bartek Klin, Sławomir Lasota & Anca Muscholl, editors: *33rd International Conference on Concurrency Theory*, *Leibniz International Proceedings in Informatics* 243, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, pp. 30:1–30:26, doi:10.4230/LIPIcs.CONCUR.2022.30.

[5] Michele Boreale & Davide Sangiorgi (1998): *A fully abstract semantics for causality in the π-calculus*. *Acta Inform.* 35(5), pp. 353–400, doi:10.1007/s002360050124.

[6] Gérard Boudol, Ilaria Castellani, Matthew Hennessy & Astrid Kiehn (1994): *A Theory of Processes with Localities*. *Formal Aspects Comput.* 6(2), pp. 165–200, doi:10.1007/BF01221098.

[7] Ilaria Castellani (1995): *Observing distribution in processes: static and dynamic localities*. *Int. J. Found. Comput. Sci.* 6(04), pp. 353–393, doi:10.1142/S0129054195000196.

[8] V. Cheval, R. Crubillé & S. Kremer (2022): *Symbolic Protocol Verification with Dice: Process Equivalences in the Presence of Probabilities*. In: *2022 2022 IEEE 35th Computer Security Foundations Symposium (CSF) (CSF)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 303–318, doi:10.1109/CSF54842.2022.00020.

[9] Vincent Cheval, Steve Kremer & Itsaka Rakotonirina (2019): *Exploiting Symmetries When Proving Equivalence Properties for Security Protocols*. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang & Jonathan Katz, editors: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, ACM, pp. 905–922, doi:10.1145/3319535.3354260.

[10] Silvia Crafa, Daniele Varacca & Nobuko Yoshida (2012): *Event Structure Semantics of Parallel Extrusion in the Pi-Calculus*. In Lars Birkedal, editor: *Foundations of Software Science and Computational Structures - 15th International Conference, FOSSACS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, *LNCS* 7213, Springer, pp. 225–239, doi:10.1007/978-3-642-28729-9_15.

[11] Ioana Cristescu, Jean Krivine & Daniele Varacca (2015): *Rigid Families for CCS and the π-calculus*. In Martin Leucker, Camilo Rueda & Frank D. Valencia, editors: *Theoretical Aspects of Computing - ICTAC 2015 - 12th International Colloquium Cali, Colombia, October 29-31, 2015, Proceedings*, *LNCS* 9399, Springer, pp. 223–240, doi:10.1007/978-3-319-25150-9_14.

[12] Pierpaolo Degano, Rocco De Nicola & Ugo Montanari (1989): *Partial orderings descriptions and observations of nondeterministic concurrent processes*. In J. W. de Bakker, W. P. de Roever & G. Rozenberg, editors: *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, Springer, pp. 438–466, doi:10.1007/BFb0013030.

[13] Yuxin Deng, Matthew Hennessy, Rob van Glabbeek & Carroll Morgan (2008): *Characterising Testing Preorders for Finite Probabilistic Processes*. *Log. Methods Comput. Sci.* Volume 4, Issue 4, doi:10.2168/LMCS-4(4:4)2008.

[14] Ihor Filimonov, Ross Horne, Sjouke Mauw & Zach Smith (2019): *Breaking Unlinkability of the ICAO 9303 Standard for e-Passports Using Bisimilarity*. In Kazue Sako, Steve A. Schneider & Peter Y. A. Ryan, editors: *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I*, *LNCS* 11735, Springer, pp. 577–594, doi:10.1007/978-3-030-29959-0_28.

[15] Rob van Glabbeek (1990): *The refinement theorem for ST-bisimulation semantics*. Technical Report R 9002, Centre for Mathematics and Computer Science. Available at https://ir.cwi.nl/pub/5765.

[16] Rob van Glabbeek (2001): *The linear time-branching time spectrum I. The semantics of concrete, sequential processes*. In J. A. Bergstra, A. Ponse & S. A. Smolka, editors: *Handbook of process algebra*, Elsevier, pp. 3–99, doi:10.1016/b978-044482830-9/50019-9.

[17] Rob van Glabbeek & Gordon D. Plotkin (2009): *Configuration structures, event structures and Petri nets*. *Theor. Comput. Sci.* 410(41), pp. 4111–4159, doi:10.1016/j.tcs.2009.06.014.

[18] Rob van Glabbeek & Frits W. Vaandrager (1997): *The Difference between Splitting in n and $n+1$*. *Inf. Comput.* 136(2), pp. 109–142, doi:10.1006/inco.1997.2634.

[19] Roberto Gorrieri & Cosimo Laneve (1995): *Split and ST Bisimulation Semantics*. *Inf. Comput.* 118(2), pp. 272–288, doi:10.1006/inco.1995.1066.

[20] Matthew Hennessy (1995): *Concurrent Testing of Processes*. *Acta Informatica* 32(6), pp. 509–543, doi:10.1007/BF01178906.

[21] Ross Horne & Sjouke Mauw (2021): *Discovering ePassport Vulnerabilities using Bisimilarity*. *Log. Meth. Comput. Sci.* 17(2), p. 24, doi:10.23638/LMCS-17(2:24)2021.

[22] Ross Horne, Sjouke Mauw & Semen Yurkov (2021): *Compositional Analysis of Protocol Equivalence in the Applied π-Calculus Using Quasi-open Bisimilarity*. In Antonio Cerone & Peter Csaba Ölveczky, editors: *Theoretical Aspects of Computing - ICTAC 2021 - 18th International Colloquium, Virtual Event, Nur-Sultan, Kazakhstan, September 8-10, 2021, Proceedings*, *LNCS* 12819, Springer, pp. 235–255, doi:10.1007/978-3-030-85315-0_14.

[23] Christian Johansen (2016): *ST-structures*. *J. Log. Algebraic Methods Program.* 85(6), pp. 1201–1233, doi:10.1016/j.jlamp.2015.10.009.

[24] Madhavan Mukund & Mogens Nielsen (1992): *CCS, Location and Asynchronous Transition Systems*. In R. K. Shyamasundar, editor: *Foundations of Software Technology and Theoretical Computer Science, 12th Conference, New Delhi, India, December 18-20, 1992, Proceedings*, *LNCS* 652, Springer, pp. 328–341, doi:10.1007/3-540-56287-7_116.

[25] Joachim Parrow, Johannes Borgström, Lars-Henrik Eriksson, Ramunas Gutkovas & Tjark Weber (2021): *Modal Logics for Nominal Transition Systems*. *Log. Meth. Comput. Sci.* 17(1), pp. 6:1–6:49, doi:10.23638/LMCS-17(1:6)2021.

[26] Alexander Rabinovich & Boris Avraamovich Trakhtenbrot (1988): *Behavior Structures and Nets*. *Fund. Inform.* 11(4), pp. 357–404, doi:10.3233/FI-1988-11404.

[27] Davide Sangiorgi (1996): *Locality and interleaving semantics in calculi for mobile processes*. *Theor. Comput. Sci.* 155(1), pp. 39–83, doi:10.1016/0304-3975(95)00020-8.

[28] Davide Sangiorgi (1996): *A Theory of Bisimulation for the pi-Calculus*. *Acta Inform.* 33(1), pp. 69–97, doi:10.1007/s002360050036.

[29] Davide Sangiorgi & David Walker (2001): *On Barbed Equivalences in pi-Calculus*. In Kim Guldstrand Larsen & Mogens Nielsen, editors: *CONCUR 2001 - Concurrency Theory, 12th International Conference, Aalborg, Denmark, August 20-25, 2001, Proceedings*, *LNCS* 2154, Springer, pp. 292–304, doi:10.1007/3-540-44685-0_20.

[30] Daniele Varacca & Nobuko Yoshida (2010): *Typed event structures and the linear pi-calculus*. *Theor. Comput. Sci.* 411(19), pp. 1949–1973, doi:10.1016/j.tcs.2010.01.024.

[31] Walter Vogler (1991): *Failures semantics based on interval semiwords is a congruence for refinement*. *Distributed Computing* 4(3), pp. 139–162, doi:10.1007/BF01798961.

[32] Walter Vogler (1996): *The Limit of Split$_n$-Language Equivalence*. *Inf. Comput.* 127(1), pp. 41–61, doi:10.1006/inco.1996.0048.

From Legal Contracts to Legal Calculi: the code-driven normativity

Silvia Crafa
Dipartimento di Matematica "Tullio Levi-Civita"
Università di Padova
Italy
silvia.crafa@unipd.it

Using dedicated software to represent or enact legislation or regulation has the advantage of solving the inherent ambiguity of legal texts and enabling the automation of compliance with legal norms. On the other hand, the so-called code-driven normativity is less flexible than the legal provisions it claims to implement, and transforms the nature of legal protection, potentially reducing the capability of individual human beings to invoke legal remedies.

In this article we focus on software-based legal contracts; we illustrate the design of a legal calculus whose primitives allow a direct formalisation of contracts' normative elements (*i.e.*, permissions, prohibitions, obligations, asset transfer, judicial enforcement and openness to the external context). We show that interpreting legal contracts as interaction protocols between (untrusted) parties enables the generalisation of formal methods and tools for concurrent systems to the legal setting.

1 Code is law, really?

Ethereum's smart contracts popularised the *Code is Law* principle[1], that is the idea of relying on software code to provide unambiguous definition and automatic execution of transactions between (mutually untrusted) parties; and when in disputes, the code of the contract, which is always publicly available, shall prevail. This principle is rooted in the blockchain's dogma that trust is hardwired into intermediary transparent algorithms. On this account, several governments have recognised that smart contracts, and more generally programs operating over distributed ledgers, may indeed have legal value [8, 6, 3].

This approach encompasses the blockchain technologies, since most of the benefits of digitally encoding legally binding agreements come from the precise definition and the automatic execution of a piece of programmable software, not necessarily operating over a blockchain. Accordingly, there is an increasing trend, called *Code-Driven Law* [5], using dedicated software to represent or enact legislation or regulation. Technologies like Rules as Code [7], Catala [9] or Akoma Ntoso [2] propose to create a machine-consumable version of some types of rules issued by governments and public administrations, *e.g.*, the tax office, student grant provision or social security agency. This helps identify potential inconsistencies in regulation, reduce the complexity and the ambiguity of legal texts and support the automation of legal decisions by the code-driven enforcement of rules: instead of relying on *ex-post* enforcement by third parties (*i.e.*, courts and police), the rules hardwired into code are enforced *ex-ante*, making it very difficult for people to breach them in the first place [26].

However, transposing legal rules into technical rules is a delicate process, since the inherent ambiguity of the legal system is necessary to ensure a proper application of the law on a case-by-case basis. Regulation by code is instead always more specific and less flexible than the legal provisions it purports

[1] originally proposed by Lawrence Lessing [29]

V. Castiglioni and C. A. Mezzina (Eds): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics 2022 (EXPRESS/SOS 2022).
EPTCS 368, 2022, pp. 23–42, doi:10.4204/EPTCS.368.2

to implement, thereby giving software developers and engineers the power to embed their own interpretation of the law into the technical artefacts that they create [26]. More precisely, the process of translating parties' intentions, promises, actions, powers and prohibitions into computer code, although public and unambiguous for the machine, is problematic and does not solve the problem but moves it into another dimension ([20]). Secondly, the code-driven law is based on the the automation of compliance with preset rules: if certain conditions are met the code will self-execute whatever it was programmed to do, not leaving room for disagreement about the right way to interpret the norms. Even if the need for judicial arbitration cannot be eliminated (*e.g.* one always has the right to appeal to the court if the code adopted an incorrect tax rate), the code-driven normativity transforms the nature of legal protection potentially reducing the capability of individual human beings to invoke legal remedies [5].

As an example, the Ethereum's code-is-law dogma declined with the famous TheDAO attack [34]. Indeed, from the code-is-law perspective, a problem in the source code leading to unexpected behaviour of the smart contract, is a feature of the code and not an error. But the first hard fork of the Ethereum blockchain showed that this principle is not satisfactory in practice: when large volumes of money are at stake, no one is really willing to consider a security error in a program as part of the contract they have signed. Moreover, a less naive look nowadays leads us to state that blockchain does not hardwire trust into algorithms, but rather reassigns trust to a whole series of actors (miners, programmers, companies and foundations) who implement, manage and enable the functioning of this technological platform.

2 Form Legal Contracts to Legal Calculi

Despite the difficulties highlighted above, a sensible process of digitisation of legal texts has clear advantages. In this article we discuss a specific line of research, conducted in collaboration with Cosimo Laneve and Giovanni Sartor, focusing on a specific subset of legal documents, that is the legal contracts ([19, 21, 18] and other submitted articles). Legal contracts are defined as "those agreements that are intended to give rise to a binding legal relationship or to have some other legal effect" [35]. The principle of *freedom of form* in contracts, which is shared by modern legal systems, says that parties are free to express their agreement using the language and medium they prefer, including a programming language. Therefore, by this principle, software-based contracts may count as legal contracts. However, a contract produces the intended effects, declared by the parties, only if it is legally valid: the law may deny validity to certain clauses (*e.g.*, excessive interests rate) and/or may establish additional effects that were not stated by the parties (*e.g.*, consumer's power to withdraw from an online sale, warranties, etc.). Moreover, the contract's institutional effects are guaranteed by the possibility of activating judicial enforcements. That is, each party may start a lawsuit if she believes that the other party has failed to comply with the contract. Therefore, the assimilation of software-based contracts to legally binding contracts, or rather the double nature of digital contracts as computational mechanisms and as legal contracts, raises both legal and technological issues.

First of all, in [19] we observe that different kinds of software-based solutions can be valuable in the different phases of the lifecycle of a legal contract, which goes through negotiation, contract storage/notarizing, performance, enforcement and monitoring, possible modification and dispute resolution. Accordingly, several projects are being developed for defining code-driven legal contracts, *e.g.* [30, 17, 37, 28, 24, 11, 31]. We focus here in the problem of defining suitable programming languages to write legal contracts, since finding the suitable abstraction level for legal languages is still an open issue. Indeed, such a language should be easy-to-use and to understand for legal practitioners, but at the same time, the language should be fairly expressive, have a running environment with a precise

semantics, and possibly supply sensible analyzers.

The solution we discuss in this article is the *Stipula* programming language, whose design is based on the following main remarks:

- it is an *intermediate domain-specific language*: a core calculus more concrete than a user-friendly contract specification language, and more abstract than a full-fledged programming language. This is in line with the research approach of desugaring the high level programming language into a core *Legal Calculus* [12, 25], pivoted on few selected, concise and intelligible primitives, together with a precise formalisation. This is the case of the Catala [31] language for modelling statutes and regulations clauses, the Orlando [11] language for modelling conveyances in property law, and the Silica language [16] language for generic smart contracts;
- the basic primitives of *Stipula* has been designed to easily map the *building blocks of legal contracts* into template programs and design patterns. Therefore, the direct formalisation of normative elements (*i.e.*, permissions, prohibitions, obligations, judicial enforcement and openness to the external context) as programming patterns, increases the transparency and the understanding of the link between executable instructions and institutional-normative effects;
- a legal contract is interpreted as an *interaction protocol*, that dynamically regulates permissions, prohibitions and obligations between parties, which behave concurrently as time flows. Accordingly, the definition of *Stipula* is influenced by the theory of concurrent systems, both in the definition of the operational semantics (with a precise control of nondeterminism) and in the definition of a bisimulation-based observational equivalence, that equates contracts that are syntactically different but are legally equivalent since they exhibit the same observable normative elements;
- the language definition is *implementation-agnostic*, and can be either implemented as a centralised platform or it can be run on top of a distributed system, such as a blockchain. Implementing *Stipula* in terms of smart contracts (*e.g.*, compiling in Solidity), would bring in the advantages of a public and decentralised blockchain platform. However, digital legal contracts are more general and encompass smart contracts: they can provide benefits in terms of automatic execution and enforcement of contractual conditions, traceability, and outcome certainty even without using a blockchain. In particular, running a legal contract over a secured centralised system allows for more efficiency, energy save, additional privacy. Moreover, a controlled level of intermediation can better monitor the contract enforcement, dealing with disputes between contract's parties and carrying out judicial enforcements. A prototype centralised implementation of *Stipula* as a Java application is available in [22].

We think that, even if only a concrete implementation can properly address specific issues, studying the theory of a domain-specific legal calculus is a first interesting step, that sheds some light on the digitalisation of legal texts.

3 Stipula and the code-driven normativity

A preliminary interdisciplinary research recognised that most real legal contracts are written by combining the following basic elements:

1. the *meeting of the minds*, that involves the contract's subscribers to accept the terms of the contract, and identifies the moment when legal effects are triggered;
2. a number of *permissions*, *prohibitions* and *obligation* clauses that may dynamically change, *e.g.*, the permission to use a good until a deadline;

legal contracts	*Stipula* **contracts**
meeting of the minds	agreement primitive
permissions, prohibitions	state-aware programming
obligations	event primitive
currency and tokens	asset-aware programming
openness to the environment	intermediary pattern
judicial enforcement and exceptional behaviours	authority pattern

Figure 1: Correspondence between legal elements and *Stipula* features

3. transfer of *currency* or other *assets*, *e.g.*, the property of a physical or digital good, to be used for payments, escrows and securities;
4. the *openness* to external conditions or data, *e.g.*, a triggering condition depending on the value of a stock at a given date;
5. the possibility of activating *judicial enforcements* triggered by a dispute resolution mechanism or by a third party monitoring conditions that can be hardly digitalised, as the diligent care or the good faith.

Accordingly, the basic primitives of *Stipula* has been designed to easily map these building blocks of legal contracts into template programs and design patterns, as summarised in Figure 1. More precisely, the `agreement` construct directly encodes the meeting of the minds. Normative elements are expressed by a strictly regimented behaviour in legal contracts: permissions and empowerments correspond to the possibility of performing an action at a certain stage, prohibitions correspond to the interdiction of doing an action, while obligations are recast into commitments that are checked at a specific time limit and issue a corresponding penalty if the obligation has not been met. Moreover, to model the dynamic change of the set of normative elements according to the actions that have been done (or not), *Stipula* commits to a *state-aware programming style*, inspired by the state machine pattern widely used in smart contracts (c.f. Solidity [1] and Obsidian [4]). This technique allows one to enforce the intended behaviour by prohibiting, for instance, the invocation of a function before another specific function is called.

In order to promote an *asset-aware programming* ([33, 23, 13]), assets are a specific value type, and asset manipulation is syntactically distinguished from standard operations, to stress the fact that assets cannot be destroyed nor forged but only transferred. Contract clauses depending on external data are implemented by means of a party that takes the role of intermediary and assumes the legal responsibility of timely retrieving data from the external source agreed in the terms of the contract (see the bet contract below). The contract's intermediary need not to be a third party authority, but one of the party can assume also the role of intermediary, provided that all the others agree. This is different from relying on Oracles web services, to whom legal responsibilities can hardly be attributed. Finally, dispute resolutions, judicial enforcement of legal clauses and exceptional behaviours due, *e.g.*, to force majeure, are implemented by including in the contract a party that takes the legal responsibility of interfacing with a court or an Online Dispute Resolutions platform[2].

[2]as The European ODR platform at `https://ec.europa.eu/consumers/odr`.

```
1  stipula Subscription {
2      assets wallet
3      fields cost, deposit
4
5      agreement (Editor,Buyer) {
6          Editor, Buyer: cost, deposit
7      } ⇒ @Inactive
8
9      @Inactive Buyer : subscribe [h]
10         (h == deposit) {
11             h ⊸ wallet
12             now + 1 month ≫ @To_Pay { wallet ⊸ Editor } ⇒ @End
13     } ⇒ @To_Pay
14
15     @To_Pay Buyer : annualFee [h]
16         (h == cost) {
17             h ⊸ Editor
18             now + 1 year  ≫ @Payed {} @To_Pay
19             now + 1 year + 1 month ≫ @To_Pay { wallet ⊸ Editor } ⇒ @End
20     } ⇒ @Payed
21
22     @Payed Buyer : terminate {
23            wallet ⊸ Buyer
24     }⇒ @End
25 }
```

Listing 1: The subscription contract

We illustrate the expressivity of *Stipula* by showing the contracts for a set of archetypal acts (taken form [19]). They are simple but they represent the distinctive elements that can be found in most contracts.

3.1 Subscription contract: obligation of periodic payment

We define a simple contract representing the annual subscription to a magazine or a service. Upon subscription the buyer must pay a deposit, then she must pay the annual fee. If she has not paid within one month, the deposit is transferred to the editor. At the end of the year an event changes the status of the contract so to enable the payment of the annual fee with a maximum delay of one month. If the buyer is up to date with the payments, she can terminate the subscription and get back the deposit.

The code in Listing 1 shows that a contract is similar to a class in an OOL, containing a set of fields, a constructor and a number of functions. Contract's fields are distinguished into standard fields (`cost` and `deposit` store numbers corresponding to the fees and the deposit) and assets. The contract's asset field `wallet` is initially empty and will hold the buyer's money in escrow. The agreement (lines 5-7) is a sort of constructor for the contract: it is intended as a multiparty synchronization between the parties, *i.e.* `Editor` and `Buyer`, who have to agree about the initial values of `cost` and `deposit`. After the agreement has been reached, the contract enters into the initial state `@Inactive`.

The possible states of the contract are `@Inactive, @To_Pay, @Payed`, and the contract's functions `subscribe`, `annualFee` and `terminate` are defined so that only the buyer (who subscribed the agree-

ment) can call them, and `subscribe` can be called only once at the beginning. The parameter `h` is an amount of assets, and a pre-condition checks that it corresponds to the expected amount. The operation `h —∘ wallet` transfers the assets `h` into the constract's `wallet`, while `h —∘ Editor` moves them to the editor.

Lines 12,18 and 19 issue the events corresponding to the annual payment obligation. Line 12 and 19, schedule an event that, after one month from now, resp. form the end of the paid year, check whether the (first) annual fee has not been paid (*i.e.* the state is still `To_Pay`), and in that case transfer the deposit to the editor and terminate the contract. Line 18 issues an event that in a year's time will allow the new payment by moving the contract's state from `@Payed` back to `@To_Pay`. Finally, the buyer is allowed to terminate the subscription only if all payments are regular; accordingly, the function `terminate` can be invoked only in state `@Payed` and the deposit is refunded to the buyer.

3.2 The Digital Licensee contract: usage and purchase, dispute resolution

Let us consider a contract corresponding to a licence to access a digital service, like a software or an ebook: the digital service can be freely accessed for a while, and can be permanently bought with an explicit communication within the evaluation period (for a similar example, see [27]). The licensing contractual clauses can be described as follows:

Article 1. `Licensor` grants `Licensee` for a licence to evaluate the product and fixes (*i*) the *evaluation period* and (*ii*) the *cost* of the product if `Licensee` will bought it.

Article 2. `Licensee` will pay the product in advance; he will be reimbursed if the product will not be bought with an explicit communication within the evaluation period. The refund will be the 90% of the cost because the 10% is payed to the `Authority` (see Article 3).

Article 3. `Licensee` must not publish the results of the evaluation during the evaluation period and `Licensor` must reply within 10 hours to the queries of `Licensee` related to the product; this is supervised by `Authority` that may interrupt the licence and reimburse either `Licensor` or `Licensee` according to whom breaches this agreement.

Article 4. This license will terminate automatically at the end of the evaluation period, if the `Licensee` does not buy the product.

Compared to the previous example, the licence contract holds two different assets: an indivisible non fungible `token` providing an handle to the digital service, and a `wallet` that is a fungible asset corresponding to the amount of currency kept in custody inside the contract.

A further important feature of the contract is Article 3 that defines specific constraints about the off-line behaviour of `Licensor` and `Licensee`, that is their behaviour in the physical world. This exemplifies the very general situations where contract's violations cannot be fully monitored by the (on-line) software, *i.e.* by the platform that runs the software (either a blockchain or a centralized application), such as the publication of a post in a social network, or the leakage of a secret password, or any non-automatically verifiable contextual circumstance. The intrinsic *open nature* of legal contracts is exactly this mix of external behaviour and automatic enforcement of contract clauses by means of software. The code in Listing 2 illustrates the *Stipula* programming pattern that relies on a trusted third party, the `Authority` included in the agreement, to supervise the disputes occurring from the off-line monitoring and to provide a trusted on-line dispute resolution mechanism.

The `agreement` of Listing 2 involves three parties: `Licensor` and `Licensee`, which agree to the parameters of the contract, according to Article 1. (line 6), and `Authority`, which does not need to agree

```
stipula Licence {
  assets token, wallet
  fields cost, t_start, t_limit

  agreement (Licensor,Licensee,Authority){
    Licensor, Licensee : cost, t_start, t_limit
   } ⇒ @Inactive

  @Inactive Licensor : offerLicence [t] {
    t ⊸ token
    now + t_start ≫ @Proposal { token ⊸ Licensor } ⇒ @End
  } ⇒ @Proposal

  @Proposal Licensee : activateLicence [h]
    (h == cost){
       h ⊸ wallet
       wallet*0,1 ⊸ wallet, Authority
       uses(token,Licensee) → Licensee
       now + t_limit ≫ @Trial {
                        wallet ⊸ Licensee
                        token ⊸ Licensor
                      } ⇒ @End
  } ⇒ @Trial

  @Trial Licensee : buy {
    wallet ⊸ Licensor
    token ⊸ Licensee
  } ⇒ @End

  @Trial Authority : compensateLicensor {
    wallet ⊸ Licensor
    token ⊸ Licensor
  } ⇒@End

  @Trial Authority : compensateLicensee {
    wallet ⊸ Licensee
    token ⊸ Licensor;
  } ⇒ @End
}
```

Listing 2: The contract for a digital licence

upon the contracts' parameters, but it is important that it is involved in the agreement synchronization. By calling the function `offerLicence`, the `Licensor` transfers to the contract the token corresponding to the full access to the digital service. This transfer is necessary to implement the fact that, after the activation of the the licence (within the agreed time limit `t_start`, see the event in line 11), the licensor has the legal prohibition of preventing the access to the digital service. The `Licensee` can then call `activateLicence` together with an amount of assets equal to the fixed `cost` of the license, that is then stored in the `wallet` (line 16). In line 17 a fraction of asset is moved towards the authority as a fee, while in line 18 a personal usage code associated to the token is communicated to the `Licensee`.

Once entered in the `Trial` state, the contract can terminate in three ways: (*i*) the licensee expresses its willingness to buy the licence by calling the function `buy` which grants him the full token, or (*ii*) the time limit for the free evaluation period is reached, thus the event scheduled in line 19 refunds the licensee (but for the fees) and gives the token back to the licensor, or (*iii*) during the evaluation period a violation to Article 3 is identified and the authority pre-empts the license by calling either the function `compensateLicensor` or `compensateLicensee`. Notice that it is important that the code guarantees that in all possible cases the assets, both the token and the wallet, are not indefinitely locked in the contract.

3.3 Bike Rental contract: access to a good without transfer of ownership

We now consider a realistic contract for a city bike rental service[3], which exemplifies a general rental contract (this is taken from [18]). It involves two parties, the lender and the borrower, which initially agree about what good is rented, what use should be made of it, the time limit (or in which case it must be returned), the estimated of value and any defects in the good. Upon agreement, the payment triggers the legal bond, that is the borrower has the permission to use the bike and the lender has the prohibition of preventing him from doing so. Note that there is no transfer of ownership, but only the right to use the good. The contract terminates either when the borrower returns the bike, or when the time limit is reached. Litigations could arise when the borrower violates the obligations of diligent storage and care, the obligations of using the good only as intended, and not granting the use to a third party without the lender's consent. In these cases the lender may demand a compensation for the damage. On the other hand, the borrower is entitled to compensation if the good has defects that were known to the lender but that he did not initially disclose.

This example puts forward the fact that, when a legal contract refers to a *physical* good, the digital contract needs a digital handle (an avatar) for that good. Moreover, the rent legal contract grants just the *usage* of a good without the transfer of ownership. Many technological solutions, such as smart locks of IoT devices, are actually available. In *Stipula* we abstract from the specific nature of such a digital handle, and we simply represent it as an asset, which intuitively corresponds to a non fungible token associated to the physical good. Moreover, while the communication of the token provides full control of the associated physical good, we assume an operation `uses(token)` (resp. `use_once(token)` or `uses(token,A)`) that generates a usage-code, say a string, providing access to the object associated to the token (resp. a usage-code only valid (once) for the party `A`). Therefore, a physical object can be handled as a digital one using the same pattern used in the digital license contract above.

Figure 2 uses connected boxes to highlight the correspondence between the normative elements of a standard bike rental contract and the corresponding editing in *Stipula*. The parties agrees on the time limit for the rental and the cost of the service, which corresponds to the double of the fee in order to

[3]For instance see the contract in `http://www.thebicyclecellar.com/wp-content/uploads/2013/10/Bike-Rental-Contract-BW.pdf`

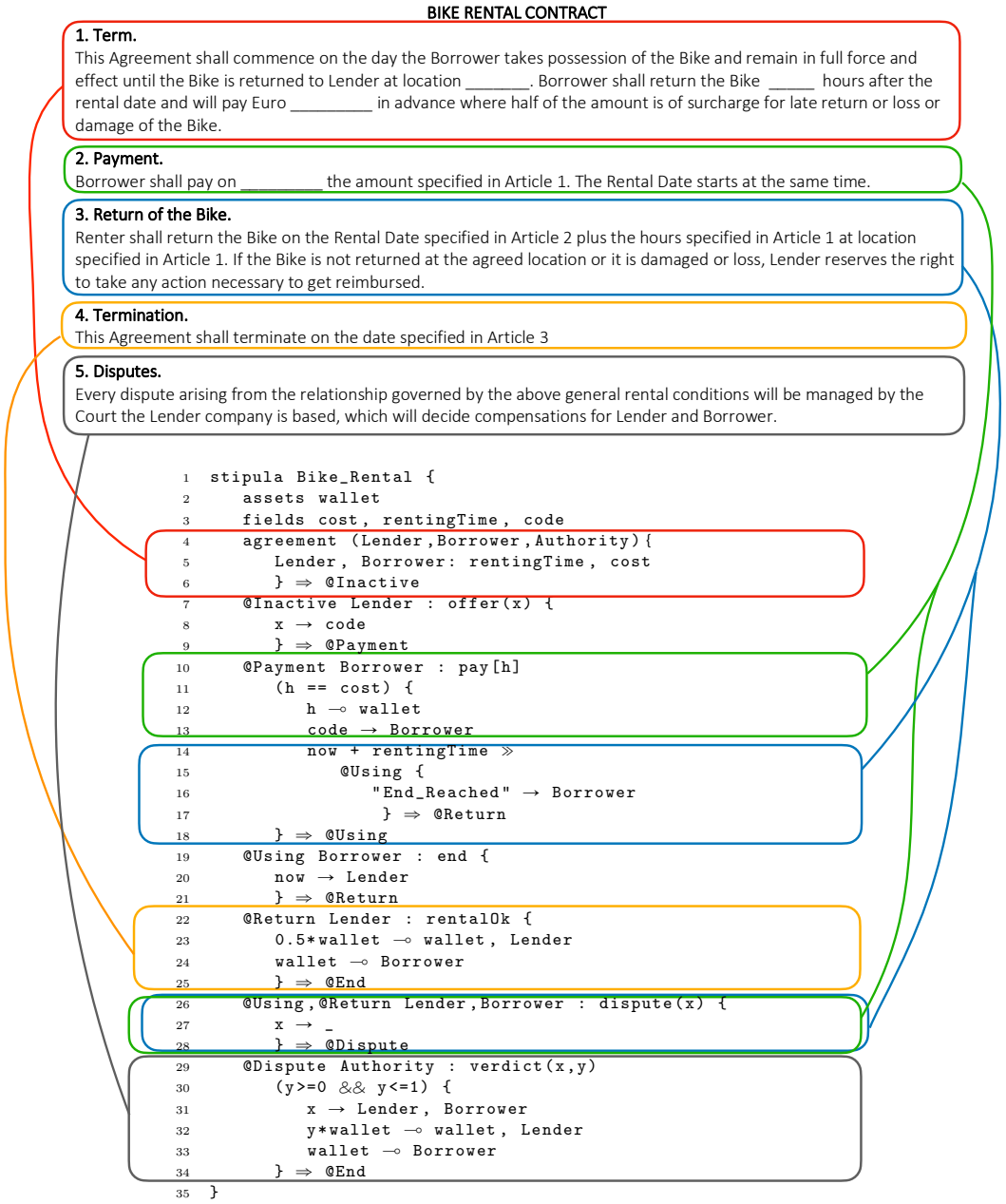


Figure 2: A standard Bike Rental contract and its modelling in *Stipula*

safeguard lender from damages, late returns or loss of the bike. For simplicity, in this code the `Lender` sends to the contract a simple usage code for the bike by calling the function `offer`. Then the `Borrower` pays the expected amount and receives the bike's usage code. Lines 14-18 issue an event corresponding to the obligation of returning the bike within the agreed time limit. Indeed, at time `now + rentingTime` the event is automatically triggered by the systems, and if the bike has not been already returned (*i.e.*, the state of the contract is still `@Using`), a message of returning the bike is sent to the borrower and the contract moves to the state `@Return`. The termination of the rental requires the `Borrower` to call the function `end`, after which the `Lender` has to confirm the absence of damages by invoking `rentalOK`. Only this sequence of actions allows the lender to be payed and the borrower to get back the money deposited as security. For the sake of simplicity this contract does not impose a penalty to the borrower for late return, but it is not difficult to modify the code with an additional state `@LateReturn` so to let the `Lender` keep the entire contract's wallet when `rentalOK` is called in the state `@LateReturn`.

The function `dispute` may be invoked either by the `Lender` or by the `Borrower`, either in state `@Using` or `@Return`, and carries the reasons for kicking the dispute off (`x` is intended to be a string). Once the reasons are communicated to every party (we use the abbreviation "`_`" instead of writing three times the sending operation) the contract transits into a state `@Dispute` where the `Authority` will analyze the issue and emit a verdict. This is performed by permitting in the state `@Dispute` only the invocation of the `verdict` function, that has two arguments: a string of motivations `x`, and a coefficient `y` that denotes the part of the wallet that will be delivered to `Lender` as reimbursement; the `Borrower` will get the remaining part. It is worth to spot this point: the statement `y*wallet ⊸ wallet, Lender` *takes* the `y` part of `wallet` (`y` is in [0..1]) and sends it to `Lender`; *at the same time* the `wallet` is reduced correspondingly. The remaining part is sent to `Borrower` with the statement `wallet ⊸ Borrower` (which is actually a shortening for `1*wallet ⊸ wallet, Borrower`) and the `wallet` is emptied.

3.4 Bet contract: dependency on external data

The bet contract is a simple example of a legal contract that contains an element of randomness (*alea*), *i.e.* where the existence of the performances or their extent depends on an event which is entirely independent of the will of the parties. The main element of the contract is a future, aleatory event, such as the winner of a football match, the delay of a flight, the future value of a company's stock.

A digital encoding of a bet contract requires that the parties explicitly agree on the source of data, usually an accredited web page or a specific online service – stored in the field `data_source` – that will publish the final value of the aleatory event. This value will be communicated by the party that assumes the role of `DataProvider`, taking the legal responsibility of supplying the correct data from the agreed source. In particular, it is not necessary that the actual data is directly provided by a trusted institution or an accredited online service, such as an Oracle service, who could hardly take an active legal responsibility in a bet contract. But two betters, say Alice and Bob, can agree to rely on a third party Carl for supplying data, or they can simply agree on the fact that Alice takes both the role of `Better1` and `DataProvider`.

It is also important that the digital contract provides precise time limits for accepting payments and for providing the actual value of the aleatory event. Indeed there can be a number of issues: the legal bond must be established before the occurrence of the aleatory event, the aleatory event might not happen, *e.g.* the football match is cancelled, or the data provider might fail to provide the required value, *e.g.* the online service is down.

The *Stipula* code in Listing 3 corresponds to the case where `Better1` and `Better2` place in `val1` and `val2` their bets, while the agreed amount of currency is stored in the contract's assets `wallet1` and

```
stipula Bet {
  assets wallet1, wallet2
  fields alea_fact, val1, val2, data_source, fee, amount, t_before, t_after

  agreement(Better1,Better2,DataProvider){
    DataProvider, Better1, Better2 : fee, data_source, t_after, alea_fact
    Better1, Better2 : amount, t_before
   } ⇒ @Init

  @Init Better1 : place_bet(x)[h]
    (h == amount){
        h ⊸ wallet1
        x → val1
        t_before ≫ @First { wallet1 ⊸ Better1 } ⇒ @Fail
    } ⇒ @First

  @First Better2: place_bet(x)[h]
    (h == amount){
        h ⊸ wallet2
        x → val2
        t_after ≫ @Run {
            wallet1 ⊸ Better1
            wallet2 ⊸ Better2 } ⇒ @Fail
  } ⇒ @Run

  @Run DataProvider : data(x,y,z)[]
    (x == data_source && y==alea_fact){
        if (z==val1 && z != val2){           // The winner is Better1
            fee ⊸ wallet2,DataProvider
            wallet2 ⊸ Better1
            wallet1 ⊸ Better1
        }
        else if (z==val2 && z != val1){      // The winner is Better2
            fee ⊸ wallet1,DataProvider
            wallet1 ⊸ Better2
            wallet2 ⊸ Better2
        }
        else {                                    //No winner
            fee*0.5 ⊸ wallet1,DataProvider
            fee*0.5 ⊸ wallet2,DataProvider
            wallet2 ⊸ Better1
            wallet1 ⊸ Better1

        }
   } ⇒ @End
}
```

Listing 3: The contract for a bet

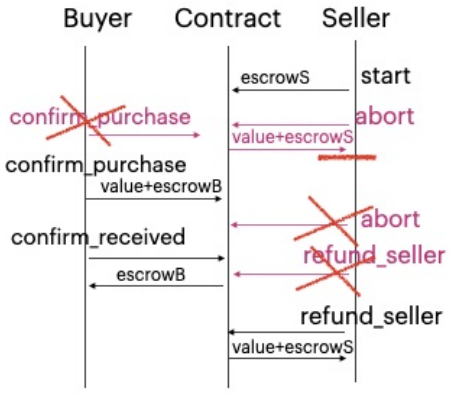


Figure 3: Safe Remote Purchase

`wallet2`[4]. Observe that both bets must be placed within an (agreed) time limit `t_before` (line 14), to ensure that the legal bond is established before the occurrence of the aleatory event. The second timeout, scheduled in line 21, is used to ensure the contract termination even if the `DataProvider` fails to provide the expected data, through the call of the function `data`.

Compared to the Authority pattern in the Digital Licence and Bike Rental examples, the role of the `DataProvider` here is less pivotal than that of the `Authority`. While it is expected that `Authority` will play its part, `DataProvider` is much less than a peer of the contract, that is entitled (and legally bound) to call the contract's function to supply the expected external data. The crucial point of trust here is the `data_source`, not the `DataProvider`. As usual, any dispute that might render the contract voidable or invalid, *e.g.*, one better knew the result of the match in advance, or the DataProvider supplied an incorrect value, can be handled by including an `Authority` party, according to the pattern illustrated above.

3.5 Safe Remote Purchase contract: a distributed interaction protocol

In a remote purchase[5], the buyer would like to receive an item from the seller and the seller would like to get money (or an equivalent) in return. The problematic part is the shipment: there is no way to determine for sure that the item arrived at the buyer. The typical solution is to define the interaction protocol so that both parties have *an incentive to resolve the situation* or otherwise their money is locked forever.

The idea is that both parties have to put an amount into the contract as escrow. As soon as this happened, the money will stay locked inside the contract until the buyer confirms that he received the item. The intended protocol is the following sequence of actions (depicted in black in Figure 3): (1) the seller starts the transaction sending its escrow to the contract, (2) the buyer confirms the purchase by sending to the contract the money corresponding to the price of the good plus the escrow, (3) upon reception of the good, the buyer has to confirm the reception to the contract in order to get back the escrow, (4) finally the seller can receive from the contract the price of the good and the money he deposited in escrow.

Besides the intended sequence of actions, many situations can happen in a remote purchase:

[4]For simplicity, this code requires `Better1` to place its bet before `Better2`, however it is easy to add similar function to let the two bets be placed in any order.

[5]This example is taken from `https://docs.soliditylang.org/en/develop/solidity-by-example.html#safe-remote-purchase`, but most of e-commerce platforms has similar use cases.

- if the seller starts but the buyer does not confirm the purchase, seller can take back its escrow with a call to `abort`,
- if the seller does not start, the buyer does not send the escrow, so no money is locked,
- if the buyer confirms the purchase, the seller cannot take back its escrow (and the payment) until he sends the good (and it is received),
- if the buyer has confirmed the purchase but he does not confirm the reception, either because the good is not arrived or because the Buyer is cheating, nobody can take back escrow. Therefore we add to the contract a time limit, after which it is up to an Authority party to decide off-line who is to blame and then implement the decision by calling `refund`. In other terms, the mutual escrow is used as an incentive for the parties to collaborate, but *progress is not ensured* thus the contract requires *timeouts*.

```
stipula Purchase {
  asset wallet
  field value, escrow

  agreement(Buyer,Seller,Authority){
     Seller, Buyer, Authority, : value, escrow, time_limit
  } => @Init

   @Init Seller : start [h] (h == escrow) { h —∘ wallet } =>@Created

   @Created  Seller: abort { wallet —∘ Seller } =>@Inactive

   @Created Buyer : confirmPurchase [h] (h == value + escrow) {
       h —∘ wallet
       now + time_limit >> @Locked {
                           "nothing received (maybe!)" → Buyer
                           "nothing received (maybe!)" → Seller
                         }=> @Dispute
   } =>@Locked

   @Locked Buyer : confirmReceived { escrow —∘ wallet, Buyer } =>@Release

   @Release Seller : refundSeller  {
      wallet —∘ Seller // equal to (value+escrow) —∘ wallet, Seller
   } =>@Inactive

   @Dispute Authority : refund(x,y) (wallet==x+y) {
       x —∘ wallet, Buyer
       y —∘ wallet, Seller
   } =>@Inactive
}
```

Listing 4: The safe remote purchase contract

We remark that the contract in Listing 4 does not solve all legal issues. For instance in a purchase the consumer has the power to withdraw from an online sale, and there are usually warranties if the good was damaged or different from the sellers' description. To deal with all these situations the contract can be enriched with a more complex Authority pattern as in the previous examples.

3.6 Mutual Dissent and contract modification

There is a last distinctive element in legal contracts that deserves a comment: the management of exceptional behaviours, that is all those behaviours that cannot be anticipated due to the occurrence of unforeseeable and extraordinary events. For instance, legal contracts can always be dissolved if the parties agree.

We can model the *mutual dissent* by including a specific function in the contract, which can be activated with the agreement of both parties, that causes the contract to go into a stand-by state, which blocks the execution of all functions not yet performed. This prevents the contract from continuing when both parties no longer want it to. More precisely, the following code shows the Mutual Dissent pattern for a generic contract `C` where parties `P1` and `P2` may express mutual dissent:

```
stipula Rescindable_C {
  assets a1, ..., an
  fields ...
  agreement(P1,P2,....,Authority) ....

  // add a copy of this function for any state X of the contract C
  @X P1: dissent { now + 1 day >> @OneDissented { }=>@X } =>@OneDissented
  @OneDissented P2: dissent {} => @Rescinded

  @X P2: dissent { now + 1 day >> @TwoDissented { }=>@X } =>@TwoDissented
  @TwoDissented P1: dissent{} => @Rescinded

  @Rescinded  Authority : terminate {
    a1  --o Authority
    ...
    an  --o Authority
  } =>@End
```

Listing 5: The mutual dissent pattern

To prevent assets being locked indefinitely in the contract, the function `terminate` sends all the assets to the authority. More complex assets reallocation to the parties can also be implemented, provided that they mutually agree on the reallocation.

Finally, parties have the power of dynamically change the terms of the contract if they agree to it. Contract modification can be modelled by the termination of the running contract `C` (with the mutual dissent pattern), and the activation of a new contract `C'`, to which the assets remaining in `C` are transferred. The basic *Stipula* language does not allow to pass contracts' names as arguments, nor allows to invoke external contracts' activations or inter-contracts functions invocations (differently from, *e.g.* Solidity smart contracts). Therefore, the bridge between the termination of `C` and the activation of `C'` with remaining assets must be performed off-line by the Authority.

4 Legal contracts and the power of formal methods

As already discussed, the advantage of using a Legal Calculus to draft legal contracts is that a concise and well-defined language reduces the ambiguities (and therefore the grey areas) characteristics of traditional legal drafting. In particular, we remark that there are three levels of formalisation, corresponding to three different aspects of a language: the syntax, the semantics, and the analysis and verification tools.

Almost all projects for Code-Driven Law put forward a legal language based on a well-defined syntax. This is indeed the base to mechanise the writing of dedicated software that encodes legal content –not just legal contracts but any kind of legal data. These projects often come with templates for standard legal documents, that can be customised by setting template's parameters with appropriate values. There are legal language definitions based on context free grammars (as Lexon [30]), or domain specific markup languages and ontologies to wrap logic and other contextual informations around traditional legal prose (as OpenLaw [37], Accord [17], SLCML [24]), or legal specification languages based on visual programming interfaces (as in [32, 36]).

More complex is instead the formalisation of the semantics of a programming language, which is however essential to have the full understanding of the software, and the certainty of the dynamic contract's behaviour. Legal calculi, such as Catala [31], Orlando [11] and Stipula [19], have the suitable size to fully handle their formal semantics. We discuss below the case of *Stipula*, which acknowledges the concurrent nature of legal contracts as interaction protocols, and resorts to concurrency theory to define the semantics of contracts and to precisely control complex aspects like nondeterminism.

Finally, the most powerful benefit of formal methods is the deployment of automated tools to (statically) analyse the legal software in order to check safety properties, verify the absence of specific errors and possibly the reachability of convenient states. This level of formalisation is still at an initial stage in the literature, since it requires a robust definition (and implementation) of the language semantics, and because the identification of the desirable properties of legal software is still an open question.

4.1 Defining a formal semantics

The full definition of *Stipula*'s operational semantics (currently submitted to publication, but a preliminary version is available in [19]) is given in terms of a labelled transition system $\mathbb{C},\mathbb{t} \xrightarrow{\mu} \mathbb{C}',\mathbb{t}'$ that highlights the open nature of the contracts' behaviour, whose execution requires the interaction with the external context. The *runtime configuration* $\mathbb{C},\mathbb{t}$ is a pair where $\mathbb{C}$ is the runtime status of the running contract (storing its current state and the pending events), and $\mathbb{t}$ is the time value of the system's global clock. The actions that can be performed by a contract time $\mathbb{t}$ are the following

$$\mu \quad ::= \quad \tau \quad | \quad (\overline{A}, \overline{A_i} : \overline{v_i}^{i\in 1,..,n}) \quad | \quad A : \mathtt{f}(\overline{u})[\overline{v}] \quad | \quad v \to A \quad | \quad a \multimap A \,.$$

where the label $(\overline{A}, \overline{A_i} : \overline{v_i}^{i\in 1,..,n})$ observes the agreement that the parties are going to sign, that is who is taking the legal responsibility for which contract's role, and what are the terms of the contract, *i.e.*, the agreed initial values of the contract's fields. The label $A : \mathtt{f}(\overline{u})[\overline{v}]$ observes the possibility (at time $\mathbb{t}$) for the party A to call the function f. The labels $v \to A$ and $a \multimap A$ observe that (at time $\mathbb{t}$) the party A can receive a value and an asset, respectively. Contract's field updates, internal asset moves and event scheduling, as well as time progress, are not observed (label τ).
The behaviour of a *Stipula* legal contract can be described as the following procedure:

1. the first action is always an agreement, which moves the contract to an idle state;
2. in an idle state, if there is a ready event with a matching state, then its handler is completely executed, moving again to a (possibly different) idle state;
3. in an idle state, if there is no event to be triggered, either advance the system's clock or call any permitted function (*i.e.* with matching state and preconditions). A function invocation amounts to execute its body until the end, which is again an idle state.

Therefore, the semantics has three sources of nondeterminism: (*i*) the order of the execution of ready events' handlers, (*ii*) the order of the calls of permitted functions, and (*iii*) the delay of permitted function calls to a later time (thus, possibly, after other event handlers). We remark that a nondeterministic behaviour is not necessarily an error: even the execution of legal contracts written in natural language might lead to nondeterministic executions, in particular when the contract leaves room for a participant not to timely perform an action that was expected to do. Depending on how the contract is written, this may be admissible or may cause a legally uncertain situation that can only be solved by a court. Therefore, the precise formalisation of a contract's behaviour in terms of an operational semantics has the advantage of explicitly knowing what are the sources of nondeterminism, and allows to precisely control it.

4.2 Observing legal contracts through Normative Equivalence

One of the difficulties of writing contracts in natural language is the fact that the same legal bindings can be expressed with many similar texts. Then it is often difficult to properly check when two contracts that are syntactically different are instead legally equivalent, meaning that the parties using them cannot distinguish one from the other. By relying on the operational semantics, that formally defines the observable actions of a contract behaviour, we can define a *bisimulation-based observation equivalence*, where two contracts are deemed to be legally equivalent if they involve the same parties observing the same interactions during the contracts' lifetime.

More precisely, the so-called *Normative Equivalence* (see [19]) equates two contracts if

- they provide the same agreement, that is the same parties take the same legal responsibility and agree on the same terms of the contract (expressed by the action $(\overline{A}, \overline{A_i} : \overline{v_i}^{i\in 1..n})$);
- every party is subject to the same dynamic set of permissions, prohibitions and obligations;
- every party receives from the contract the same assets and values (actions $v \to A$ and $a \multimap A$);
- the bisimulation game abstracts away the ordering of the observations within the same time clock, and enforces a transfer property that shifts the time of observation to the next time unit.

To observe the permissions and prohibitions at time $\mathbb{t}$, we observe whether any party can invoke, resp. cannot invoke, any function (expressed by the action $A : \mathtt{f}(\overline{u})[\overline{v}]$). Obligations are captured implicitly by shifting the observation at a specific point in time, and observing –in the future– the effects of executing the event that encodes the legal commitment. In particular, the system's clock needs not to be directly observed: by checking the set of permissions and prohibitions at any time units, and since only a contract's state change can modify the set of valid permissions and prohibitions, it is sufficient to observe whether a function can be executed before of after another function or an event, disregarding its precise execution time unit.

As a consequence, the Normative Equivalence safely abstracts away the ordering of the observations within the same time unit: if a party receives two messages in different order it might be due to delays of communications, rather to sensible differences in the contracts. Nevertheless, the equivalence does not overlook essential precedence constraints, which are important in legal contracts, as the requirement that a function delivering a service can only be invoked after another specific function, say a payment. Additionally, the Normative Equivalence abstracts away from the names of the contract's assets, fields and internal states, and it is also independent from future clock values, allowing to garbage-collect events that cannot be triggered anymore because the time for their scheduling is already elapsed.

4.3 Verification of contracts' properties

By looking at legal contracts as interaction protocols and by relying on a well defined operational semantics, the rich theory of formal methods for concurrent systems can be a great source of inspiration to develop analysis and verification tools. However, first of all it is essential to conduct an interdisciplinary investigation to properly identify what are the errors and the properties that should be targeted by the techniques providing safety and liveness guarantees.

An important class of errors are those related to unsafe usage of assets, which must obey to a linear semantics (no forging, no duplication, no loss) and whose content must be meaningful. For instance, in *Stipula* the assets corresponding to currency, as the asset `wallet` in the examples above, must always contain a non negative amount of money. Accordingly, an asset transfer what would leave a contract's asset with a negative (unsafe) asset, *e.g.*, `100` $\multimap$ `wallet,A` when `wallet` holds less than 100 coins, is not executed and results in a stuck configuration. Similarly, if the contract's asset `token` already contains a non fungible token providing access to a good, say a digital service, then the operation `t` $\multimap$ `token` that would accumulate or overwrite the `token` with the asset `t` must not be executed. Moreover, assets must not be indefinitely locked into contracts: at any time it should be possible, at least for some party, to redeem the assets stored into the contract; this is often called *liquidity* property ([10]). These issues are at the core of the research about *resource-aware languages* as Obsidian [4, 15] Nomos [23, 14], Flint [33] and Move [13]; and even the questions "What is the type `Money` in a programming language? What are its suitable abstractions?" and "What is the difference between the more general type `Asset` and the type `Money`?" are still open issues.

Other kinds or errors are those related to non collaborative parties, that might prevent the progress of the contract or might move it to a problematic state. We have described *Stipula* design patterns, as the authority pattern or the mutual dissent pattern, that can be inserted in the drafting of the digital contract as a sort of escape hatch; however, a static analysis of the runtime behaviour of the contract would be very useful.

5 Conclusions

In this article we discussed the role of Legal Calculi in the process of digitisation of legal contracts. We illustrated the design choices of *Stipula*, whose primitives naturally support the encoding of contracts' normative elements (permissions, prohibitions, obligations, asset transfer, judicial enforcement and openness to the external context). We also remarked that legal contracts can be interpreted as interaction protocols between concurrent parties, leading to a fruitful connection with the rich toolset of formal methods available for concurrent systems.

Studying the theory of domain-specific legal calculi is a useful research line, that supplement the development of the Code-Driven Law trend. On the other hand, it is important to keep a lively connection between these calculi and other two fundamental abstraction levels: the effective implementation and the interdisciplinary assessment. The actual implementation of legal calculi brings in specific challenges, such as the *legally robust* management of the identities of the parties and their valid agreement to the legal bonds. Moreover, the implementation of obligations by scheduling an event that issues a corresponding penalty if the obligation has not been met, may not be always feasible, and asks for an accurate management of time, which is a well-known challenge in distributed platforms.

Finally, the dialogue with legal researchers and professionals provides valuable insights, non just on the usability of legal programming languages, but mainly on the actual meaning (in the epistemic sense) of their abstractions. This is important to unveil when partial or erroneous interpretations of the law

has been embedded in the technical artefacts, and to understand the actual extent of the legal protection provided by the software normativity. A main lesson that we learned is the intrinsic open nature of legal contracts, that is incompatible with the automatic execution of software-based rules claimed by the Code-Driven Law. Indeed, a contract produces the intended effects, declared by the parties, only if it is legally valid: the law may deny validity to certain clauses, as an excessive interests rate. The intervention of the law is particularly significant when the contractor (usually the weaker party, such as the worker in an employment contract or the consumer in an online purchase) agrees without having awareness of all clauses in the contract, nor having the ability to negotiate them, due to the existing unbalance of power ([19]). Therefore, any technical solution based on a legal programming language must provide an escape mechanism (as the authority pattern in *Stipula*) that allows a flexible, and legally valid, link between what is true on-line and off-line.

References

[1] *Solidity Documentation: State Machine Common Pattern*. `https://docs.soliditylang.org/en/v0.8.0/common-patterns.html#state-machine`.

[2] (2018): *Akoma Ntoso XML for parliamentary, legislative and judiciary documents*. At `http://www.akomantoso.org/`.

[3] (2018): *Malta MDIA Act*. At `https://mdia.gov.mt/wp-content/uploads/2018/10/MDIA.pdf`.

[4] (2018): *Obsidian: A safer blockchain programming language*. Language Site at `http://obsidian-lang.com/`.

[5] (2019): *The CoHuBiCoL research project*. At `https://www.cohubicol.com/about`.

[6] (2019): *Smart contract legislation and enforceability in Italy*. Gazzetta Ufficiale, Law of 11 febbraio 2019, n. 12, Art. 8 ter, at `https://www.gazzettaufficiale.it/eli/id/2019/02/12/19G00017/sg`.

[7] (2020): *Cracking the Code: Rulemaking for humans and machines*. At `https://oecd-opsi.org/publications/cracking-the-code/`.

[8] (2021): *Wyoming Regulation Act*. At `https://www.wyoleg.gov/Legislation/2021/SF0038`.

[9] (2022): *Catala in action*. Language site at `https://catala-lang.org/`.

[10] Massimo Bartoletti & Roberto Zunino (2019): *Verifying Liquidity of Bitcoin Contracts*. In Flemming Nielson & David Sands, editors: *Principles of Security and Trust*, Springer International Publishing, pp. 222–247, doi:10.1007/978-3-030-17138-4_10.

[11] Shrutarshi Basu, Nate Foster & James Grimmelmann (2019): *Property Conveyances as a Programming Language*. In: *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2019, Association for Computing Machinery, New York, NY, USA, p. 128–142, doi:10.1145/3359591.3359734.

[12] Shrutarshi Basu, Anshuman Mohan, James Grimmelmann & Nate Foster (2022): *Legal Calculi*. Technical Report, ProLaLa 2022 ProLaLa Programming Languages and the Law. At `https://popl22.sigplan.org/details/prolala-2022-papers/6/Legal-Calculi`.

[13] Sam Blackshear & et al. (2021): *Move: A Language With Programmable Resources*. `https://developers.diem.com/papers/diem-move-a-language-with-programmable-resources/2020-04-09.pdf`.

[14] Sam Blackshear, David L. Dill, Shaz Qadeer, Clark W. Barrett, John C. Mitchell, Oded Padon & Yoni Zohar (2020): *Resources: A Safe Language Abstraction for Money*. *CoRR*. arXiv:2004.05106.

[15] Michael J. Coblenz, Jonathan Aldrich, Brad A. Myers & Joshua Sunshine (2020): *Can advanced type systems be usable? An empirical study of ownership, assets, and typestate in Obsidian*. *Proc. ACM Program. Lang.* 4(OOPSLA), pp. 132:1–132:28, doi:10.1145/3428200.

[16] Michael J. Coblenz, Reed Oei, Tyler Etzel, Paulette Koronkevich, Miles Baker, Yannick Bloem, Brad A. Myers, Joshua Sunshine & Jonathan Aldrich (2020): *Obsidian: Typestate and Assets for Safer Blockchain Programming*. *ACM Trans. Program. Lang. Syst.* 42(3), pp. 14:1–14:82, doi:10.1145/3417516.

[17] Open Source Contributors (2018): *The Accord Project*. `https://accordproject.org`.

[18] Silvia Crafa & Cosimo Laneve (2022): *Programming legal contracts - a beginner guide*. In: *The Logic of Software. A Tasting Menu of Formal Methods. Essays Dedicated to Reiner Hähnle on the Occasion of His 60th Birthday*, *Lecture Notes in Computer Science* 13360, Springer, doi:10.1007/978-3-031-08166-8.

[19] Silvia Crafa, Cosimo Laneve & Giovanni Sartor (2021): *Pacta sunt servanda: legal contracts in Stipula*. *CoRR*. arXiv:2110.11069.

[20] Silvia Crafa, Cosimo Laneve & Giovanni Sartor (2022): *Le forme del falso negli smart contract*. In: *Le forme del falso*, Bologna University Press, pp. 85–98, doi:10.30682/9791254770146.

[21] Silvia Crafa, Cosimo Laneve & Giovanni Sartor (2022): *Stipula: a domain specific language for legal contracts*. Technical Report, ProLaLa 2022 ProLaLa Programming Languages and the Law. At `https://popl22.sigplan.org/details/prolala-2022-papers/6/Legal-Calculi`.

[22] Silvia Crafa, Cosimo Laneve & Adele Veschetti (2022): *Stipula Prototype*. Available on github: `https://github.com/stipula-language`.

[23] A. Das, S. Balzer, J. Hoffmann, F. Pfenning & I. Santurkar (2021): *Resource-Aware Session Types for Digital Contracts*. In: *2021 2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 111–126, doi:10.1109/CSF51468.2021.00004.

[24] Vimal Dwivedi, Alex Norta, Alexander Wulf, Benjamin Leiding, Sandeep Saxena & Chibuzor Udokwu (2021): *A Formal Specification Smart-Contract Language for Legally Binding Decentralized Autonomous Organizations*. *IEEE Access* 9, pp. 76069–76082, doi:10.1109/ACCESS.2021.3081926.

[25] Vimal Dwivedi, Vishwajeet Pattanaik, Vipin Deval, Abhishek Dixit, Alex Norta & Dirk Draheim (2021): *Legally Enforceable Smart-Contract Languages: A Systematic Literature Review*. *ACM Comput. Surv.* 54(5), doi:10.1145/3453475.

[26] Primavera De Filippi & Samer Hassan (2016): *Blockchain technology as a regulatory technology: From code is law to law is code*. *First Monday* 21(12), doi:10.5210/fm.v21i12.7113.

[27] Guido Governatori, Florian Idelberger, Zoran Milosevic, Regis Riveret, Giovanni Sartor & Xiwei Xu (2018): *On legal contracts, imperative and declarative smart contracts, and blockchain systems*. *Artificial Intelligence and Law* 26, pp. 377–409, doi:10.1007/s10506-018-9223-3.

[28] Xiao He, Bohan Qin, Yan Zhu, Xing Chen & Yi Liu (2018): *SPESC: A Specification Language for Smart Contracts*. In: *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, 01, pp. 132–137, doi:10.1109/COMPSAC.2018.00025.

[29] Lawrence Lessig (1999): *Code and Other Laws of Cyberspace*. Basic Books, Inc., USA.

[30] Lexon Foundation (2019): *Lexon Home Page*. `http://www.lexon.tech`.

[31] Denis Merigoux, Nicolas Chataing & Jonathan Protzenko (2021): *Catala: A Programming Language for the Law*. *Proc. ACM Program. Lang.* 5(ICFP), doi:10.1145/3473582.

[32] Christian Reitwiebner (2018): *Babbage—A Mechanical Smart Contract Language*. At `https://medium.com/@chriseth/babbage-a-mechanical-smart-contract-language-5c8329ec5a0e`.

[33] Franklin Schrans, Susan Eisenbach & Sophia Drossopoulou (2018): *Writing Safe Smart Contracts in Flint*. In: *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*, Programming'18 Companion, ACM, New York, NY, USA, p. 218–219, doi:10.1145/3191697.3213790.

[34] David Siegel (2016): *Understanding the dao attack*. Available at `https://top-forex-brokers.com/2021/10/07/understanding-the-dao-attack/`.

[35] Study Group on a European Civil Code & Research Group on EC Private Law (Acquis Group) (2009): *Principles, Definitions and Model Rules of European Private Law: Draft Common Frame of Reference (DCFR),*

Outline Edition. Sellier. Available at https://www.ccbe.eu/fileadmin/speciality_distribution/public/documents/EUROPEAN_PRIVATE_LAW/EN_EPL_20100107_Principles__definitions_and_model_rules_of_European_private_law_-_Draft_Common_Frame_of_Reference__DCFR_.pdf.

[36] Tim Weingaertner, Rahul Rao, Jasmin Ettlin, Patrick Suter & Philipp Dublanc (2018): *Smart Contracts Using Blockly: Representing a Purchase Agreement Using a Graphical Programming Language*. In: *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pp. 55–64, doi:10.1109/CVCBT.2018.00012.

[37] Aaron Wright, David Roon & ConsenSys AG (2019): *OpenLaw Web Site*. https://www.openlaw.io.

A Generic Type System for Higher-Order Ψ-calculi

Alex Rønning Bendixen
Department of Computer Science
Aalborg University, Denmark

Bjarke Bredow Bojesen
Department of Computer Science
Aalborg University, Denmark

Hans Hüttel
Department of Computer Science
University of Copenhagen, Denmark
Department of Computer Science
Aalborg University, Denmark
hans.huttel@di.ku.dk

Stian Lybech*
Department of Computer Science
Reykjavík University, Iceland
stian21@ru.is

The Higher-Order Ψ-calculus framework (HOΨ) is a generalisation of many first- and higher-order extensions of the π-calculus. It was proposed by Parrow et al. who showed that higher-order calculi such as HOπ and CHOCS can be expressed as HOΨ-calculi. In this paper we present a generic type system for HOΨ-calculi which extends previous work by Hüttel on a generic type system for first-order Ψ-calculi. Our generic type system satisfies the usual property of subject reduction and can be instantiated to yield type systems for variants of HOπ, including the type system for termination due to Demangeon et al.. Moreover, we derive a type system for the ρ-calculus, a reflective higher-order calculus proposed by Meredith and Radestock. This establishes that our generic type system is richer than its predecessor, as the ρ-calculus cannot be encoded in the π-calculus in a way that satisfies standard criteria of encodability.

1 Introduction

Process calculi are formalisms for modelling and reasoning about concurrent and distributed computations; a prominent example being the π-calculus of Milner et al. [19, 25], which models computation as communication between processes, by passing messages on named channels. Since its inception, a multitude of variants of the π-calculus have appeared; e.g. Dπ [11], the calculus of explicit fusions [9], the spi-calculus with correspondence assertions [1] and the $^{e}\pi$-calculus [6]. These calculi are all *first-order*, in the sense that only atomic channel names can be passed around, not processes themselves. Bengtson et al. [3, 4] created Ψ-*calculi* as a generalisation of these first-order variants and extensions, allowing a range of calculi, including all of the aforementioned, to be expressed as *instances* of the Ψ-calculus framework through appropriate settings of a small number of parameters. However, there also exist *higher-order* variants of the π-calculus, such as the Higher-Order π-calculus, HOπ, [24, 23], that also allow *processes* to be sent across channels. Parrow et al. [21] have extended the Ψ-calculus framework with a construct for higher-order communication, creating the *Higher-Order* Ψ-calculus, HOΨ. Calculi such as HOπ and CHOCS [26] can now be represented as HOΨ-instances, as well as every calculus that the 'first-order' Ψ-calculus framework can represent.

One of the techniques for reasoning about processes is that of *type systems*. The first type system for a process calculus is due to Milner [19] and deals with the notion of correct usage of channels in the π-calculus: In a well-typed process only names of the correct type can be communicated. Pierce and Sangiorgi [22] later described a type system that uses subtyping and capability tags to control the use

*The work by this author was partly supported by the Icelandic Research Fund Grant No. 218202-05(1-3).

V. Castiglioni and C. A. Mezzina (Eds): Combined Workshop on
Expressiveness in Concurrency and Structural Operational Semantics 2022
(EXPRESS/SOS 2022).
EPTCS 368, 2022, pp. 43–59, doi:10.4204/EPTCS.368.3

of names as input or output channels; and also many of the aforementioned first-order extensions of the π-calculus have been given type systems to capture such properties as secrecy, authenticity and access control.

In [12], Hüttel noted that these type systems, despite arising in different settings, share certain characteristics: The type judgments for processes P are all of the form $\Gamma \vdash P$ where Γ is a type environment recording the types of the free names in P, so processes are only classified as being either well-typed or not. On the other hand, terms M are given a type T, so type judgements for terms are of the form $\Gamma \vdash M : T$. Based on these shared characteristics, Hüttel then created a generic type system for the first-order Ψ-calculus framework, that generalises several of the type systems for the π-calculus and its variants. This generic type system can similarly be instantiated through parameter settings to yield both well-known and new type systems for the calculi that are representable as first-order Ψ-calculi. An important advantage of this approach is that a general result of type system soundness can be formulated, which is then inherited by all instances of the type system.

There has been some other works on generic type systems, notably those of König [16], Caires [5] and Igarashi and Kobayashi [15]. However, these are formulated for variants of the first-order π-calculus, which thus limits their applicability to languages that can be represented in the first-order paradigm. Stated otherwise, they exclude languages such as the aforementioned ${}^{e}\pi$-calculus, which cannot be encoded into the first-order π-calculus, as shown in [6], but which nevertheless can be given a type system using the generic approach of Hüttel, indicating that the latter is a more general framework for first-order calculi.

However, the generic type system of Hüttel can only type *first-order* calculi; it cannot be instantiated to yield type system for *higher-order* calculi, such as HOπ or CHOCS. Both of these higher-order calculi can be encoded into the first-order π-calculus, as shown by Sangiorgi in [24], and may therefore also be represented in just the first-order Ψ-calculus. Not surprisingly, there is therefore little work on type systems for higher-order calculi, since these encodings allow us to disregard the higher-order behaviour and instead just type the first-order translations. One exception is the type system for termination in variants of HOπ, due to Demangeon et al. [7]. As these authors argue, it may not always be desirable (or even possible) to type a higher-order language through a first-order representation, if the language contains features that are difficult (or impossible) to encode. For example, higher-order behaviour may alternatively be viewed as a special case of *reflection*; i.e. the ability of a program to turn code into data, modify or compute with it, and reinstantiate it as running code; and process mobility here appears as a special case where data (code/processes) are transmitted without modification.

This reflective capability is inherent in the Reflective Higher-Order (RHO or ρ) calculus of Meredith and Radestock [18], and this calculus cannot be uniformly encoded in the π-calculus, as shown in [17]. This calculus therefore gives us an example of a language that cannot easily be represented in the first-order paradigm, thus making it difficult (or impossible) to adapt any of the existing first-order type systems to this language. Yet it *can* be instantiated as a HOΨ-calculus, as we shall show in the following.

The goal of the present paper is therefore to extend the aforementioned generic type system by Hüttel, to create a generic type system for the HOΨ-calculus framework that will allow us to capture typability in the higher-order paradigm. It allows us to identify what should be required of type systems for higher-order process calculi that are instances of the HOΨ-calculus, and these requirements here take the form of a number of assumptions that must hold for each instance. Like its predecessor, our generic type system also satisfies a general subject reduction property that is inherited by all instances. We use this to formulate simple type systems for HOπ, and we show that the type system for termination by Demangeon et al. also can be captured as an instance of our type system. Lastly, we show that our generic type system can be instantiated to yield a type system for the ρ-calculus, which establishes that our type system is richer than the first-order type systems. To our knowledge, no type system has hitherto been

published for this calculus, so we regard this instance as a further contribution of the present paper. A technical report with full proofs of most results is available in [2].

2 The higher-order Ψ-calculus

The Higher-Order Ψ-calculus extends the original Ψ-calculus [3] with primitives for higher-order communication, i.e. process mobility. In this section we first review the syntax of HOΨ as given in [21], and then proceed to give a reduction semantics for the calculus.

2.1 Syntax

The Higher-Order Ψ-calculus is a general framework, which is intended to allow many different calculi to be obtained as instances, by setting a small number of parameters which takes the form of definitions of three (not necessarily disjoint) sets of *terms, conditions* and *assertions*. To allow the framework to be as general and flexible as possible, the authors of [3, 21] identify only a few restrictions that must be imposed on these sets: they must be *nominal datatypes*. Informally, a *nominal set*, in the sense of Gabbay and Pitts [8], is a set whose members can be affected by names being bound or swapped. If a, b are names and X is an element of a nominal set, then the *transposition* of a and b on X, written $(a,b)\cdot X$, swaps all occurrences of a for b in X and vice versa. A function on a nominal set is *equivariant*, if it is unaffected by name swapping; and a *nominal datatype* is a nominal set together with a set of equivariant functions on it. This requirement is very mild and allows e.g. non-well-founded sets to be used in an instantiation. The utility of this shall become apparent later, when we create a HOΨ-calculus instance where the set of processes (which itself contains terms) is included in the set of terms.

Another important notion is that of support: if X is an element of a nominal set, the *support* of X, written $\mathrm{n}(X)$, is the set of names that occur in X. Conversely, a name a is *fresh for* X, written $a\#X$, if $a \notin \mathrm{n}(X)$; and we extend this to sets of names A such that $A\#X$ if it is the case that $\forall a \in A.a \notin \mathrm{n}(X)$. This is pointwise extended to lists of elements $X_1,\ldots,X_n$, so we write $A\#X_1,\ldots,X_n$ for $A\#X_1 \wedge \ldots \wedge A\#X_n$.

As mentioned above, any Ψ-calculus instance requires a specification of three nominal datatypes: the terms, conditions and assertions. The datatype of *terms*, ranged over by $M,N \in \mathbb{T}$, contains the terms that can be communicated and used as channels. These could be e.g. single names, as in the monadic π-calculus, vectors of names as in ${}^e\pi$ and the polyadic π-calculus; or elements of a composite datatype (e.g. the integers). The datatype of *conditions*, ranged over by $\varphi \in \mathbb{C}$ contains the conditions that can be used in conditional process expressions. Finally, and importantly, we have the nominal datatype of *assertions*, ranged over by $\Psi \in \mathbb{A}$. Each of the datatypes $\mathbb{T}$, $\mathbb{C}$ and $\mathbb{A}$ must include an equivariant *substitution function*, written $(\cdot)[\widetilde{a} := \widetilde{M}]$, substituting tuples of terms $\widetilde{M}$ for tuples of names $\widetilde{a}$ of equal arity. It must be defined such that it satisfies the following *substitution laws:*

1. If $\widetilde{a} \subseteq \mathrm{n}(X)$ and $b \in \mathrm{n}(\widetilde{Y})$ then $b \in \mathrm{n}\Big(X[\widetilde{a} := \widetilde{Y}]\Big)$
2. If $\widetilde{u}\#X, \widetilde{v}$ then $X[\widetilde{v} := \widetilde{Y}] = \big((\widetilde{u},\widetilde{v})\cdot X\big)[\widetilde{u} := \widetilde{Y}]$

The requirements are quite general and should be satisfied by any ordinary definition of substitution: The first law states that names cannot be lost in substitution, i.e. the names present in $\widetilde{Y}$ must also be present when the substitution has been performed; whilst the second law states that substitution cannot be affected by transposition.

Since the calculus allows arbitrary terms to be used as channels, any Ψ-calculus instance requires a definition of two equivariant operators, *channel equivalence* $\leftrightarrow$ and *assertion composition* $\otimes$, a *unit*

element 1 of assertions, and an *entailment relation* $\Vdash$, defined on the respective nominal datatypes and with the following signatures:

$$\begin{array}{ll} \leftrightarrows : \mathbb{T} \times \mathbb{T} \to \mathbb{C} \text{ channel equivalence} & 1 \in \mathbb{A} \text{ assertion unit} \\ \otimes : \mathbb{A} \times \mathbb{A} \to \mathbb{A} \text{ assertion composition} & \Vdash \subseteq \mathbb{A} \times \mathbb{C} \text{ entailment relation} \end{array}$$

We write the entailment relation as $\Psi \Vdash \varphi$ instead of $(\Psi, \varphi) \in \Vdash$ to denote that the condition φ holds, given the assertions Ψ. Note that comparison by channel equivalence $M_1 \leftrightarrows M_2$ is itself a condition, which may or may not be entailed by some assertions Ψ, according to the definition of the entailment relation.

The set of HOΨ-calculus *processes* $\mathscr{P}_\Psi$ is generated by the formation rules:

$$\begin{array}{rl} P \in \mathscr{P}_\Psi ::= & \mathbf{0} \mid P_1 \mid P_2 \mid \overline{M}N.P \mid \underline{M}(\lambda \widetilde{x} : \widetilde{T})N.P \\ & \mid \mathbf{run}\, M \mid \mathbf{case}\, \widetilde{\varphi} : \widetilde{P} \mid (\nu x : T)\, P \mid\ !P \mid (\!|\Psi|\!) \end{array}$$

where $\widetilde{\varphi} : \widetilde{P} \triangleq \varphi_1 : P_1 \ [] \ \ldots \ [] \ \varphi_n : P_n$.

Most of these constructs are similar to those of the π-calculus; the input and output prefixes generalise those of the π-calculus, since here both subject and object are *terms* rather than just names. Thus $\overline{M}N.P$ outputs the term N on M and continues as P, whilst $\underline{M}(\lambda \widetilde{x} : \widetilde{T})N.P$ receives a term (e.g. K) on M that must match the pattern N. Here, $\widetilde{x}$ is a list of pattern variables, binding into N and P, that is used to extract subterms from K that will then be substituted for the occurrences of $\widetilde{x}$ within the continuation P. Unlike the presentation in [21], we here use a typed version of the language: thus the types of the pattern variables are found in the list $\widetilde{T}$ where $|\widetilde{x}| = |\widetilde{T}|$, and likewise, in the restriction $(\nu x : T)\, P$, we annotate the name x bound in P with its type T.

The selection construct $\mathbf{case}\, \widetilde{\varphi} : \widetilde{P}$ is a shorthand for a list of cases and is to be understood as saying: If condition φ_i is entailed by the assertions Ψ, we continue as P_i. If more than one condition is entailed, the process is chosen non-deterministically. This construct thus generalises the choice and matching operators of the π-calculus.

Higher-order communication is handled by representing processes as terms, thus allowing them to be communicated. We assume the existence of assertions of the form $M \Leftarrow P$. By writing such an assertion, M becomes a *handle* of the process P, and we can then send P by sending its handle. Thence M may be used to activate the process P, and for this we use the only construct that is new to the higher-order setting, the invocation construct $\mathbf{run}\, M$. Note that the set of processes may itself be included in the set of terms, thus allowing assertions of the form $P \Leftarrow P$ whereby a process becomes a handle for itself.

Lastly, an assertion $(\!|\Psi|\!)$ is said to be *guarded*, if it occurs as a subterm of an input or output, and *unguarded* otherwise. The authors in [21] impose the restriction that no assertion may occur unguarded in the processes in a conditional expression $\mathbf{case}\, \widetilde{\varphi} : \widetilde{P}$, nor in a replicated process $!P$, nor in processes spawned by a $\mathbf{run}\, M$ operator. We say that processes conforming to this criterion are *well-formed*, and we shall only consider well-formed processes in the following.

2.2 Reduction semantics

Unlike previous presentations such as [4, 21] we here use reduction semantics, as this will simplify our account of the generic type system. As in other reduction semantics for process calculi, we introduce a notion of structural congruence, $\equiv_S$, as the least congruence on process terms containing α-equivalence, the commutative monoidal rules for parallel composition, and the rule for scope extrusion:

$$[\text{S-SCOPE}]\ (\nu x : T)\, P \mid Q \equiv_S (\nu x : T)\, (P \mid Q) \quad \text{if } x \# Q$$

$$[\text{E-RES}]\frac{\Psi \rhd P \underline{\ggg} P'}{\Psi \rhd (\nu x:T)P \underline{\ggg} (\nu x:T)P'}\ (x\#\Psi) \qquad [\text{E-CASE}]\frac{\Psi \Vdash \varphi_i}{\Psi \rhd \mathbf{case}\ \widetilde{\varphi} : \widetilde{P} \underline{\ggg} P_i}$$

$$[\text{E-STRUCT}]\frac{P \equiv_S P'}{\Psi \rhd P \underline{\ggg} P'} \qquad [\text{E-RUN}]\frac{\Psi \Vdash M \Leftarrow P}{\Psi \rhd \mathbf{run}\ M \underline{\ggg} P}$$

$$[\text{E-PAR}]\frac{\Psi \otimes \mathscr{F}_\Psi(Q) \rhd P \underline{\ggg} P'}{\Psi \rhd P \mid Q \underline{\ggg} P' \mid Q}\ (\mathscr{F}_\nu(Q)\#\Psi, \mathscr{F}_\nu(P), P) \qquad [\text{E-REP}]\frac{}{\Psi \rhd\ !P \underline{\ggg} P \mid\ !P}$$

$$[\text{R-COM}]\frac{\Psi \Vdash M \leftrightarrows K}{\Psi \rhd \overline{M}N[\widetilde{x} := \widetilde{L}].P \mid \underline{K}(\lambda \widetilde{x}:\widetilde{T})N.Q \to P \mid Q[\widetilde{x} := \widetilde{L}]}$$

$$[\text{R-EVAL}]\frac{\Psi \rhd P \underline{\ggg} Q \quad \Psi \rhd Q \to P'}{\Psi \rhd P \to P'} \qquad [\text{R-RES}]\frac{\Psi \rhd P \to P'}{\Psi \rhd (\nu x:T)P \to (\nu x:T)P'}\ (x\#\Psi)$$

$$[\text{R-PAR}]\frac{\Psi \otimes \mathscr{F}_\Psi(Q) \rhd P \to P'}{\Psi \rhd P \mid Q \to P' \mid Q}\ (\mathscr{F}_\nu(Q)\#\Psi, \mathscr{F}_\nu(P), P)$$

Figure 1: Reduction semantics for the HOΨ-calculus

We introduce a parametrised, asymmetric *evaluation relation* $\cdot \rhd \cdot \underline{\ggg} \cdot$ to properly handle case expressions and unfolding of **run** M terms, both of which may depend on the assertions currently in effect. It replaces the usual structural congruence rule in the reduction semantics to ensure that neither of these operations may be reversed by a reverse reading of the rules, whilst including $\equiv_S$ for the other kinds of process rewrites where symmetry is unproblematic. The *reduction relation* $\cdot \rhd \cdot \to \cdot$, including the evaluation relation, is then given by the rules in figure 1, and reductions are thus on the form $\Psi \rhd P \to P'$, i.e. relative to a global Ψ containing the assertions currently in effect.

New assertions $(\!|\Psi|\!)$ may also appear in the syntax and therefore become enabled during the evolution of the program. These are collected by the *frame function* $\mathscr{F}_\Psi(P)$ in the [R-PAR] and [E-PAR] rules; and likewise are any new names $(\nu x:T)$ collected by $\mathscr{F}_\nu(P)$, used in the side conditions to ensure freshness of x w.r.t. Ψ and the process in parallel composition. The relevant clauses for $\mathscr{F}_\Psi(P)$ and $\mathscr{F}_\nu(P)$ are:

$$\begin{aligned} \mathscr{F}_\Psi(P \mid Q) &\triangleq \mathscr{F}_\Psi(P) \otimes \mathscr{F}_\Psi(Q) & \qquad \mathscr{F}_\nu(P \mid Q) &\triangleq \mathscr{F}_\nu(P) \cup \mathscr{F}_\nu(Q) \\ \mathscr{F}_\Psi((\nu x:T)P) &\triangleq \mathscr{F}_\Psi(P) & \qquad \mathscr{F}_\nu((\nu x:T)P) &\triangleq \{x\} \cup \mathscr{F}_\nu(P) \\ \mathscr{F}_\Psi((\!|\Psi|\!)) &\triangleq \Psi \end{aligned}$$

and with all remaining clauses of the forms $\mathscr{F}_\Psi(P) \triangleq 1$ and $\mathscr{F}_\nu(P) \triangleq \emptyset$ respectively.

3 The generic type system

The goal of our generic type system is to be able to instantiate it such that we obtain a sound type system for a given HOΨ-calculus instance. As in other type systems, we need to describe when processes are well-typed, but since we in the HOΨ-calculus also have terms, conditions and assertions, we shall therefore also need a way to decide when *they* are well-typed. However, since these nominal datatypes are parameters to the HOΨ-calculus, we cannot specify a set of type rules for them, as we can with processes.

Instead, such rules must likewise be provided as parameters to create an instance of the generic type system, and these rules must then satisfy a number of requirements, here denoted *instance assumptions*, which we shall need in the proof for subject reduction. We describe them in detail below, in section 3.3.

3.1 Types and type judgements

Types can contain names, and we assume that the set of types ***Types*** is a nominal datatype ranged over by T; however, we do not allow substitution of terms for names *inside* types.[1] Furthermore, we need the concept of a type environment Γ to record the types of free names; thus Γ is a partial function with finite support $\Gamma : \mathcal{N} \rightharpoonup \textbf{\textit{Types}}$. We can think of Γ as a set of tuples $\Gamma \subseteq \mathcal{N} \times \textbf{\textit{Types}}$ where $(x,T) \in \Gamma$ if $\Gamma(x) = T$, and we write $\Gamma, x : T$ to denote the type environment Γ extended by the name x with type T.

As usual, our type judgments will be relative to a type environment Γ. However, due to the presence of assertions which may affect the well-typedness of a process, term, condition, or indeed an assertion, our type judgments must also be relative to a global assertion Ψ. As it may be composed with assertions appearing in a process, we shall therefore also need the notion of a specialisation preorder on assertions. We say that $\Psi_1 \leq \Psi_2$ if there exists a Ψ such that $\Psi_2 = \Psi_1 \otimes \Psi$, and $\mathrm{n}(\Psi_1) \subseteq \mathrm{n}(\Psi_2)$.

Given the above, type judgements for processes will be of the form $\Gamma, \Psi \vdash P$. As previously mentioned, the type rules for terms, assertions and conditions will depend on how these parameters are defined for a specific instance of the HOΨ-calculus, and they must therefore be provided as part of the instantiation of the generic type system. However, like type judgments for processes, they must also be relative to a type environment Γ and a global Ψ, so we require that they be of the form $\Gamma, \Psi \vdash \mathscr{J}$, where $\mathscr{J}$ is defined by the formation rules:

$$\mathscr{J} \triangleq M : T \;\Big|\; \varphi \;\Big|\; \Psi$$

3.2 Channel compatibility

When we type an input or output prefix term, the type of the subject M and the type of the object (the term transmitted on channel M) must be compatible w.r.t. a *compatibility predicate* $\hookleftarrow\!\!\rho$ that describes which types of values can be carried by channels of a given types. Thus, $T_1 \hookleftarrow\!\!\rho\, T_2$ denotes that channels of type T_1 can carry terms of type T_2, and we require that the set of types be defined such that this holds. Furthermore, we distinguish between output compatibility $\hookleftarrow\!\!\rho^+$, and input compatibility $\hookleftarrow\!\!\rho^-$, and we write $T_1 \hookleftarrow\!\!\rho\, T_2$ if both $T_1 \hookleftarrow\!\!\rho^+ T_2$ and $T_1 \hookleftarrow\!\!\rho^- T_2$.

As an example, consider the channel types in the sorting system by Milner [19]. Here, a name has type $\mathrm{ch}(T)$, if it is a channel that can be used to transmit names of type T, so in that case we would therefore require that $\mathrm{ch}(T) \hookleftarrow\!\!\rho\, T$.

In our definition of compatibility, we assume given a subtype ordering $\leq$ on types. If $T_1 \leq T_2$, then a term of T_1 can be used wherever a term of type T_2 is needed. Thus we require the usual subsumption rule for types, namely that a term of a given type T_1 can also be typed with a supertype T_2:

$$[\textsc{subsume}]\frac{\Gamma, \Psi \vdash M : T_1 \quad T_1 \leq T_2}{\Gamma, \Psi \vdash M : T_2}$$

The compatibility predicate for a type T must further satisfy the following requirements w.r.t. the subtyping relation:

[1] As we shall see in our examples of instantiations of the type system, a type T may itself contain a type environment Γ, which thus may contain names with type annotations. By this requirement, we disallow that such names may be substituted for terms.

$$
\begin{aligned}
[\text{T-ENV-WEAK}]\ & \Gamma,\Psi \vdash \mathscr{J} \implies \Gamma,x:T,\Psi \vdash \mathscr{J} \\
[\text{T-ENV-STRENGTH}]\ & \Gamma,x:T,\Psi \vdash \mathscr{J} \wedge x \notin \mathrm{n}(\mathscr{J}) \implies \Gamma,\Psi \vdash \mathscr{J} \\
[\text{T-COMP-TERM}]\ & \Gamma,\Psi \vdash M[\widetilde{x}:=\widetilde{L}] : F(\widetilde{T}) \implies \Gamma,\Psi \vdash \widetilde{L}:\widetilde{T} \\
[\text{T-ASS-WEAK}]\ & \Gamma,\Psi \vdash \mathscr{J} \wedge \Psi \leq \Psi' \wedge \mathrm{n}(\Psi') \subseteq \mathrm{dom}(\Gamma) \implies \Gamma,\Psi' \vdash \mathscr{J} \\
[\text{T-WEAK-CHANEQ}]\ & \Psi \Vdash M_1 \leftrightarrows M_2 \implies \Psi \otimes \Psi' \Vdash M_1 \leftrightarrows M_2 \\
[\text{T-SUBS}]\ & \Gamma,\Psi \vdash \widetilde{L}:\widetilde{T} \wedge \Gamma,\widetilde{x}:\widetilde{T},\Psi \vdash \mathscr{J} \implies \Gamma,\Psi \vdash \mathscr{J}[\widetilde{x}:=\widetilde{L}] \\
[\text{T-EQUAL}]\ & \Gamma,\Psi \vdash M:T \wedge \Psi \Vdash M \leftrightarrows N \implies \Gamma,\Psi \vdash N:T \\
[\text{T-ENV-CLAUS}]\ & \Gamma,\Psi \vdash M:T \wedge T \curvearrowleft \Gamma' \implies \mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(\Gamma') \\
[\text{T-WEAK-ASS-CLAUS}]\ & \Psi \Vdash M \Leftarrow P \wedge \Gamma,\Psi \vdash M \Leftarrow P \wedge \Psi \leq \Psi' \wedge \mathrm{n}(\Psi) \subseteq \Gamma \implies \Psi' \Vdash M \Leftarrow P \\
[\text{T-SUBS-RUN}]\ & \Gamma,\Psi \vdash M:T \wedge T \curvearrowleft \Gamma' \wedge \Psi \Vdash M[\widetilde{x}:=\widetilde{L}] \Leftarrow P \implies \Gamma',\Psi \vdash P
\end{aligned}
$$

Figure 2: Instance assumptions for the generic type system.

1. If a channel type can carry two distinct types, then the types have to be related by the subtype ordering. That is, if $d \in \{+,-\}$, $T \leftrightarrow\!\!\!\rho^d\ T_1$ and $T \leftrightarrow\!\!\!\rho^d\ T_2$ with $T_1 \neq T_2$, then $T_1 \leq T_2$ or $T_2 \leq T_1$.
2. Output compatibility is *contravariant*. That is, if $T \leftrightarrow\!\!\!\rho^+\ T_2$ and $T_1 \leq T_2$, then also $T \leftrightarrow\!\!\!\rho^+\ T_1$. This requirement mirrors that of [22]. If $T_1 \leq T_2$, then a term of type T_1 can be used where ever a term of type T_2 is needed, and a channel that outputs terms of the more general type T_2 can therefore be used, where ever a channel of the specialized type T_1 is required.
3. Input compatibility is *covariant*. That is, if $T \leftrightarrow\!\!\!\rho^-\ T_1$ and $T_1 \leq T_2$, then also $T \leftrightarrow\!\!\!\rho^-\ T_2$. This requirement, too, mirrors that of [22]. Here, if $T_1 \leq T_2$, a channel that accepts terms of type T_1 can also be used to accept terms of type T_2.

3.3 Instance assumptions

In order to ensure soundness, we introduce a collection of assumptions, given in Figure 2, that must hold for an instance of the generic type system to be valid. They pertain to the type judgments $\Gamma,\Psi \vdash \mathscr{J}$ for terms, conditions and assertions, which, as previously mentioned, we cannot specify in advance, but on which we must nevertheless impose certain restrictions to allow us to prove subject reduction for the generic type system. Specifically, the assumptions will guarantee that the properties of weakening and strengthening and the substitution lemma will hold for any instance that satisfies these assumptions.

Firstly, we require every instance of our generic type system to satisfy certain natural requirements about the use of type environments Γ; these are similar to those of Hüttel in [12]. The assumptions [T-ENV-WEAK], [T-ENV-STRENGTH], [T-COMP-TERM] and [T-ASS-WEAK] are the usual requirements of weakening and strengthening; these must hold for type environments as well as for assertions. [T-WEAK-CHANEQ] tells us that channel equivalence is closed under weakening of assertions. The assumptions [T-SUBS] and [T-EQUAL] tell us that typability must be invariant under substitution and channel equivalence.

Other assumptions are particular to the higher-order setting and thus new. Here, one particularly important question is *which* type environment Γ a process P should be typed in relation to, if P is transmitted using the higher-order process mobility construct, with M as a handle for P. To solve this, we write $T \curvearrowleft \Gamma$ to express that if M is a handle for some process P and has type T, then we can *extract*

$$[\text{T-IN}]\frac{T \leftrightharpoons T' \quad \Gamma,\Psi\vdash M:T \quad \Gamma,\widetilde{x}:\widetilde{T},\Psi\vdash N:T' \quad \Gamma,\widetilde{x}:\widetilde{T},\Psi\vdash P}{\Gamma,\Psi\vdash \underline{M}(\lambda\widetilde{x}:\widetilde{T})N.P} \qquad [\text{T-RUN}]\frac{T \curvearrowleft \Gamma' \quad \Gamma,\Psi\vdash M:T \quad \Psi \Vdash M \Leftarrow P \quad \Gamma',\Psi\vdash P}{\Gamma,\Psi\vdash \mathbf{run}\ M} \qquad [\text{T-OUT}]\frac{T \leftrightharpoons T' \quad \Gamma,\Psi\vdash M:T \quad \Gamma,\Psi\vdash N:T' \quad \Gamma,\Psi\vdash P}{\Gamma,\Psi\vdash \overline{M}N.P}$$

$$[\text{T-PAR}]\frac{\Gamma,\mathscr{F}_{\mathcal{V}}(Q),\Psi\otimes\mathscr{F}_{\Psi}(Q)\vdash P \quad \Gamma,\mathscr{F}_{\mathcal{V}}(P),\Psi\otimes\mathscr{F}_{\Psi}(P)\vdash Q}{\Gamma,\Psi\vdash P \mid Q}\ \begin{pmatrix}\mathscr{F}_{\mathcal{V}}(P)\,\#\Psi, \mathscr{F}_{\mathcal{V}}(Q), Q\\ \mathscr{F}_{\mathcal{V}}(Q)\,\#\Psi, \mathscr{F}_{\mathcal{V}}(P), P\end{pmatrix}$$

$$[\text{T-NEW}]\frac{\Gamma,x:T,\Psi\vdash P}{\Gamma,\Psi\vdash(\nu x:T)P}\ (x\#\Psi) \qquad [\text{T-NIL}]\frac{}{\Gamma,\Psi\vdash\mathbf{0}} \qquad [\text{T-REPL}]\frac{\Gamma,\Psi\vdash P}{\Gamma,\Psi\vdash\,!P}$$

$$[\text{T-CASE}]\frac{\Gamma,\Psi\vdash\varphi_i \quad \Gamma,\Psi\vdash P_i}{\Gamma,\Psi\vdash\mathbf{case}\ \widetilde{\varphi}:\widetilde{P}} \qquad [\text{T-ASSERT}]\frac{\Gamma,\Psi\vdash\Psi'}{\Gamma,\Psi\vdash(\!|\Psi'|\!)}$$

Figure 3: Type judgements for processes

the type environment Γ for typing P from the type T of the handle M. This is thus another requirement we impose on how the set of types must be defined. As a simple example, suppose that every type of a term would consist of a channel component and a run type component (T,Γ); then we could define the $\curvearrowleft$ relation to be $(T,\Gamma)\curvearrowleft\Gamma$.

The new assumptions are as follows:

- The assumption [T-ENV-CLAUS] tells us that that the type environment extracted from the type of a handle M must mention at least the free names of M.
- The assumption [T-WEAK-ASS-CLAUS] is necessary to prove weakening of assertion environments; i.e. by allowing unused assertions to be added. It states that if M is a handle for P, then M must still remain a handle for the same process P if the assertion environment is weakened.
- The assumption [T-SUBS-RUN] is needed to ensure that typability is preserved by substitution also in the higher-order case. It states that if a term M becomes a handle for a new process P after a substitution, then the new process must still be well-typed in the environment we obtain from M's type T.

3.4 Type rules for processes

Unlike the aforementioned type rules for terms, conditions and assertions, the type rules for processes are common to every instance. As in [12], we only consider type judgements that are *well-formed*; that is, if $\mathrm{n}(\Psi)\cup\mathrm{n}(P)\subseteq\mathrm{dom}(\Gamma)$, so every name mentioned in the term or process in the judgement is bound in the type environment. The rules are given in Figure 3; they are mostly similar to those of [12], except for the rule [T-RUN] used to type the **run** M construct, which is the only construct that is new to the higher-order setting.

We shall comment on the rules in some detail: In the rule for input, [T-IN], the subject M must have type T, which must be compatible with the type T' according to the aforementioned compatibility relation $\leftrightharpoons$. The pattern N must then have this type T', given the assumptions that the variables $\widetilde{x}$ have types $\widetilde{T}$,

and lastly, the continuation P must be well-typed given these assumptions. The output rule, [T-OUT] then mirrors the input rule as usual. In both cases, the type judgment $\Gamma, \Psi \vdash M : T$ appears in the premise, and as previously mentioned, the rules for this judgment must be provided as part of the instantiation.

In the rule [T-PAR] we require that for a parallel composition $P \mid Q$ to be typable, P and Q must both be typable within type environments and assertions that add information extracted from the other component; thus we here overload the function $\mathscr{F}_V(\cdot)$ for $(\nu x : T)\, P$ to mean $\mathscr{F}_V((\nu x : T)\, P) \triangleq x : T, \mathscr{F}_V(P)$. This is a natural requirement, since P can, among other things, mention handles for processes established in Q. The side condition then asserts that all new names declared in P, using the $(\nu x : T)$ construct, must be fresh for Ψ *and* both the free and new names occurring in Q, and vice versa for Q, similar to the side conditions for the [E-PAR] and [R-PAR] rules in the semantics.

Likewise in the rule [T-NEW], we require that the new name x must be fresh for Ψ, again mirroring the side conditions in the corresponding semantic rules [E-RES] and [R-RES], and P must then be well-typed given the assumption that x has type T.

The rules for the nil process and replication, [T-NIL] and [T-REPL] are as usual, and the rule for **case** $\widetilde{\varphi} : \widetilde{P}$ is also quite straightforward. Here, we write $\Gamma, \Psi \vdash \varphi_i$ and $\Gamma, \Psi \vdash P_i$ to say that every condition φ_i in the list of conditions $\widetilde{\varphi}$, and every process P_i in the list of processes $\widetilde{P}$, must be well-typed w.r.t. the same Γ and Ψ. As in the cases for input and output above, the rules for concluding $\Gamma, \Psi \vdash \varphi_i$ must be provided as part of the instantiation; and likewise for concluding $\Gamma, \Psi \vdash \Psi'$, which appears in the premise of the [T-ASSERT] rule.

Lastly, since a key motivation for the present type system is the ability to type higher-order behaviour, we must be able to describe what can happen when a handle $M \Leftarrow P$ is released by a **run** M. This is handled by the rule [T-RUN], which states that **run** M is well-typed for Γ and Ψ if M is a handle for P in Ψ and P is well-typed in the environment Γ' extracted from M, using the aforementioned $\curvearrowleft$ relation.

4 Properties of the generic type system

Type systems normally ensure two properties of well-typed programs: a *subject reduction* property guarantees that a well-typed program remains well-typed under reduction; and a *safety* property ensures that if a program is well-typed then a certain safety predicate holds. The latter will depend on the particular instance of the type system and must therefore be shown individually, for each instance, but subject reduction can be shown for the generic type system. We establish this through a series of lemmas, beginning with the usual results of *weakening* and *strengthening* of the type environment:

Lemma 1 (Weakening and strengthening)**.**

- *If* $\Gamma, \Psi \vdash P$ *then* $\Gamma, x : T, \Psi \vdash P$
- *If* $\Gamma, x : T, \Psi \vdash P$ *and* $x \# P, \Psi$ *then* $\Gamma, \Psi \vdash P$

A similar result holds for assertions. Any process that is well-typed remains well-typed after a composition of any assertion in the assertion environment, so long as all names in the new assertion environment are in the support of the type environment:

Lemma 2 (Assertion environment weakening)**.** *If* $\Gamma, \Psi \vdash P$, $\mathrm{n}(\Psi') \subseteq \mathrm{dom}(\Gamma)$ *and* $\Psi \leq \Psi'$ *then* $\Gamma, \Psi' \vdash P$.

This lemma is necessitated by the syntax of the HOΨ-calculus itself, which allows guarded assertions in continuations to become unguarded after a reduction. It is in the proof of this result that the instance assumptions [T-ASS-WEAK], [T-ENV-CLAUS] and [T-WEAK-ASS-CLAUS] become important.

As we here use reduction semantics with an asymmetric evaluation relation to handle unfolding of **case** and **run** expressions, we shall also need two results that describe how frames can evolve during

evaluation. The former, given in Lemma 3 is used in the proof of subject reduction (Theorem 1) to find any new assertions that may have become composed onto the pre-existing assertion environment after a reduction. The latter, given in Lemma 4 states that the assertions in a process are unaltered by an evaluation: This is mainly ensured by the criterion for well-formed processes, asserting that all processes under replication or in a **case** expression, and all processes spawned by a **run** M operator, may not contain unguarded assertions. The proof then establishes that the property of being assertion-guarded is preserved by the evaluation relation $\ggg$.

Lemma 3 (Frame post reduction)**.** *If* $\Psi \rhd P \to P'$ *then* $\mathscr{F}_\Psi(P) \leq \mathscr{F}_\Psi(P')$

Lemma 4 (Frame post evaluation)**.** *If* $\Psi \rhd P \underline{\ggg} P'$ *then* $\mathscr{F}_\Psi(P) = \mathscr{F}_\Psi(P')$.

The above lemmas can now be used to prove that a well-typed process remains well-typed after an evaluation:

Lemma 5 (Subject evaluation)**.** *If* $\Gamma, \Psi \vdash P \wedge \Psi \rhd P \underline{\ggg} P'$ *then* $\Gamma, \Psi \vdash P'$.

Lastly, we need a standard result of *substitution*, which states that a well-typed process remains well-typed after a well-typed substitution. The proof of this lemma requires the instance assumptions [T-SUBS] and [T-SUBS-RUN].

Lemma 6 (Subject substitution)**.** *If* $\Gamma, \widetilde{x} : \widetilde{T}, \Psi \vdash P$ *and* $\Gamma, \Psi \vdash \widetilde{L} : \widetilde{T}$ *then* $\Gamma, \Psi \vdash P[x := \widetilde{L}]$.

This, at last, gives us our main result:

Theorem 1 (Subject reduction)**.** *If* $\Gamma, \Psi \vdash P \wedge \Psi \rhd P \to P'$ *then* $\Gamma, \Psi \vdash P'$.

Outline. Induction in the reduction rules. In many of the cases, the instance assumptions are needed. An example is that in the case of the [R-COM] rule, the substitution assumption [T-SUBS] is needed to ensure that the substitution of the received message can be well-typed and the weakening assumptions [T-ENV-WEAK] and [T-ASS-WEAK] are needed to ensure that the resulting process can be typed within the same type environment as before. □

The subject reduction theorem holds for all valid instances of the generic type system. This is all that we can guarantee in our generic setting, as a notion of *safety* will also depend on a definition of runtime error, which will be specific to each instance. Safety must therefore be proved individually for each instance.

5 Instances of the generic type system

We now show how our generic type system can be applied to provide sound type systems for higher-order process calculi. We first consider type systems for a version of the HOπ-calculus [24], and then a type system for the ρ-calculus [18] introduced by Meredith and Radestock.

5.1 The Higher-Order π-calculus

Parrow et al. [21] give several examples of HOΨ-instances with process mobility: for example, by including the set of processes $\mathscr{P}_\Psi$ in $\mathbb{T}$, a process P may appear as the object of an output. If for all $P \in \mathscr{P}_\Psi . P \Leftarrow P$ is entailed by all assertions, a language similar to Thomsen's Plain CHOCS [27] is obtained, and by further allowing both names and processes to appear as objects of an output, we get a simplified version of Sangiorgi's HOπ-calculus, similar to the one described in [20]. We set the parameters for $\mathbb{T}, \mathbb{C}$ and entailment thus:

$$\mathbb{T} \triangleq \mathscr{N} \cup \mathscr{R}_\Psi$$
$$\mathbb{C} \triangleq \{\, a \leftrightarrows b \mid a,b \in \mathscr{N} \,\} \cup \{\, P \Leftarrow Q \mid P,Q \in \mathscr{R}_\Psi \,\} \cup \{\, \top \,\}$$
$$\Vdash \triangleq \{\, (1, a \leftrightarrows a) \mid a \in \mathscr{N} \,\} \cup \{\, (1, P \Leftarrow P) \mid P \in \mathscr{R}_\Psi \,\} \cup \{\, (1, \top) \,\}$$

and (initially) with $\mathbb{A} \triangleq \{\emptyset\}$, $\otimes \triangleq \cup$ and $1 \triangleq \emptyset$. We also include the symbol $\top$ in $\mathbb{C}$ to represent a condition that is entailed by all assertions, and use that for every condition in a **case** $\widetilde{\varphi} : \widetilde{P}$ construct to obtain a representation of non-deterministic choice. This parameter setting obviously allows unwanted processes such as

$$\overline{a}P.\mathbf{0} \mid \underline{a}(\lambda x)x.\overline{x}b.\mathbf{0} \to \overline{P}b.\mathbf{0}$$

where the process P is substituted for the *subject* x in the output construct $\overline{x}b.\mathbf{0}$ after a reduction step. However, we can now use our generic type system to create an instantiation that will disallow such possibilities. We define the types of terms as:

$$T \in \textbf{\textit{Types}} ::= \mathrm{ch}(T) \;\big|\; \mathrm{drop}(\Gamma)$$

The behaviour of channels and first-order variables is captured in the same manner as the simple sorting system for the π-calculus [19]. Process terms and higher-order variables will have the type $\mathrm{drop}(\Gamma)$, where the processes are well-typed in Γ. Type errors can then be expressed as a simple error predicate, with

$$\frac{\Gamma, \Psi \vdash M : \mathrm{drop}(\Gamma')}{\Gamma, \Psi \vdash \underline{M}(\lambda x)x.Q \to \mathbf{WRONG}} \qquad \frac{\Gamma, \Psi \vdash M : \mathrm{ch}(T)}{\Gamma, \Psi \vdash \mathbf{run}\ M \to \mathbf{WRONG}}$$

as the most relevant rules. We now define the instance parameters:

$$[\text{T-CON}]\frac{}{\Gamma, \Psi \vdash \top} \qquad [\text{T-ASS}]\frac{P : \mathrm{drop}(\Gamma) \in \Psi'}{\Gamma, \Psi \vdash (\!|\Psi'|\!)} \qquad [\text{T-END}]\frac{}{\mathrm{drop}(\Gamma) \curvearrowleft \Gamma}$$

$$[\text{T-CHA}]\frac{}{\mathrm{ch}(T) \leftharpoonup T} \qquad [\text{TERM}_1]\frac{\Gamma(x) = \mathrm{ch}(T)}{\Gamma, \Psi \vdash x : \mathrm{ch}(T)} \qquad [\text{TERM}_2]\frac{P : \mathrm{drop}(\Gamma') \in \Psi \quad \Gamma', \Psi \vdash P}{\Gamma, \Psi \vdash P : \mathrm{drop}(\Gamma')}$$

Here we let the type environment in a drop type be the same type environment that is exposed to the processes, when it is defined as an object in an output prefix, i.e. if we have $\Gamma, \Psi \vdash \overline{a}P$, we want $\Gamma, \Psi \vdash P : \mathrm{drop}(\Gamma)$. In this way, when we run the process, we can recall the bound variables and their types at the time when the process was sent. To implement this, we shall use the (previously unused) assertions Ψ as type environments for processes. Thus we redefine $\mathbb{A}$ as follows:

$$\mathbb{A} \triangleq \wp\big(\{\, P : T \;\big|\; P \in \mathscr{P} \wedge T \in \textbf{\textit{Types}} \,\}\big)$$

We can now show safety for the type system instance:

Theorem 2. *If* $\Gamma, \Psi \vdash P$ *then* $P \not\to$ ***WRONG***.

The proof is by induction in the rules of $\Gamma, \Psi \vdash P$. Details are given in [2].

5.2 A type system for termination

We now turn our attention to an instance of the generic type system that captures a liveness property. Demangeon et al. [7] present a type system for checking termination in variants of the HOπ-calculus: for any well-typed process P we have that $P \to^* P' \not\to$. These authors study HOpi_2, a higher-order process calculus in which only processes can be communicated. The syntax of HOpi_2 is given by the formation rules

$$P ::= \mathbf{0} \;\mid\; a(X).P \;\mid\; \overline{a}\langle Q\rangle.P \;\mid\; P_1 \mid P_2 \;\mid\; (\nu a : T)P \;\mid\; X$$

In this type system, processes P are typed with a type n, where n is a natural number called the *level* of P. Names a have types of the form $\mathrm{ch}^k(\diamond)$, where $\diamond$ denotes the type of processes and k is a natural number, the level of a. This is interpreted as saying that a is only used to carry processes whose level n is less than k. Type judgements are of the form $\Gamma \vdash P : n$. The type rules, shown below, ensure that the level of processes that are sent on any channel a will be strictly smaller than that of a.

$$[\textsc{In}]\ \frac{\Gamma, X : (k-1) \vdash P : n \quad \Gamma(a) = \mathrm{ch}^k(\diamond)}{\Gamma \vdash a(X).P : n} \qquad [\textsc{Out}]\ \frac{\Gamma \vdash Q : m \quad \Gamma \vdash P : n \quad \Gamma(a) = \mathrm{ch}^k(\diamond) \quad m < k}{\Gamma \vdash \overline{a}\langle Q\rangle.P : \max(k,n)} \qquad [\textsc{Nil}]\ \frac{}{\Gamma \vdash \mathbf{0} : 0}$$

$$[\textsc{New}]\ \frac{\Gamma, a : \mathrm{ch}^k(\diamond) \vdash P : n}{\Gamma \vdash (\nu a : T)P : n} \qquad [\textsc{Par}]\ \frac{\Gamma \vdash P : m \quad \Gamma \vdash Q : n}{\Gamma \vdash P \mid Q : \max(m,n)} \qquad [\textsc{Var}]\ \frac{\Gamma(X) = n}{\Gamma \vdash X : n}$$

It is straightforward to represent the HOpi_2 calculus as an instance of the Higher-Order Ψ-calculus, using a variant of the parameter setting described in section 5.1. In order to represent the type system, we introduce assertions of the form

$$\Psi ::= n \;\mid\; n^- \;\mid\; n^+$$

We use assertions to indicate in which way a channel is to be used; an input use can only be typed in the presence of an assertion n^- and output use must be used with an assertion n^+. We have that $n \otimes n^- = n^- \otimes n = n$; that $n \otimes n^+ = n^+ \otimes n = n$; and that $n_1 \otimes n_2 = \max(n_1, n_2)$. We distinguish explicitly between input uses ($\mathrm{ch}^k_-(\diamond)$) and output uses ($\mathrm{ch}^k_+(\diamond)$) of channels:

$$T \in \textbf{\textit{Types}} ::= n \;\mid\; \mathrm{ch}^k_-(\diamond) \;\mid\; \mathrm{ch}^k_+(\diamond)$$

and we let $\mathrm{ch}^n(\diamond) \curvearrowleft (n-1)$ and $\mathrm{ch}^n(\diamond) \curvearrowleft k$ whenever $k < n$. Type judgements are of the form $\Gamma, m \vdash M : T$ for terms and $\Gamma, m \vdash P$ for processes. We represent the judgement $\Gamma \vdash P : n$ as $\Gamma, n \vdash P$. The type rules for channels are thus:

$$[\textsc{Ch-in}]\ \frac{\Gamma(a) = \mathrm{ch}^k_-(\diamond)}{\Gamma, n^- \vdash a : \mathrm{ch}^k_-(\diamond)} \qquad [\textsc{Ch-out}]\ \frac{\Gamma(a) = \mathrm{ch}^k_+(\diamond)}{\Gamma, n^+ \vdash a : \mathrm{ch}^k_+(\diamond)}$$

5.3 The ρ-calculus

The Reflective Higher-Order calculus of Meredith and Radestock [18] is less well-known than e.g. CHOCS and HOπ, so we recall it in some detail. Unlike other calculi, the ρ-calculus does not assume an infinite set of names: instead, names and processes are both built from the same syntax, so names are structured terms, rather than atomic entities. The syntax for both processes and names is given by the formation rules:

$$\begin{aligned} P &::= \mathbf{0} \;\mid\; P \mid P \;\mid\; x\langle\!|P|\!\rangle \;\mid\; x(y).P \;\mid\; \ulcorner x \urcorner \\ x, y &::= \ulcorner P \urcorner \end{aligned}$$

where the syntax for names is simply $\ulcorner P \urcorner$, pronounced *quote P*. Names can be passed around as in the π-calculus, as well as un-quoted (called *drop*), and thus higher-order behaviour becomes an inherent property of the calculus, rather than just an extension on top of an already computationally complete language.

The parallel, and the input construct $x(y).P$, are similar to their π-calculus counterparts. The *lift* operation, $x\langle\!|P|\!\rangle$ is an output construct that quotes the process P, thereby creating the *name* $\ulcorner P \urcorner$, and sends it out on x; thus the calculus can generate new names at runtime without the need of a ν-operator. The converse of lift is the *drop* operation, $\urcorner x \ulcorner$: it is a request to run the process within a name, by removing the quotes around it. This is not performed by a reduction, but rather by a form of substitution

$$\urcorner x \ulcorner \{\ulcorner P \urcorner / x'\} = P \qquad \text{if } x \equiv_{\mathcal{N}} x'$$

where the entire *process* $\urcorner x \ulcorner$ is replaced with the process P found within the substituted name, similar to how process variables are replaced by processes in e.g. HOπ. Notably, this means that if x is a *free* name, then $\urcorner x \ulcorner$ will be a *deadlock*, since x can never be touched by a substitution at runtime. Otherwise, substitution is the standard, capture-avoiding substitution of names for names, and note in particular that substitution does *not* recur into processes under quotes; i.e. $\ulcorner P \urcorner \{x/y\} = \ulcorner P \urcorner$ if $y \not\equiv_{\mathcal{N}} \ulcorner P \urcorner$ regardless of whether the name y exists somewhere within $\ulcorner P \urcorner$.

The reduction semantics is given by the standard rules for parallel composition and structural congruence (as in e.g. the π-calculus) plus the following rule for communication:

$$[\rho\text{-COM}]\frac{x_1 \equiv_{\mathcal{N}} x_2}{x_1(y).P \mid x_2\langle\!|Q|\!\rangle \to P\{\ulcorner Q \urcorner / y\}}$$

One subtlety of this calculus concerns the notion of structural congruence, $\equiv$. It is the usual least congruence on processes, containing α-equivalence, $\equiv_\alpha$, and the abelian monoid rules for parallel composition with $\mathbf{0}$ as the unit element. However, with *structured* terms as names, $\equiv_\alpha$ in turn requires a notion of *name equivalence*, written $\equiv_{\mathcal{N}}$, that is also used for comparing subjects in the [ρ-COM] rule above. It is defined as the smallest equivalence relation on *quoted* processes, closed forward under the rules:

$$[\rho\text{-NAMEEQ}_1]\frac{P \equiv Q}{\ulcorner P \urcorner \equiv_{\mathcal{N}} \ulcorner Q \urcorner} \qquad [\rho\text{-NAMEEQ}_2]\frac{}{\ulcorner \urcorner x \ulcorner \urcorner \equiv_{\mathcal{N}} x}$$

This yields a mutual recursion between name equivalence, structural congruence and α-equivalence, albeit one that always terminates as proved in [18], because both the sets of names and processes are well-founded; their smallest elements being $\mathbf{0}$ (the inactive process) and $\ulcorner \mathbf{0} \urcorner$ respectively.

5.3.1 Instantiation as a Ψ-calculus

The ρ-calculus is interesting in the present setting, because it cannot be encoded in the π-calculus in a way that satisfies a number of generally accepted criteria of encodability, similar to those of [10]. This has been established by one of the authors in [17].

The key reason for this impossibility lies in the ability of the ρ-calculus to generate new, *free*, and hence observable, names at runtime, whilst this is not possible in the π-calculus; and, dually, its use of name equivalence, which will equate more names than strict syntactic equality. However, the ρ-calculus *can* be represented in the HOΨ-framework as follows. We define

$$\begin{aligned}
\mathbb{T} &\triangleq \mathcal{N} \cup \{\ulcorner P \urcorner \mid P \in \mathscr{P}_\Psi\} \cup \{\langle\!|\ulcorner P \urcorner|\!\rangle \mid P \in \mathscr{P}_\Psi\} \\
\mathbb{C} &\triangleq \{M \dot{\leftrightarrow} N \mid M,N \in \mathbb{T}\} \cup \{P_1 \equiv P_2 \mid P_1,P_2 \in \mathscr{P}_\Psi\} \\
&\quad \cup \{M \Leftarrow P \mid M \in \mathbb{T} \wedge P \in \mathscr{P}_\Psi\}
\end{aligned}$$

and (initially) with $\mathbb{A} \triangleq \{\emptyset\}$, $\otimes \triangleq \cup$ and $1 \triangleq \emptyset$ as before. Note the two different kinds of terms: we use terms of the form $\ulcorner P \urcorner$ to represent a *statically* quoted name in the ρ-calculus, which can never be dropped and never substituted into. Conversely, we use $\langle\!|\ulcorner P \urcorner|\!\rangle$ for the equivalent of the object of a $x\langle\!|P|\!\rangle$,

which in the ρ-calculus is a *process* that therefore *can* be substituted into, and which later may be dropped. Furthermore, we shall assume that all *bound names* are implemented as distinct atomic names $x \in \mathscr{N}$; this is a trivial conversion, since their structure has no semantic meaning in the ρ-calculus. The encoding is then given by the translation:

$$\begin{array}{rlrl} [\![\mathbf{0}]\!] &= \mathbf{0} & [\![\ulcorner x \urcorner]\!] &= \mathbf{run}\ x \\ [\![P_1 \mid P_2]\!] &= [\![P_1]\!] \mid [\![P_2]\!] & [\![\ulcorner\urcorner P \ulcorner\urcorner]\!] &= \mathbf{0} \\ [\![n(x).P]\!] &= \underline{[\![n]\!]}(\lambda x)\langle\!\langle x \rangle\!\rangle.[\![P]\!] & [\![\ulcorner P \urcorner]\!] &= \ulcorner \mathscr{N}[\![P]\!] \urcorner \\ [\![n\langle\!\langle P \rangle\!\rangle]\!] &= \overline{[\![n]\!]}\langle\!\langle \ulcorner [\![P]\!] \urcorner \rangle\!\rangle.\mathbf{0} & [\![x]\!] &= x \end{array}$$

where $\mathscr{N}[\![P]\!]$ is similar to $[\![P]\!]$ *except* that $\mathscr{N}[\![\ulcorner\urcorner P \ulcorner\urcorner]\!] = \mathbf{run}\ \ulcorner \mathscr{N}[\![P]\!] \urcorner$.

Note the two translations of drop for processes: the process $\ulcorner\urcorner P \ulcorner\urcorner$ has no reduction in the ρ-calculus and is therefore behaviourally equivalent to $\mathbf{0}$; but its counterpart $\mathbf{run}\ \ulcorner P \urcorner$ *might* have a reduction, since $\mathbf{run}\ M$ is not evaluated eagerly in the HOΨ-calculus. For the purpose of preserving operational correspondence, we therefore translate the drop of a *free name* $\ulcorner P \urcorner$ as $\mathbf{0}$, and the drop of an atomic name x as $\mathbf{run}\ x$, since atomic names are bound by construction. However, we cannot do this within *names*, since name equivalence is determined by the structure, rather than the behaviour of the process within quotes. Thus we use the second level translation $\mathscr{N}[\![P]\!]$ for statically quoted names, since these can never be dropped.

Lastly, we shall define entailment such that it contains the rule $\Psi \Vdash \ulcorner P \urcorner \Leftarrow P$, making every term $\ulcorner P \urcorner$ a handle for the process P within, to mirror the duality of names and processes in the ρ-calculus. We furthermore include the following rules for entailment of *channel equivalence* $\dot{\leftrightarrow}$, mirroring the rules [ρ-NAMEEQ$_1$] and [ρ-NAMEEQ$_2$] for concluding name equivalence:

$$[\textsc{ChaneQ}_1]\frac{}{\Psi \Vdash \ulcorner \mathbf{run}\ M \urcorner \dot{\leftrightarrow} M} \qquad [\textsc{ChaneQ}_2]\frac{\Psi \Vdash P_1 \equiv P_2}{\Psi \Vdash \ulcorner P_1 \urcorner \dot{\leftrightarrow} \ulcorner P_2 \urcorner}$$

including the symmetric and transitive closure of $\dot{\leftrightarrow}$. We then let the entailment relation for conditions of *structural congruence* $\equiv$ be defined such that $\equiv$ contains α-equivalence; that $(\mathscr{P}_{/\equiv}, \mid, \mathbf{0})$ is an abelian monoid; and containing the four congruence rules derived from the above translation:

$$[\textsc{Par}]\frac{\Psi \Vdash P_1 \equiv P_2}{\Psi \Vdash P_1 \mid R \equiv P_2 \mid R} \qquad [\textsc{In}]\frac{\Psi \Vdash M_1 \dot{\leftrightarrow} M_2 \qquad \Psi \Vdash P_1 \equiv P_2}{\Psi \Vdash \underline{M_1}(\lambda x_1)\langle\!\langle x_1 \rangle\!\rangle.P_1 \equiv \underline{M_2}(\lambda x_2)\langle\!\langle x_2 \rangle\!\rangle.P_2}$$

$$[\textsc{Run}]\frac{\Psi \Vdash M_1 \dot{\leftrightarrow} M_2}{\Psi \Vdash \mathbf{run}\ M_1 \equiv \mathbf{run}\ M_2} \qquad [\textsc{Out}]\frac{\Psi \Vdash M_1 \dot{\leftrightarrow} M_2 \qquad \Psi \Vdash P_1 \equiv P_2}{\Psi \Vdash \overline{M_1}\langle\!\langle \ulcorner P_1 \urcorner \rangle\!\rangle \equiv \overline{M_2}\langle\!\langle \ulcorner P_2 \urcorner \rangle\!\rangle}$$

This translation is sound and complete w.r.t. operational correspondence up to a reasonable notion of behavioural equivalence $\simeq$:

Theorem 3 (Operational correspondence)**.** *Let $\simeq$ be a notion of behavioural equivalence for processes of the* HOΨ*-instance of the ρ-calculus, that includes at least structural congruence and the axiom* $\mathbf{run}\ \ulcorner [\![P]\!] \urcorner \simeq [\![P]\!]$. *Then* $P \to P' \iff [\![P]\!] \to \simeq [\![P']\!]$.

Outline. The proof requires a number of steps. First we show that the translation preserves name equivalence; i.e. $x_1 \equiv_{\mathscr{N}} x_2 \iff 1 \Vdash [\![x_1]\!] \dot{\leftrightarrow} [\![x_2]\!]$ by induction in the rules of name equivalence and structural congruence. Then we show that substitution can be moved out of the translation; i.e. $[\![P\sigma]\!] \simeq [\![P]\!][\![\sigma]\!]$ and $1 \Vdash [\![(n)\sigma]\!] \dot{\leftrightarrow} ([\![n]\!])[\![\sigma]\!]$, where $\sigma \triangleq \{\ulcorner Q \urcorner / x\}$ and $[\![\sigma]\!] \triangleq [[\![x]\!] := [\![\ulcorner Q \urcorner]\!]]$, by induction in the clauses of the translation function. This step relies on our assumption about $\simeq$. Lastly we can show soundness and completeness w.r.t. operational correspondence by induction in the two semantics. In both cases, the interesting clauses are the communication rules, which require the aforementioned substitution and name equivalence preservation results. Details are available in [2]. □

5.3.2 A type system for reflection

Other higher-order calculi such as CHOCS and HOπ can be encoded in the π-calculus and may thus be typable through translation, but as we noted above there cannot be such an encoding of the ρ-calculus into the π-calculus. Thus, we cannot hope to create a type system for the ρ-calculus by adapting an existing first-order type system. In fact, we are not aware of any type system for the ρ-calculus, so we shall now create one by instantiating our generic type system. We let types for names be of the form

$$T \in \textbf{\textit{Types}} ::= \langle T, \Gamma \rangle \quad \mid \quad \langle B, \Gamma \rangle$$

where B is a basis type, and Γ is a type environment representing the possibility of executing the process within the name. Furthermore we shall use assertions as type environments for processes as we previously did with HOπ, so we update the definition accordingly.

$$\mathbb{A} \triangleq \wp(\{ \ulcorner P \urcorner : T \mid P \in \mathscr{P}_\Psi \wedge T \in \textbf{\textit{Types}} \} \cup \{ \langle\!| \ulcorner P \urcorner |\!\rangle : T \mid P \in \mathscr{P}_\Psi \wedge T \in \textbf{\textit{Types}} \})$$

with assertion unit and composition as $1 \triangleq \emptyset$ and $\otimes \triangleq \cup$ respectively. Note that by construction $\forall x \in \mathscr{N}.x\#\ulcorner P \urcorner$, so substitution can only occur in terms of the form $\langle\!| \ulcorner P \urcorner |\!\rangle : T$. We then append an assertion to the encoding of input and output:

$$\begin{aligned} [\![\ulcorner R \urcorner \langle P \rangle]\!] &\triangleq \overline{\ulcorner [\![R]\!] \urcorner} \langle\!| \ulcorner [\![P]\!] \urcorner |\!\rangle . \mathbf{0} \mid (\!| \{ \ulcorner [\![R]\!] \urcorner : T, \langle\!| \ulcorner [\![P]\!] \urcorner |\!\rangle : T' \} |\!) \\ [\![\ulcorner R \urcorner (x) . P]\!] &\triangleq \underline{\ulcorner [\![R]\!] \urcorner} (\lambda x) \langle\!| x |\!\rangle . [\![P]\!] \mid (\!| \{ \ulcorner [\![R]\!] \urcorner : T \} |\!) \end{aligned}$$

Lastly, we also need to take the type information into account when concluding channel equivalence, to ensure that two terms with initially dissimilar types cannot become channel equivalent after a substitution. Thus we redefine the entailment rule [CHANEQ$_2$] as follows:

$$[\textsc{ChaneQ}_2] \frac{\Gamma, \Psi \Vdash P_1 \equiv P_2 \qquad \Gamma, \Psi \vdash \ulcorner P_1 \urcorner : T \iff \Gamma, \Psi \vdash \ulcorner P_2 \urcorner : T}{\Gamma, \Psi \Vdash \ulcorner P_1 \urcorner \leftrightarrows \ulcorner P_2 \urcorner}$$

Now we can instantiate the generic type system by defining the instance parameters:

$$[\textsc{Term-1}] \frac{\ulcorner P \urcorner : \langle T, \Gamma' \rangle \in \Psi \qquad \Gamma', \Psi \vdash P}{\Gamma, \Psi \vdash \ulcorner P \urcorner : \langle T, \Gamma' \rangle} \quad [\textsc{Term-2}] \frac{\langle\!| \ulcorner P \urcorner |\!\rangle : \langle T, \Gamma' \rangle \in \Psi \qquad \Gamma', \Psi \vdash P}{\Gamma, \Psi \vdash \langle\!| \ulcorner P \urcorner |\!\rangle : \langle T, \Gamma' \rangle}$$

$$[\textsc{T-ass}] \frac{P : T \in \Psi' \implies T \curvearrowleft \Gamma}{\Gamma, \Psi \vdash (\!| \Psi' |\!)} \quad [\textsc{T-cha}]\ \langle T, \Gamma \rangle \leftrightarrows T \quad [\textsc{T-end}]\ \langle T, \Gamma \rangle \curvearrowleft \Gamma \quad [\textsc{Term-3}] \frac{\Gamma(x) = T}{\Gamma, \Psi \vdash x : T}$$

Note in particular the rules [TERM-1] and [TERM-2]: these rules say that the process within a term must be well-typed w.r.t. the type environment in the second component of its type, and that the process-type pair must be represented in the assertion.

Since we include [CHANEQ$_2$] in order to properly simulate the ρ-calculus, all names that eventually become equal during reduction must have the same type. This amounts to requiring that the programmer must know in advance all the names that will be generated by the program during execution. We have yet to find a type system for the ρ-calculus without this constraint.

6 Conclusions and future work

We have presented a generic type system for higher-order Ψ-calculi, which extends a previous type system for first-order Ψ-calculi. Like its predecessor, type judgements for processes are of the form $\Gamma \vdash P$ and are

given by a fixed set of rules. Terms, assertions and conditions are assumed to form nominal datatypes, and only a few requirements on type rules are imposed.

The generic type system allows us to identify what should be required of type systems for higher-order process calculi that are instances of the Ψ-calculus; these requirements take the form of instance assumptions. Thus it may also yield important insights into the general structure of type systems for higher-order calculi, and it may therefore also be taken as a starting point for developing more advanced type systems for any language that can be shown to be an instance of higher-order Ψ-calculi.

Our type system satisfies a general subject-reduction property and can be instantiated to yield type systems with a notion of channel safety for higher-order calculi such as CHOCS, HOπ and also the ρ-calculus. The latter in particular is interesting, as there is no valid encoding of the ρ-calculus into the π-calculus, and thus we cannot capture higher-order typability in a purely first-order setting. This establishes that our generic type system is richer than first-order type systems. However, typability in the ρ-calculus comes at the cost of necessitating that we include type information directly in the definition of channel equivalence. This amounts to saying that the programmer must know (and specify) in advance the type of all names that will be generated during the course of program evaluation. We do not know whether it is possible to create other (non-trivial) type systems for the ρ-calculus without such a restriction.

There are two important lines of future work in this direction: In [13], Hüttel extends the generic type system to consider more general notions of subtyping and resource awareness, and in [14] he also considers *session types* for psi-calculi. Both of these extensions are formulated for first-order Ψ-calculi only, and they would therefore be relevant to also consider in the higher-order setting.

References

[1] Martín Abadi & Andrew D. Gordon (1999): *A Calculus for Cryptographic Protocols: The Spi Calculus*. *Information and Computation* 148(1), pp. 1–70, doi:10.1006/inco.1998.2740.

[2] Alexander R. Bendixen, Bjarke B. Bojesen, Hans Hüttel & Stian Lybech (2022): *Typing Reflection in Higher-Order Psi-calculi*. Technical Report, Department of Computer Science, Aalborg University. Available at `http://icetcs.ru.is/stian/2022/hopsitypes2022techreport.pdf`.

[3] Jesper Bengtson, Magnus Johansson, Joachim Parrow & Björn Victor (2009): *Psi-calculi: Mobile processes, nominal data, and logic*. In: *2009 24th Annual IEEE Symposium on Logic In Computer Science*, IEEE, pp. 39–48, doi:10.1016/S1571-0661(05)80361-5.

[4] Jesper Bengtson, Magnus Johansson, Joachim Parrow & Björn Victor (2011): *Psi-calculi: a framework for mobile processes with nominal data and logic*. *Logical Methods in Computer Science* Volume 7, Issue 1, doi:10.2168/LMCS-7(1:11)2011. Available at `https://lmcs.episciences.org/696`.

[5] Luís Caires (2007): *Logical Semantics of Types for Concurrency*. In Till Mossakowski, Ugo Montanari & Magne Haveraaen, editors: *Algebra and Coalgebra in Computer Science*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 16–35, doi:10.1006/inco.1994.1093.

[6] Marco Carbone & Sergio Maffeis (2003): *On the Expressive Power of Polyadic Synchronisation in Pi-Calculus*. *Nordic Journal of Computing* 10(2), pp. 70–98, doi:10.1016/S1571-0661(05)80361-5.

[7] Romain Demangeon, Daniel Hirschkoff & Davide Sangiorgi (2010): *Termination in higher-order concurrent calculi*. *The Journal of Logic and Algebraic Programming* 79(7), pp. 550–577, doi:10.1016/j.jlap.2010.07.007. The 20th Nordic Workshop on Programming Theory (NWPT 2008).

[8] Murdoch Gabbay & Andrew Pitts (2002): *A New Approach to Abstract Syntax with Variable Binding*. *Formal Asp. Comput.* 13, pp. 341–363, doi:10.1007/s001650200016.

[9] Philippa Gardner & Lucian Wischik (2000): *Explicit fusions*. In: *International Symposium on Mathematical Foundations of Computer Science*, Springer, pp. 373–382, doi:10.1007/3-540-44612-5_33.

[10] Daniele Gorla (2010): *Towards a unified approach to encodability and separation results for process calculi*. *Information and Computation* 208(9), pp. 1031–1053, doi:10.1016/j.ic.2010.05.002.

[11] Matthew Hennessy & James Riely (2002): *Resource Access Control in Systems of Mobile Agents*. *Information and Computation* 173(1), pp. 82–120, doi:10.1006/inco.2001.3089.

[12] Hans Hüttel (2011): *Typed ψ-calculi*. In: *International Conference on Concurrency Theory*, Springer, pp. 265–279, doi:10.1007/978-3-642-23217-6_18.

[13] Hans Hüttel (2014): *Types for Resources in ψ-calculi*. In Martín Abadi & Alberto Lluch Lafuente, editors: *Trustworthy Global Computing*, Springer International Publishing, Cham, pp. 83–102, doi:10.1007/978-3-319-05119-2_6.

[14] Hans Hüttel (2016): *Binary Session Types for Psi-Calculi*. In Atsushi Igarashi, editor: *Programming Languages and Systems*, Springer International Publishing, Cham, pp. 96–115, doi:10.1007/978-3-319-47958-3_6.

[15] Atsushi Igarashi & Naoki Kobayashi (2004): *A generic type system for the Pi-calculus*. *Theoretical Computer Science* 311(1), pp. 121–163, doi:10.1016/S0304-3975(03)00325-6.

[16] Barbara König (2005): *Analysing input/output-capabilities of mobile processes with a generic type system*. *The Journal of Logic and Algebraic Programming* 63(1), pp. 35–58, doi:10.1016/j.jlap.2004.01.004. Special issue on The pi-calculus.

[17] Stian Lybech (2022): *Encodability and Separation for a Reflective and Higher-Order Language*. *Electronic Proceedings in Theoretical Computer Science* 368, Open Publishing Association, pp. 95–112, doi:10.4204/EPTCS.368.6.

[18] L.G. Meredith & Matthias Radestock (2005): *A Reflective Higher-order Calculus*. *Electronic Notes in Theoretical Computer Science* 141(5), pp. 49 – 67, doi:10.1016/j.entcs.2005.05.016. Proceedings of the Workshop on the Foundations of Interactive Computation (FInCo 2005).

[19] Robin Milner (1993): *The Polyadic π-Calculus: a Tutorial*. In: *Logic and Algebra of Specification*, Springer Berlin Heidelberg, pp. 203–246, doi:10.1007/978-3-642-58041-3_6.

[20] Joachim Parrow (2001): *An introduction to the π-calculus*. In: *Handbook of Process Algebra*, Elsevier, pp. 479–543, doi:10.1016/B978-044482830-9/50026-6.

[21] Joachim Parrow, Johannes Borgström, Palle Raabjerg & Johannes Åman Pohjola (2014): *Higher-order psi-calculi*. *Mathematical Structures in Computer Science* 24(2), doi:10.1017/S0960129513000170.

[22] Benjamin Pierce & Davide Sangiorgi (1993): *Typing and subtyping for mobile processes*. In: *[1993] Proceedings Eighth Annual IEEE Symposium on Logic in Computer Science*, IEEE, pp. 376–385, doi:10.1109/LICS.1993.287570.

[23] Davide Sangiorgi (1993): *Expressing mobility in process algebras: first-order and higher-order paradigms*. Ph.D. thesis, University of Edinburgh. Available at `http://hdl.handle.net/1842/6569`.

[24] Davide Sangiorgi (1993): *From π-calculus to higher-order π-calculus — and back*. In M. C. Gaudel & J. P. Jouannaud, editors: *TAPSOFT'93: Theory and Practice of Software Development*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 151–166, doi:10.1007/3-540-56610-4_62.

[25] Davide Sangiorgi & David Walker (2003): *The pi-calculus: a Theory of Mobile Processes*. Cambridge university press.

[26] Bent Thomsen (1989): *A Calculus of Higher Order Communicating Systems*. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL' 89*, POPL'89, ACM Press, New York, NY, USA, pp. 143–154, doi:10.1145/75277.75290.

[27] Bent Thomsen (1993): *Plain CHOCS A Second Generation Calculus for Higher Order Processes*. *Acta Inf.* 30(1), p. 1–59, doi:10.1007/BF01200262.

From CCS to CSP: the m-among-n Synchronisation Approach

Gerard Ekembe Ngondi, Vasileios Koutavas, Andrew Butterfield
Trinity College Dublin, Lero - the SFI Software Research Centre
gerard.ekembe, vkoutav, andrew.butterfield @tcd.ie

We present an alternative translation from CCS to an extension of CSP based on m-among-n synchronisation (called CSPmn). This translation is correct up to strong bisimulation. Unlike the g-star renaming approach ([4]), this translation is not limited by replication (viz., recursion with no nested parallel composition). We show that m-among-n synchronisation can be implemented in CSP based on multiway synchronisation and renaming.

1 Introduction

In [4], the authors present a translation from CCS [1] into CSP [22, 20], *ccs2csp*, which is correct up to strong bisimulation (cf. [10]). This means that a CCS process is strong bisimilar to its CSP translation. *ccs2csp* has been implemented in Haskell (cf. [23]), which allows using the model-checker FDR [7] for analysing translated CCS terms. In the course of the same work, the authors have proposed an alternative translation, $ccs2csp_2$, correct up to failure equivalence. Both translations differ in the translation of the prefix term $\tau.P$, translated into $(tau \rightarrow ccs2csp(P))\backslash_{csp}\{tau\}$ in the first case, and $ccs2csp_2(P)$ in the second case.

In this paper we present yet a third alternative, $ccs2csp_3$, achieved by first extending CSP with m-among-n synchronisation [9], from which we can derive multiway (or n-among-n) synchronisation, the default CSP synchronisation mechanism, and binary syncronisation (used in CCS). Then, we translate CCS parallel composition into the binary version of CSP parallel operator. The resulting translation is correct up to strong bisimulation.

The translations in [4] were achieved by hard coding binary synchronisation into CCS before going to CSP. Using a renaming function, g^*, the translations generated unique pairs of indices between any two pairs of complementary prefixes in a parallel composition, e.g., $(a,\bar{a}) \mapsto \{(a_{12},\bar{a}_{12}),(a_{13},\bar{a}_{13})\}$. This effectively made synchronising prefix pairs unique. Although these indices were generated in CCS, the g^*-renaming approach shows how to enforce binary synchronisation even in CSP: given a CSP process $P \underset{a}{\parallel} Q \underset{a}{\parallel} R$, to ensure binary synchronisations on a, assign unique indices to a accordingly, through renaming. E.g., $P[\{a_{12},a_{13}\}/a] \parallel Q[a_{12}/a] \parallel R[a_{13}/a]$ ensures that pairs of processes (P,Q) and (P,R) can synchronise respectively, but not (Q,R). This approach, which we call the Gstar approach, has been encoded in the translation tool and the resulting CSP terms can be analysed in FDR immediately.

m-among-n synchronisation [9] demands adding new rules to CSP, hence it would require updating FDR first. In other words, the CSP terms resulting from our new translation, $ccs2csp_3$, cannot immediately be analysed in FDR. Nonetheless, function g^* implements binary synchronisation, hence, can be taken for an implementation of 2-among-n synchronisation.

The Gstar approach does *not* allow translating recursive terms with nested parallelism (or replication). That is because function g^* needs to generate every synchronisation index so the translation can

V. Castiglioni and C. A. Mezzina (Eds): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics 2022 (EXPRESS/SOS 2022).
EPTCS 368, 2022, pp. 60–74, doi:10.4204/EPTCS.368.4

terminate. With m-among-n synchronisation, we need only one index to separate interleaving from synchronisation, i.e., we map every CCS name unto two CSP events, e.g., $a \mapsto \{a, a_S\}$, where a_S is the synchronisation event. Therefore, this new translation is not limited by parallel under recursion.

Our main contribution in this paper hence is a new translation from CCS into CSP which is correct up to strong bisimulation, is not limited by parallel under recursion, but cannot be immediately analysed with FDR. As a byproduct, we define m-among-n synchronisation for CSP processes. We call the corresponding extension CSPmn. We show that CSPmn preserves CSP axioms by defining m-among-n sysnchronisation in terms of both multiway synchronisation and renaming. The translation from CSPmn into CSP is limited by parallel under recursion as it requires generating unique indices for all possible combinations of synchronising processes.

2 Correct Translation, CCS(Tau), CSP, CCS-to-CSP

2.1 Correct Translations

A correct translation of one language into another is a mapping from the valid expressions in the first language to those in the second, that preserves their meaning (for some definition of meaning). Below we recall the two main definitions of correctness from [10].

Let $\mathscr{L} = (\mathbb{T}_{\mathscr{L}}, [\![\,]\!]_{\mathscr{L}})$ denote a language as a pair of a set $\mathbb{T}_{\mathscr{L}}$ of valid expressions in $\mathscr{L}$ and a surjective mapping $[\![\,]\!]_{\mathscr{L}} : \mathbb{T}_{\mathscr{L}} \to \mathscr{D}_{\mathscr{L}}$ from $\mathbb{T}_{\mathscr{L}}$ to some set of meanings $\mathscr{D}_{\mathscr{L}}$. Candidate instances of $[\![\,]\!]_{\mathscr{L}}$ are *traces* and *failures* (cf. [14, 21]).

Definition 1 (Correct Translation up to Semantic Equivalence [10])**.** *A translation $\mathsf{T} : \mathbb{T}_{\mathscr{L}} \to \mathbb{T}_{\mathscr{L}'}$ is correct up to a semantic equivalence $\approx$ on $\mathscr{D}_{\mathscr{L}} \cup \mathscr{D}_{\mathscr{L}'}$ when $[\![E]\!]_{\mathscr{L}} \approx [\![\mathsf{T}(E)]\!]_{\mathscr{L}'}$ for all $E \in \mathbb{T}_{\mathscr{L}}$.*

Operational correspondence allows matching the transitions of two processes, which can help determine the appropriate relation (semantic equivalence) between a term and its translation. Let the operational semantics of $\mathscr{L}$ be defined by the labelled transition system $(\mathbb{T}_{\mathscr{L}}, Act_{\mathscr{L}}, \to_{\mathscr{L}})$, where $Act_{\mathscr{L}}$ is the set of labels and $E \xrightarrow{\lambda}_{\mathscr{L}} E'$ defines transitions with $E, E' \in \mathbb{T}_{\mathscr{L}}$ and $\lambda \in Act_{\mathscr{L}}$.

Definition 2 (Labelled Operational Correspondence, [8, 19])**.** *Let $\mathsf{T} : \mathbb{T}_{\mathscr{L}} \to \mathbb{T}_{\mathscr{L}'}$ be a mapping from the expressions of a language $\mathscr{L}$ to those of a language $\mathscr{L}'$, and let $\mathsf{f} : Act_{\mathscr{L}} \to Act_{\mathscr{L}'}$ be a mapping from the labels of $\mathscr{L}$ to those of $\mathscr{L}'$. A translation $\langle \mathsf{T}, \mathsf{f} \rangle$ is operationally corresponding w.r.t. a semantic equivalence $\approx$ on $\mathscr{D}_{\mathscr{L}} \cup \mathscr{D}_{\mathscr{L}'}$ if it is:*

- *Sound: $\forall E, E' : E \xrightarrow{\lambda}_{\mathscr{L}} E'$ imply that $\exists F : \mathsf{T}(E) \xrightarrow{\mathsf{f}(\lambda)}_{\mathscr{L}'} F$ and $F \approx \mathsf{T}(E')$*
- *Complete: $\forall E, F : \mathsf{T}(E) \xrightarrow{\lambda'}_{\mathscr{L}'} F$ imply that $\exists E' : E \xrightarrow{\lambda}_{\mathscr{L}} E'$ and $F \approx \mathsf{T}(E') \wedge \lambda' = \mathsf{f}(\lambda)$*

The previous two definitions coincide when the semantic equivalence $\approx$ is strong bisimulation (Def.3) and f is the identity.

2.2 CCS, CCSTau

CCS. CCS (Calculus of Communicating Systems) [17, 1] is a process algebra that allows reasoning about concurrent systems. CCS represents programs as *processes*, whose behaviour is determined by

Table 1: SOS rules for CCS

$$Prefix : \alpha.P \xrightarrow{\alpha} P \qquad Sum : \frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'} \qquad Par : \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q}$$

$$Com : \frac{P \xrightarrow{\overline{a}} P' \quad Q \xrightarrow{a} Q'}{P|Q \xrightarrow{\tau} P'|Q'} \qquad Res : \frac{P \xrightarrow{\alpha} P' \quad \alpha \notin B}{P \upharpoonright B \xrightarrow{\alpha} P' \upharpoonright B} \qquad Rec : \frac{P[\mu X.P/X] \xrightarrow{\alpha} P'}{\mu X.P \xrightarrow{\alpha} P'}$$

rules specifying their possible execution steps. The syntax of CCS processes is defined by the following BNF:

$$CCS ::= 0 \,|\, \alpha.P \,|\, P+Q \,|\, P|Q \,|\, P \upharpoonright B \,|\, \mu X.P$$
$$\alpha ::= \tau \,|\, \overline{a} \,|\, a$$

Let $\mathscr{N}$ denote an infinite set of *names*; let $a,b,c,...$ range over $\mathscr{N}$. Let $\overline{\mathscr{N}} = \{\bar{a} | a \in \mathscr{N}\}$ denote the set of conames. Let $\overline{\overline{a}} = a$. Let $\mathscr{L} = \mathscr{N} \cup \overline{\mathscr{N}}$ denote the set of all possible labels. The set of labels of a process P is denoted by $\mathscr{L}(P)$ ([17, Def.2, p52]). Let τ denote the silent or invisible action. Let $Act = \mathscr{N} \cup \overline{\mathscr{N}} \cup \{\tau\}$ denote the set of all possible actions that a process can perform. Let $\alpha, \beta, ..$ range over Act. The SOS semantics of CCS are given in Table 1.

Informally: 0 (or *NIL*) is the process that performs no action. $\alpha.P$ is the process that performs an action α and then behaves like P. $P+Q$ is the process that behaves either like P or like Q. $P|Q$ is the process that executes P and Q in parallel: if both P and Q can engage in an action a then, their execution corresponds to interleaving, e.g. $a.0|a.0 \equiv a.a.0$; if P can engage in action a, Q in the complementary action $\bar{a}$, then, either P and Q interleave on a or they synchronise and the result of synchronisation is the invisible action τ, e.g. $a.0|\bar{a}.0 \equiv a.\bar{a}.0 + \bar{a}.a.0 + \tau.0$. $P \upharpoonright B$ is the process that cannot engage in actions in B except for synchronisation, e.g., $(a.0|\bar{a}.0) \upharpoonright \{a\} \equiv \tau.0$, $(a.0) \upharpoonright \{a\} \equiv 0$. $\mu X.P$ is the process that executes P recursively.

Equivalence based on bisimulations is the preferred choice for discriminating among CCS processes. We will use strong bisimulation to prove the correctness of our translation.

Definition 3 (Strong Bisimulation [21, 17])**.** *A strong bisimulation is a symmetric binary relation $\mathscr{R}$ on processes satisfying the following: $P\mathscr{R}Q$ and $P \xrightarrow{\alpha} P'$ imply that*

$$\exists Q' : Q \xrightarrow{\alpha} Q' \wedge P'\mathscr{R}Q'$$

P is strong bisimilar to Q, written $P \sim Q$, if $P\mathscr{R}Q$ for some strong bisimulation $\mathscr{R}$.

CCSTau. CCSTau [4] extends CCS with visible synchronisations, viz., the result of synchronisation on a pair $(a,\bar{a})$ is the visible action $\tau[a,\bar{a}]$ instead of the visible action τ. This makes it easier to guarantee that when two processes synchronise in CCS(Tau), their CSP translation also synchronises. The syntax

of CCSTau processes is defined by the following grammar:

$$\begin{aligned} P,Q,R &::= 0 \mid \alpha.P \mid P+Q \mid P|_\tau Q \mid P \upharpoonright B \mid \mu X.P \mid P\backslash_\tau B \mid X \\ \alpha &::= \tau \mid \overline{a} \mid a \\ \beta &::= \alpha \mid \tau[a|\overline{a}] \end{aligned}$$

The parallel operator in CCSTau is denoted $|_\tau$. CCSTau also defines a hiding operator, denoted $\backslash_\tau$, which can hide all actions including $\tau[a,\bar{a}]$ actions. The restriction operator behaves as in CCS, does not apply to $\tau[a,\bar{a}]$ actions. Rules for these operators are given hereafter:

$$Par: \frac{P \xrightarrow{\beta} P'}{P|_\tau Q \xrightarrow{\beta} P'|_\tau Q} \qquad Com: \frac{P \xrightarrow{\overline{a}} P' \quad Q \xrightarrow{a} Q'}{P|_\tau Q \xrightarrow{\tau[\overline{a}|a]} P'|_\tau Q'}$$

$$Res: \frac{P \xrightarrow{\beta} P' \quad \beta = \tau[\overline{a}|a] \text{ or } \beta \notin B}{P \upharpoonright B \xrightarrow{\beta} P' \upharpoonright B}$$

$$Hide: \frac{P \xrightarrow{\beta} P' \quad \beta \notin B}{P\backslash_\tau B \xrightarrow{\beta} P'\backslash_\tau B} \qquad \frac{P \xrightarrow{\beta} P' \quad \beta \in B}{P\backslash_\tau B \xrightarrow{\tau} P'\backslash_\tau B}$$

All other CCS operators are also CCSTau operators.

CCS-to-CCSTau. Translation function $c2ccs\tau$ [4] translates CCS processes into CCSTau, is correct up to strong bisimulation. For any CCS process P other than CCS-parallel operator, $c2ccs\tau(P) = P$. For the parallel operator: [1]

$$c2ccs\tau(P|Q) \mathrel{\widehat{=}} (c2ccs\tau(P)|_\tau c2ccs\tau(Q))\backslash_\tau \{\tau[a|\overline{a}] \mid a \in \mathscr{L}(P), \bar{a} \in \mathscr{L}(Q)\} \qquad \text{(c2ccs}\tau\text{-par-def)}$$

2.3 CSP

CSP (Communicating Sequential Processes) [14, 22] is a process algebra that allows reasoning about concurrent systems. In CSP, a (concurrent) program is represented as a *process*, whose behaviour is entirely determined by the possible actions of the program, represented as *events*. The set of events that a process P can possibly perform is denoted by $\mathscr{A}(P)$. Event τ denotes invisible actions, hidden from the environment; event $\checkmark$ denotes successful termination, by opposition say to deadlock and abortion. Both denotational and operational semantics have been defined for CSP processes, in terms of traces. The syntax of some CSP processes is defined by the following BNF:

$$\begin{aligned} CSP &::= SKIP \mid STOP \mid \alpha \rightsquigarrow P \mid P \sqcap Q \mid P \,\Box\, Q \mid P \underset{B}{\parallel} Q \mid f(P) \mid P \setminus B \mid \mu X.P \\ \alpha &::= a \mid a?x \mid a!m \end{aligned}$$

The SOS semantics of CSP processes are given in Table 2. Informally: *SKIP* is the process that refuses to engage in any event, terminates immediately, and does not diverge. *STOP* is the process that is unable to interact with its environment. $\alpha \rightsquigarrow P$ is the process that first engages in event α then behaves like P. $P \,\Box\, Q$ is the process that behaves like P or Q, where the choice is decided by the environment.

[1] The set of labels of a CCS process P, $\mathscr{L}(P)$, corresponds to the set of events $\mathscr{A}(Q)$ for a CSP process Q.

Table 2: SOS rules for CSP [22]

$$Prefix : (a \rightsquigarrow P) \xrightarrow{a} P \qquad Skip : SKIP \xrightarrow{\checkmark} STOP$$

$$IntChoice : P_1 \sqcap P_2 \xrightarrow{\tau} P_1 \qquad P_1 \sqcap P_2 \xrightarrow{\tau} P_2$$

$$ExtChoice : \frac{P_1 \xrightarrow{a} P'}{P_1 \,\Box\, P_2 \xrightarrow{a} P'} \qquad \frac{P_1 \xrightarrow{\tau} P'}{P_1 \,\Box\, P_2 \xrightarrow{\tau} P' \,\Box\, P_2}$$

$$IfacePar : \frac{P_1 \xrightarrow{a} P' \quad [a \notin B^{\checkmark}]}{P_1 \underset{B}{\parallel} P_2 \xrightarrow{a} P' \underset{B}{\parallel} P_2} \qquad \frac{P_1 \xrightarrow{a} P_1' \quad P_2 \xrightarrow{a} P_2' \quad [a \in B^{\checkmark}]}{P_1 \underset{B}{\parallel} P_2 \xrightarrow{a} P_1' \underset{B}{\parallel} P_2'}$$

$$Hide : \frac{P \xrightarrow{a} P' \quad [a \notin B]}{P \setminus B \xrightarrow{a} P' \setminus B} \qquad \frac{P \xrightarrow{a} P' \quad [a \in B]}{P \setminus B \xrightarrow{\tau} P' \setminus B}$$

$$FwdRen : \frac{P \xrightarrow{a} P'}{f(P) \xrightarrow{f(a)} f(P')} \qquad \frac{P \xrightarrow{\tau} P'}{f(P) \xrightarrow{\tau} f(P')}$$

$$Rec : \frac{P \xrightarrow{\mu} P' \quad [N = P]}{N \xrightarrow{\mu} P'}$$

$P \underset{B}{\parallel} Q$ behaves like the parallel execution of P and Q where the latter must both synchronise on the set of events B. When $B = \{\}$, we say that P and Q interleave, denoted by $P \,|||\, Q$; if $B = \mathscr{A}(P) \cap \mathscr{A}(Q)$ we also write $P \parallel Q$. $f(P)$ engages in $f(a)$ whenever P engages in a. $P \setminus B$ is the process that engages in all events of P except those in B. $\mu X.P$ is the process that executes P recursively.

Equivalence based on (enriched versions of) traces is the preferred choice for distinguishing CSP processes. We kindly refer the reader to [14, 22] for details.

2.4 CCS-to-CSP Translation

Notation. Given two functions, say f_1 and f_2, $f_1 \circ f_2$ denotes functional composition, viz., $f_1(f_2)$.

In this section, we present *ccs2csp* [4], the translation from CCS-to-CSP, correct up to strong bisimulation.

Definition 4 (*ccs2csp* [4])**.** *Let P be a CCS process. Then:*

$$\begin{aligned}
ccs2csp(P) &\mathrel{\widehat{=}} ai2a \circ (t2csp \circ c2ccs\tau(P))\backslash_{csp}\{a_{ij} | a_{ij} \in \mathscr{A}\big(t2csp(c2ccs\tau(P))\big)\} \\
t2csp(P) &\mathrel{\widehat{=}} (tl \circ conm \circ g^*_{\{\}} \circ ix(P))\backslash_{csp}\{tau\} \\
g^*_S &\mathrel{\widehat{=}} \{\tau \mapsto \tau, a_i \mapsto \{a_i\} \cup \{a_{ij} | \bar{a}_j \in S, i < j\} \cup \{a_{ji} | \bar{a}_j \in S, j < i\}\} \\
conm &\mathrel{\widehat{=}} \{\tau \mapsto \tau, a_i \mapsto a_i, \bar{a}_i \mapsto \bar{a}_i, a_{ij} \mapsto a_{ij}, \bar{a}_{ij} \mapsto a_{ij}\} \\
ai2a &\mathrel{\widehat{=}} \{a_i \mapsto a\}
\end{aligned}$$

where *ix* generates unique indexed prefixes such that a name b maps to a set of indexed names $b_i, i \geq 1$; g^* generates unique double-indexed names for every pair of synchronising names; *conm* renames every

synchronising coname into the corresponding name (so they can synchronise in CSP); and tl translates CCS operators into corresponding CSP operators. We kindly refer the reader to [4] for details.

Example 1 ([4]). *The translation of CCS binary synchronisation into CSP can be illustrated succinctly as follows:*

$$
\begin{aligned}
& ccs2csp(a.0|\bar{a}.0) && \text{(ccs2csp-def)}\\
=& ai2a \circ t2csp\big(c2ccs\tau(a.0|\bar{a}.0)\big)\backslash_{csp}\{a_{ij}|..\} && \text{(c2ccs}\tau\text{-par-def)}\\
=& ai2a \circ t2csp\big((a.0|_T \bar{a}.0)\backslash_T\{\tau[a|\bar{a}]\}\big)\backslash_{csp}\{a_{ij}|..\} && \text{(t2csp-def)}\\
=& ai2a \circ tl \circ conm \circ g^*(\{\}, ix\big((a.0|_T \bar{a}.0)\backslash_T\{\tau[a|\bar{a}]\}\big))\backslash_{csp}\{tau\}\backslash_{csp}\{a_{ij}|..\} && \text{(ix-def)}\\
=& ai2a \circ tl \circ conm \circ g^*\big((a_1.0|_T \bar{a}_2.0)\big)\backslash_{csp}\{tau\}\backslash_{csp}\{a_{ij}|..\} && \text{(gstar-def)}\\
=& ai2a \circ tl \circ conm\big((a_1.0+a_{12}.0)|_T(\bar{a}_2.0+\bar{a}_{12}.0)\big)\backslash_{csp}\{tau\}\backslash_{csp}\{a_{12}\} && \text{(conm-def)}\\
=& ai2a \circ tl\big((a_1.0+a_{12}.0)|_T(\bar{a}_2.0+a_{12}.0)\big)\backslash_{csp}\{tau,a_{12}\} && \text{(tl-def)}\\
=& ai2a \circ \big((a_1 \,\Box\, a_{12} \rightsquigarrow STOP) \underset{\{a_{12}\}}{\|} (\bar{a}_2 \,\Box\, a_{12} \rightsquigarrow STOP)\big)\backslash_{csp}\{tau,a_{12}\} && \text{(ai2a-def)}\\
=& \big((a \,\Box\, a_{12} \rightsquigarrow STOP) \underset{\{a_{12}\}}{\|} (\bar{a} \,\Box\, a_{12} \rightsquigarrow STOP)\big)\backslash_{csp}\{tau,a_{12}\}
\end{aligned}
$$

In CCS, a name can be used both for interleaving and for synchronisation. This is reflected in the translation above by generating indexed names a_1 and $\bar{a}_2$ for interleaving; then for the synchronisation pair $(a_1,\bar{a}_2)$, a unique synchronisation name a_{12} is generated. More generally, there will be as many a_{ij} synchronisation names as there are of synchronisation on name a.

In the next section, we extend CSP with m-among-n synchronisation, then derive 2-among-n (binary) synchronisation. In the end, we will be able to translate CCS binary synchronisation into CSP binary synchronisation.

3 CSP plus m-among-n Synchronisation

Multiway synchronisation in CSP is *maximal*, viz., all processes that can synchronise *must* synchronise. This is also called the *maximal (or n-ary) coordination* paradigm ([9]): if n processes are ready to synchronise on event a, then all n processes must synchronise together. Can we generalise this to allow only m-among-n ($2 \leq m \leq n$) processes to synchronise instead? If the answer is yes then binary synchronisation can be defined as 2-among-n coordination and n-ary synchronisation as n-among-n coordination. Garavel and Sighireanu [9] define m/n coordination for the language E-LOTOS.

First, let us generalise CSP (n-ary) interface parallel operator ([22]).

$$
IndxIfacePar: \frac{P_j \xrightarrow{a} P' \quad [a \notin B^{\checkmark}, k \neq j]}{\Big\|_B P_i \xrightarrow{a} (\Big\|_B P_k) \|_B P'} \qquad \frac{P_1 \xrightarrow{a} P_1' \quad \ldots \quad P_n \xrightarrow{a} P_n' \quad [a \in B^{\checkmark}]}{\Big\|_B P_i \xrightarrow{a} \Big\|_B P_i'}
$$

Definition 5 ($a\#m$ clause [9]). *Let $I = \{1,..,n\}, n \in \mathbf{N}, n \geq 2$. Let m be a natural number in the range $2,..,n$ associated to an a-event such that a clause $a\#m$ denotes that m processes are allowed to synchronise on event a at once. Each clause $\#m$ is optional: if omitted, m has default value n.*

The rules for m/n indexed interface paralell composition are given hereafter.[2]

$$M/N\text{–}IndxIfacePar : \frac{P_j \xrightarrow{a} P' \quad [a\#m \notin B^{\checkmark} \times \{2,..,n\}, k \neq j]}{\Big\|_{B\times\{2,..,n\}} P_i \xrightarrow{a} (\Big\|_{B\times\{2,..,n\}} P_k) \Big\|_{B\times\{2,..,n\}} P'}$$

$$\frac{P_1 \xrightarrow{a} P'_1 \ldots P_n \xrightarrow{a} P'_n \quad [a\#m \in B^{\checkmark} \times \{2,..,n\}, j \in J, k \neq j]}{\Big\|_{B\times\{2,..,n\}} P_i \xrightarrow{a} \bigsqcap_{\{J\subseteq I | card(J)=m\}} \left((\Big\|_{B\times\{2,..,n\}} P_k) \Big\|_{B\times\{2,..,n\}} (\Big\|_{B\times\{2,..,n\}} P'_j) \right)}$$

We can then derive binary-only synchronisation by imposing that *every* event in set B allows 2(only)-among-n processes to synchronise.

$$2/N\text{–}IndxIfacePar : \frac{P_1 \xrightarrow{a} P'_1 \ldots P_n \xrightarrow{a} P'_n \quad [a\#2 \in A^{\checkmark} \times \{2\}, j \in J, k \neq j]}{\Big\|_{B\times\{2\}} P_i \xrightarrow{a} \bigsqcap_{\{J\subseteq I | card(J)=2\}} \left((\Big\|_{B\times\{2\}} P_k) \Big\|_{B\times\{2\}} (\Big\|_{B\times\{2\}} P'_j) \right)}$$

Similarly, we derive n-ary-only synchronisation by imposing that *every* event in set B allows n-among-n processes to synchronise. We easily verify that rules N/N-IndxIfacePar and IndxIfacePar (synchronisation) are the same.

$$N/N\text{–}IndxIfacePar : \frac{P_1 \xrightarrow{a} P'_1 \ldots P_n \xrightarrow{a} P'_n \quad [a\#n \in B^{\checkmark} \times \{n\}]}{\Big\|_{B\times\{n\}} P_i \xrightarrow{a} \Big\|_{B\times\{n\}} P'_i}$$

Correctness of M/N-IndxIfacePar rule. Let us call CSPmn the extension of CSP with m-among-n synchronisation. We argue here that CSPmn is a conservative extension of CSP, i.e., CSPmn preserves the axioms of CSP.

The proof method is suggested to us by function g^* [4]. For binary synchronisation, select process pairs that must synchronise and assign them a unique synchronisation index. E.g.,

$$a \Big\|_{a\#2} a \Big\|_{a\#2} a \quad \text{maps to} \quad (a_{12} \Box a_{13}) \parallel (a_{12} \Box a_{23}) \parallel (a_{13} \Box a_{23})$$

Then, for m processes to synchronise among n, generate a unique index for all possible combinations of m processes among n, e.g.,

$$a \Big\|_{a\#2} a \Big\|_{a\#2} a \Big\|_{a\#2} a \quad \text{maps to} \quad (a_{12} \Box a_{13} \Box a_{14}) \parallel (a_{12} \Box a_{23} \Box a_{24}) \parallel (a_{13} \Box a_{23} \Box a_{34}) \parallel (a_{14} \Box a_{24} \Box a_{34})$$

$$a \Big\|_{a\#3} a \Big\|_{a\#3} a \Big\|_{a\#3} a \quad \text{maps to} \quad (a_{123} \Box a_{124} \Box a_{134}) \parallel (a_{123} \Box a_{124} \Box a_{234}) \parallel (a_{123} \Box a_{134} \Box a_{234}) \parallel (a_{124} \Box a_{134} \Box a_{234})$$

$$a \Big\|_{a\#4} a \Big\|_{a\#4} a \Big\|_{a\#4} a \quad \text{maps to} \quad a_{1234} \parallel a_{1234} \parallel a_{1234} \parallel a_{1234}$$

[2] The rules in [9] use a different rule format than CSP rules: they use predicates.

From what precedes, there exists a relational renaming, say G, such that

$$\Big\|_{a\#m,j} P_j \sim \Big\|_{G(a),j} P_j[G(a)/a]$$

We can thus define (CSPmn parallel operator) $\Big\|_{a\#m}$ in terms of both (CSP parallel operator) $\Big\|_{a}$ and (CSP relational renaming) $G(a)$. Therefore, CSPmn is a conservative extension of CSP, viz., preserves CSP axioms (cf. Appendix A for a full proof).

4 CCSTau Transformations

$$\boxed{CCS} \xrightarrow{c2ccs\tau} \boxed{CCSTau} \xrightarrow{g^2} \xrightarrow{conm} \boxed{CCSTau} \xrightarrow{tl_3} \xrightarrow{\backslash\{tau\}} \xrightarrow{\backslash\{a_S\}} \boxed{CSPmn}$$

Figure 1: CCS-to-CSPmn Translation workflow

The different stages of our translation are shown in Fig. 1.

Pairwise vs. Multiway Synchronisation Recall, a CCSTau name has both interleaving and synchronisation semantics. We hence have to generate two distinct CSP events for a single CCS name. Also, it is possible to hide $\tau[a|\bar{a}]$ synchronisation actions in CCSTau (typically, to obtain a CCS process—cf. Def.c2ccsτ-par-def). Then, it will be convenient to ignore them. Let g^2 define the function that generates a synchronisation name for any CCS name.

Definition 6 ($g^2(\alpha)$)**.**

$$g^2(S,\tau) \mathrel{\widehat{=}} \tau \qquad g^2(S,a) \mathrel{\widehat{=}} \{a\} \cup \{a_S \,|\, \bar{a} \in S\}$$
$$g^2(S,\tau[a|\bar{a}]) \mathrel{\widehat{=}} \{\tau[a,\bar{a}]\} \qquad g^2(S,B) \mathrel{\widehat{=}} \{g^2(S,a) \,|\, a \in B, \bar{a} \in S\}$$

Given a set of names generated by g^2, a-names denote interleaving, whilst a_S-names denote synchronisation. The application of g^2 to processes is given hereafter.

Definition 7 ($g^2(P)$)**.** *Let P be a CCS process. Let $g^2(P) \mathrel{\widehat{=}} g^2(\{\},P)$.*

$$g^2(S,0) \mathrel{\widehat{=}} 0 \qquad g^2(S,P \upharpoonright B) \mathrel{\widehat{=}} g^2(S,P) \upharpoonright g^2(S,B)$$
$$g^2(S,\alpha.P) \mathrel{\widehat{=}} \sum_{b \in g^2(S,\alpha)} b.g^2(S,P) \qquad g^2(S,P\backslash_\tau B) \mathrel{\widehat{=}} g^2(S,P)\backslash_\tau g^2(S \cup B,B)$$
$$g^2(S,P+Q) \mathrel{\widehat{=}} g^2(S,P) + g^2(S,Q) \qquad g^2(S,\mu X.P) \mathrel{\widehat{=}} \mu X.g^2(S,P)$$
$$g^2(S,P|_\tau Q) \mathrel{\widehat{=}} g^2(S \cup \mathscr{A}(Q),P)|_\tau g^2(S \cup \mathscr{A}(P),Q) \qquad g^2(S,X) \mathrel{\widehat{=}} X$$

Note the difference between restriction and hiding. Names $g^2(S,B)$ are generated between a process and its environment. Only those names will be restricted, understood that (restricted) B names cannot interact with their environment. Internal synchronisation on B names, however, will not be restricted (until later in CSP). In contrast, for hiding, internal synchronisation on B must be hidden as well, hence we hide names $g^2(S \cup B,B)$ instead.

Example 2. *Let us illustrate the translation of restriction.*

$$\begin{aligned}
& g^2(\{\},(a.0|_T\,\bar{a}.0)\upharpoonright\{a\}) && \text{(g2-def)}\\
&= g^2(\{\},a.0|_T\,\bar{a}.0)\upharpoonright g^2(\{\},\{a\}) && \text{(g2-res-def)}\\
&= \big(g^2(\{\bar{a}\},a.0)|_T\,g^2(\{a\},\bar{a}.0)\big)\upharpoonright\{a\} && \text{(g2-par-def)}\\
&= \big((a.0+a_S.0)|_T(\bar{a}.0+\bar{a}_S.0)\big)\upharpoonright\{a\}
\end{aligned}$$

Contrast with hiding, which hides both a and a_S. (Recall $\backslash_T\{a\}=\backslash_T\{a,\bar{a}\}$.)

$$\begin{aligned}
& g^2(\{\},(a.0|_T\,\bar{a}.0)\backslash_T\{a\}) && \text{(hide-def)}\\
&= g^2(\{\},(a.0|_T\,\bar{a}.0)\backslash_T\{a,\bar{a}\}) && \text{(g2-def)}\\
&= g^2(\{\},a.0|_T\,\bar{a}.0)\backslash_T g^2(\{a,\bar{a}\},\{a,\bar{a}\}) && \text{(g2-hide-def)}\\
&= \big(g^2(\{\bar{a}\},a.0)|_T\,g^2(\{a\},\bar{a}.0)\big)\backslash_T\{a,\bar{a},a_S,\bar{a}_S\} && \text{(g2-par-def, hide-def)}\\
&= \big((a.0+a_S.0)|_T(\bar{a}.0+\bar{a}_S.0)\big)\backslash_T\{a,a_S\}
\end{aligned}$$

Finally, consider hiding the synchronisation action $\tau[a|\bar{a}]$, this turns out to be vacuous.

$$\begin{aligned}
& g^2(\{\},(a.0|_T\,\bar{a}.0)\backslash_T\{\tau[a|\bar{a}]\}) && \text{(g2-def)}\\
&= g^2(\{\},a.0|_T\,\bar{a}.0)\backslash_T g^2(\{a\},\{\tau[a|\bar{a}]\}) && \text{(g2-hide-def)}\\
&= \big(g^2(\{\bar{a}\},a.0)|_T\,g^2(\{a\},\bar{a}.0)\big)\backslash_T\{\tau[a|\bar{a}]\} && \text{(g2-par-def)}\\
&= \big((a.0+a_S.0)|_T(\bar{a}.0+\bar{a}_S.0)\big)\backslash_T\{\tau[a|\bar{a}]\}
\end{aligned}$$

Parallel Composition. In CSP, synchronisation pairs $(a_S,\bar{a}_S)$ will not be able to synchronise. We hence update the coname function to translate conames into names.

Definition 8 (*conm*)**.** $conm \mathrel{\widehat{=}} \{\tau\mapsto\tau, a\mapsto a, \bar{a}\mapsto\bar{a}, a_S\mapsto a_S, \bar{a}_S\mapsto a_S\}$.

Link CCSTau-to-CSPmn In [4], function *tl* translates CCSTau operators into CSP operators, without consideration for differences in their respective alphabets. Hereafter, we define tl_3, to map CCS binary synchronisation into CSPmn binary synchronisation. All other operators are translated as before, viz., $tl_3(P)=tl(P)$ for all process expressions other than parallel composition. Additionally, because of the possibility to hide $\tau[a,\bar{a}]$ synchronisation actions in CCSTau, we translate CCSTau hiding operator also, translation which was not needed for *tl*.

Definition 9 (tl_3)**.** *Let tau be a CSP event that cannot synchronise.*

$$\begin{aligned}
tl_3(0) &\mathrel{\widehat{=}} STOP & tl_3(P+Q) &\mathrel{\widehat{=}} tl_3(P)\,\Box\,tl_3(Q)\\
tl_3(\tau.P) &\mathrel{\widehat{=}} tau\rightsquigarrow tl_3(P) & tl_3(P|_T Q) &\mathrel{\widehat{=}} tl_3(P) \underset{\{a\#2|a\in\mathscr{A}(P)\cap\mathscr{A}(Q)\}}{\|} tl_3(Q)\\
tl_3(a.P) &\mathrel{\widehat{=}} a\rightsquigarrow tl_3(P) & tl_3(\mu X.P) &\mathrel{\widehat{=}} \mu X.tl_3(P)\\
tl_3(P\upharpoonright B) &\mathrel{\widehat{=}} tl_3(P)\upharpoonright_{csp} B & tl_3(X) &\mathrel{\widehat{=}} X\\
tl_3(P\backslash_T B) &\mathrel{\widehat{=}} tl_3(P)\backslash_{csp} B
\end{aligned}$$

Note that $tl_3(P\backslash_T\{\tau[a|\bar{a}]\}) = tl_3(P)\backslash_{csp}\{\tau[a|\bar{a}]\} = tl_3(P)$, since $\tau[a|\bar{a}]$ actions do not occur in the translated term, $tl_3(P)$. This is necessary, as illustrated subsequently.

Example 3. *CCS process $a\,|\,\bar{a}\,|\,a$, by c2ccsτ, corresponds to CCSTau process*

$$\big((a|_T\bar{a})\backslash_T\{\tau[a|\bar{a}]\}|_T a\big)\backslash_T\{\tau[a,\bar{a}]\}$$

By g^2, this becomes process

$$\big(((a+a_S)|_T(\bar{a}+\bar{a}_S))\backslash_T\{\tau[a|\bar{a}]\}|_T(a+a_S)\big)\backslash_T\{\tau[a|\bar{a}]\}$$

Then, by tl_3, it becomes

$$\begin{aligned}&\big(((a\,\Box\,a_S)\underset{a_S\#2}{\|}(\bar{a}\,\Box\,\bar{a}_S))\backslash_{csp}\{\tau[a|\bar{a}]\}\underset{a_S\#2}{\|}(a\,\Box\,a_S)\big)\backslash_{csp}\{\tau[a|\bar{a}]\}\\ =\;&(a\,\Box\,a_S)\underset{a_S\#2}{\|}(\bar{a}\,\Box\,\bar{a}_S)\underset{a_S\#2}{\|}(a\,\Box\,a_S)\end{aligned}$$

Thanks to $\backslash_{csp}\{\tau[a,\bar{a}]\}$ being vacuous, there will be two possible synchronisations on a_S, corresponding to the original CCS behaviour.

The following abbreviation translates CCSTau into CSPmn.

Definition 10 (CCSTau to CSPmn)**.** *Let P be a CCSTau process. Then:*

$$t2csp_3(P)\;\widehat{=}\;(tl_3\circ conm\circ g^2(P))\backslash_{csp}\{tau\}$$

Link CCS-to-CSPmn. We obtain the translation from CCS to CSP by translating CCS into CCSTau first, using $c2ccs\tau$ (Def.c2ccsτ-par-def), then translating CCSTau into CSPmn, using $t2csp_3$ (Def.10), and finally hiding every a_S synchronisation event.

Definition 11 (CCS to CSPmn)**.** *Let P denote a CCS process. Then:*

$$ccs2csp_3(P)\;\widehat{=}\;(t2csp_3\circ c2ccs\tau(P))\backslash_{csp}\{a_S|a_S\in\mathscr{A}(t2csp_3\circ c2ccs\tau(P))\}$$

Example 4. *The translation of CCS binary synchronisation into CSPmn can be illustrated succinctly as follows:*

$$\begin{aligned}&ccs2csp_3(a.0|\bar{a}.0) && \text{(ccs2csp3-def.11)}\\ =\;&\big(t2csp_3\circ c2ccs\tau(a.0|\bar{a}.0)\big)\backslash_{csp}\{a_S|..\} && \text{(c2ccs}\tau\text{-par-def)}\\ =\;&t2csp_3\big((a.0|_T\bar{a}.0)\backslash_T\{\tau[a|\bar{a}]\}\big)\backslash_{csp}\{a_S\} && \text{(t2csp3-def.10)}\\ =\;&tl_3\circ conm\circ g^2(\{\},(a.0|_T\bar{a}.0)\backslash_T\{\tau[a|\bar{a}]\})\backslash_{csp}\{tau\}\backslash_{csp}\{a_S\} && \text{(g2-def.6)}\\ =\;&tl_3\circ conm\Big(\big((a.0+a_S.0)|_T(\bar{a}.0+\bar{a}_S.0)\big)\backslash_T\{\tau[a|\bar{a}]\}\Big)\backslash_{csp}\{tau\}\backslash_{csp}\{a_S\} && \text{(conm-def.8)}\\ =\;&tl_3\Big(\big((a.0+a_S.0)|_T(\bar{a}.0+a_S.0)\big)\backslash_T\{\tau[a|\bar{a}]\}\Big)\backslash_{csp}\{tau\}\backslash_{csp}\{a_S\} && \text{(tl3-def.9, CSP)}\\ =\;&\big((a\,\Box\,a_S\rightsquigarrow STOP)\underset{\{a_S\#2\}}{\|}(\bar{a}\,\Box\,a_S\rightsquigarrow STOP)\big)\backslash_{csp}\{tau,a_S\}\end{aligned}$$

Example 5. *The translation of recursion with nested parallel can be illustrated as follows. Let $P\,\widehat{=}\,\mu X.(a|\bar{a}.X)$ (or equiv. $P\,\widehat{=}\,a.0\,|\,\bar{a}.P$) be a CCS process. Then, $ix(P)=a_1\,|\,a_2.ix_{\{3..\}}(P)$, where $ix_{\{3..\}}$ denotes that indexing excludes indices 1 and 2. Let us unfold P one step, then:*

$$\begin{aligned}P&=a\,|\,\bar{a}.(a\,|\,\bar{a}.P)\\ ix(P)&=a_1\,|\,\bar{a}_2.(a_3\,|\,\bar{a}_4.ix_{\{5..\}}(P))\end{aligned}$$

The synchronisation pairs are thus $(a_1,\bar{a}_2),(a_1,\bar{a}_4),..$, *that is, the set* $\{(a_1,\bar{a}_{2k})|k\geq 1\}$. *Then:*

$$g^*(P) = (a_1 + \sum_{k\geq 1} a_{1*2k}) \,|\, (\bar{a}_2 + \bar{a}_{12}).g^*(P)$$

We will not be able to generate all the a_{1*2k} *indices since recursion is unbounded. For closure, we give the* temptative *translation of P with ccs2csp:* [3]

$$ccs2csp(P) = \big((a \,\Box\, \Box_{k\geq 1} a_{1*2k}) \parallel_{\{a_{1*2k}|k\geq 1\}} (\bar{a}_2 \,\Box\, a_{12}) \rightsquigarrow ai2a \circ t2csp \circ c2ccs\tau(P)\big)\backslash_{csp}\{a_{ij}|..\}$$

In contrast, let us define $ccs2csp_3(P)$. *Then:*

$$\begin{aligned} g^2(P) &= (a+a_S) \,|\, (\bar{a}+\bar{a}_S).g^2(P) \\ &= (a+a_S) \,|\, (\bar{a}+\bar{a}_S).\big((a+a_S) \,|\, (\bar{a}+\bar{a}_S).g^2(P)\big) \end{aligned}$$

We can unfold P multiple times, we only ever generate a single name for synchronisation. Then:

$$ccs2csp_3(P) = \big((a \,\Box\, a_S) \parallel_{a_S\#2} (\bar{a} \,\Box\, a_S) \rightsquigarrow t2csp_3 \circ c2ccs\tau(P)\big)\backslash_{csp}\{a_S\}$$

5 Gstar Implements 2/n-Synchronisation

We discuss here the relation between g^*-renaming ([4]) and m-among-n synchronisation (§3) approaches.

Recall, function g^* (Def.4, [4]) computes for a CCSTau process P all the substitute names corresponding to distinct synchronisation possibilities of P with its environment, plus interleaving. We have proposed an alternative solution based on extending CSP with 2-among-n synchronisation, derived from first extending CSP with m-among-n synchronisation. Whilst this second solution is more elegant than the gstar-renaming one, the problem of its *immediate* implementability in a tool like FDR has been raised.

Given the current version of FDR, m-among-n synchronisation cannot be implemented directly. We remark, however, that one effect of m-among-n synchronisation is to select, using non-deterministic choice, the m processes that are allowed to synchronise; effect which is precisely what function g^* achieves through renaming. We discuss how to relate both results.

Let us refer by CSPgstar the CSP process expressions resulting from translation *ccs2csp*. We can translate CSPgstar expressions into CSPmn expressions as follows.

Definition 12 (gstar2m/n)**.** *Let* a_{ij} *be an* g^* *name,* a_S *an* g^2 *name. Then:* $g^*2g^2 \mathrel{\widehat{=}} \{\tau \mapsto \tau, a_{ij} \mapsto a_S\}$

While g^*2g^2 is a simple renaming function, its application to CSP processes is modified specifically for the parallel operator such as to map $\parallel_{\{a_{ij}\}}$ unto $\parallel_{\{a_S\#2\}}$ (instead of $\parallel_{\{a_S\}}$).

Definition 13. *Let P be a CSP process.*

$$\begin{array}{ll} g^*2g^2(STOP) \mathrel{\widehat{=}} STOP & g^*2g^2(P \parallel_{\{a_{ij}\}} Q) \mathrel{\widehat{=}} g^*2g^2(P) \parallel_{\{a_S\#2\}} g^*2g^2(Q) \\ g^*2g^2(\alpha \rightsquigarrow P) \mathrel{\widehat{=}} g^*2g^2(\alpha) \rightsquigarrow g^*2g^2(P) & g^*2g^2(P\backslash_{csp}B) \mathrel{\widehat{=}} g^*2g^2(P)\backslash_{csp} g^*2g^2(B) \\ g^*2g^2(P \sqcap Q) \mathrel{\widehat{=}} g^*2g^2(P) \sqcap g^*2g^2(Q) & g^*2g^2(P \,\Box\, Q) \mathrel{\widehat{=}} g^*2g^2(P) \,\Box\, g^*2g^2(Q) \end{array}$$

[3] We are lucky that we can tell in advance what the synchronisation indices are, because process P is a simple case.

Theorem 1. *Let P be a CCS processes. Then:* $g^*2g^2 \circ ccs2csp(P) = ccs2csp_3(P)$.

Proof. By induction on the structure of CCS processes. When P does not mention CCS parallel, the proof is straightforward. We develop the proof for the parallel case only. We have:

$$
\begin{aligned}
& g^*2g^2 \circ ccs2csp(P \mid Q) && \text{(ccs2csp-def.4)}\\
=\ & g^*2g^2 \circ ai2a \circ (t2csp(P) \underset{\{a_{ij}\}}{\parallel} t2csp(Q))\backslash_{csp}\{tau, a_{ij}\} && \text{(CSP hide law)}\\
=\ & g^*2g^2 \circ (ai2a \circ t2csp(P)\backslash_{csp}\{tau\} \underset{\{a_{ij}\}}{\parallel} ai2a \circ t2csp(Q)\backslash_{csp}\{tau\})\backslash_{csp}\{a_{ij}\} && \text{(g*2g2-def.12)}\\
=\ & \big(g^*2g^2 \circ ai2a \circ t2csp(P)\backslash_{csp}\{tau\} \underset{\{a_S\#2\}}{\parallel} g^*2g^2 \circ ai2a \circ t2csp(P)\backslash_{csp}\{tau\}\big)\backslash_{csp}\{a_S\} \\
& && \text{(Induction Hyp., ccs2csp3-def.11)}\\
=\ & ccs2csp_3(P \mid Q)
\end{aligned}
$$

□

We say that g^* implements 2-among-n synchronisation.

6 Conclusion and Future Work

[4] proposes a translation of CCS into CSP based on the g^*-renaming approach whereby if two processes can synchronise on an action b, then a name unique to these two processes, say b_{ij}, is generated to substitute b. Thus, if more than two processes could initially synchronise on b, only two processes will ever be able to synchronise on b_{ij} after application of g^*.

In this paper, we propose an alternative, the m-among-n synchronisation approach, whereby we first extend CSP multiway synchronisation (or n-among-n) to m-among-n synchronisation (extension called CSPmn), from which we derive 2-among-n or binary synchronisation for CSP processes. We then translate CCS binary synchronisation into CSPmn binary synchronisation. Unlike the g^*-renaming approach, the m/n-approach is not limited by parallel under recursion since we can generate a single synchronisation name, say a_S, independently of the number of processes meant to synchronise on a_S.

We have also shown that CSPmn is a conservative extension of CSP (viz., preserves CSP axioms) by defining (CSPmn) m-among-n synchronisation in terms of both (CSP) multiway (or n-among-n) synchronisation and relational renaming.

We are tempted to affirm that m-among-n synchronisation is more expressive than both 2-among-n and n-among-n synchronisation. However, Hatzel et al. [11] propose an encoding from CSP into CCS whereby they encode CSP multiway synchronisation based on CCS binary synchronisation. Our work suggests that in trying to translate CSP into CCS, it would be easier to extend CCS with multiway synchronisation, as we have done here for CSP. Other works on the translation from CSP into CCS include [2], [3], [12], and [10].

We have proposed here the translation from CCS to CSP only. The main reason for this is our interest in using CSP tools such as FDR for reasoning about CCS processes. With regard to this concern, the g^*-renaming approach is more readily implementable than the m/n-approach. The latter would require extending FDR with semantics (viz. rules) for m-among-n synchronisation. Alternatively, m-among-n synchronisation can be implemented using function $g^*_{\#}$ (Def.15), however, with the limitation on parallel under recursion similar to g^* (cf. [4]). Mechanising our results in Isabelle theorem prover is also to be explored in the future.

Acknowledgments. This work was conducted with the financial support of the Science Foundation Ireland grant 13/RC/2094 and co-funded under the European Regional Development Fund through the Southern and Eastern Regional Operational Programme to Lero - the Irish Software Research Centre (`www.lero.ie`). For the purpose of Open Access, the author has applied a CC BY public copyright licence to any Author Accepted Manuscript version arising from this submission.

References

[1] L. Aceto, K.A. Larsen, A. Ingolfsdottir, *An Introduction to Milner's CCS*, 2005. [Last Accessed 19.Aug.2022: http://twiki.di.uniroma1.it/pub/MFS/WebHome/intro2ccs.pdf]

[2] E. Astesiano, E. Zucca, *Semantics of CSP via Translation into CCS*, Mathematical Foundations of Computer Science (MFCS), vol. 118, pp.172-182, 1981. doi:10.1007/3-540-10856-4_83

[3] S.D. Brookes, *On the Relationship of CCS and CSP*, Automata, Languages and Programming, LNCS, vol. 154, pp. 83-96, 1983. doi:10.1007/BFb0036899

[4] G. Ekembe Ngondi, V. Koutavas, A. Butterfield, *Translation of CCS into CSP, Correct up to Strong Bisimulation*, SEFM'21, pp. 243-261, LNCS vol. 13085, Dec. 2021. doi:10.1007/978-3-030-92124-8_14

[5] G. Ekembe Ngondi, *Denotational Semantics of Channel Mobility in UTP-CSP*, Formal Aspects of Computing Journal, May 2021. doi:10.1007/s00165-021-00546-3

[6] G. Ekembe Ngondi, *Denotational Semantics of Mobility in UTP (Unifying Theories of Programming)*, PhD Thesis, University of York, 2016.

[7] *FDR Documentation*, [Last Accessed 19.Aug.2022] `https://cocotec.io/fdr/manual/`

[8] Y. Fu, H. Lu, *On the Expressiveness of Interaction*, TCS, vol. 411, pp. 1387-1451, 2010. doi:10.1016/j.tcs.2009.11.011

[9] H. Garavel, M. Sighireanu, *A Graphical Parallel Composition Operator for Process Algebras*, IFIPAICT, vol. 28, pp. 185-202, 1999. doi:10.1007/978-0-387-35578-8_11

[10] R. van Glabbeek, *Musings on Encodings and Expressiveness*, EPTCS vol. 89, pp. 81–98, 2012. doi:10.4204/EPTCS.89.7

[11] M. Hatzel, C. Wagner, K. Peters, Uwe Nestmann, *Encoding CSP into CCS*, EXPRESS/SOS Workshop, EPTCS vol. 190, pp. 61-75, 2015. doi:10.4204/EPTCS.190.5

[12] M. Hennessy, Wei Li, G.D. Plotkin, *A First Attempt at Translating CSP into CCS*, International Conference on Distributed Computing (ICDC), pp105–115, 1981.

[13] J. He, C.A.R. Hoare, *CSP is a retract of CCS*, TCS, vol. 411, pp. 1311-1337, Elsevier, 2010. doi:10.1016/j.tcs.2009.12.012

[14] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.

[15] T. Hoare, Jifeng He, *Unifying Theories of Programming*, Prentice-Hall, 1998.

[16] R. Milner, *Communicating and Mobile Systems: the Pi-calculus*, Cambridge University Press, 1999.

[17] R. Milner, *Communication and Concurrency*, Prentice-Hall, 1989.

[18] R. De Nicola, M. Hennessy, *CCS without tau's*, TAPSOFT'87, LNCS, vol. 249, pp. 138-152, 1987. doi:10.1007/3-540-17660-8_53

[19] K. Peters, *Comparing Process Calculi Using Encodings*, EXPRESS/SOS Workshop, EPTCS, vol. 300, pp. 19–38, 2019. doi:10.4204/EPTCS.300.2

[20] A.W. Roscoe, *The Theory and Practice of Concurrency*, Prentice-Hall, 1998.

[21] D. Sangiorgi, *Introduction to Bisimulation and Coinduction*, Cambridge University Press, 2012.

[22] S. Schneider, *Concurrent and Real-Time Systems - The CSP Approach*, John Wiley& Sons, Ltd, 2000.

[23] *Haskell Prototype Automation of CCS-to-CSP translation*, GitHub Repository, [Last Accessed 12.Oct.2020] https://github.com/andrewbutterfield/ccs2csp

A Proof that CSPmn is a Conservative Extension

In order to prove that CSPmn is conservative, we need to define some auxillary functions. First, we uniquely index the prefixes of CSP processes.

Property 1. *Let P be a CSP process.*

$$\begin{aligned}
ix(STOP) &= STOP \\
ix(a \rightsquigarrow P) &= a_i \rightsquigarrow ix_{-i}(P) \\
ix(P \sqcap Q) &= ix_1(P) \sqcap ix_2(Q) \\
ix(P \,\Box\, Q) &= ix_1(P) \,\Box\, ix_2(Q)
\end{aligned}
\qquad
\begin{aligned}
ix(P \underset{a\#m}{\parallel} Q) &= ix_1(P) \underset{B}{\parallel} ix_2(Q) \\
B &\mathrel{\widehat{=}} \{a_i\#m \mid a_i \in \mathscr{A}(ix_1(P)) \cup \mathscr{A}(ix_2(Q))\} \\
ix(P\backslash_{csp}\{a\}) &= ix(P)\backslash_{csp}\{a_i \mid a_i \in \mathscr{A}(ix(P))\} \\
ix(\mu X.P) &= \mu X.ix(P) \\
ix(X) &= X
\end{aligned}$$

where ix_{-i} is some indexing scheme which does not assign the i-index, and ix_1, ix_2 are indexing schemes that assign disjoint indices.

Then, using *ix*-generated indices we generate unique synchronisation indices. Given a set $\{a_i\}$ of parallel prefixes and a number m of processes meant to synchronise together, $g^*_{a_i\#m}$ generates a unique synchronisation index $a_{i_1..i_m}$.

Definition 14. *Let S, B denote sets of indexed events.*

$$g^*_{a_{i_1}\#m}(S, a_{i_1}) \mathrel{\widehat{=}} \{a_{i_1..i_m} \mid i_1 < .. < i_m, \{a_{i_k} \mid 1 < k \le m\} \subseteq S\} \cup \{a_{i_m..i_1} \mid i_m < .. < i_1, \{a_{i_k} \mid 1 < k \le m\} \subseteq S\}$$

$$g^*_{\{a_k\#m_k \mid k\in\mathbf{N}\}}(S, a_i) \mathrel{\widehat{=}} \begin{cases} a_i & a_i \notin \{a_k \mid k \in \mathbf{N}\} \\ g^*_{a_i\#m_i}(S, a_i) & \end{cases}$$

Although $g^*_{a\#m}$ denotes relational renaming, we overload its application to processes such that it translates $\underset{a\#m}{\parallel}$ into $\underset{a}{\parallel}$. This corresponds to the following.

Definition 15. *Let P be an ix-indexed CSP processes. Let S be a set of ix-indexed events. Let a#m denote the set $\{a_k\#m_k \mid k \in \mathbf{N}\}$, b#n the set $\{b_j\#n_j \mid j \in \mathbf{N}\}$. Let $g^*_{a\#m}(P) \mathrel{\widehat{=}} g^*_{a\#m}(\{\}, P)$.*

$$\begin{aligned}
g^*_{a\#m}(S, STOP) &\mathrel{\widehat{=}} STOP \\
g^*_{a\#m}(S, P \sqcap Q) &\mathrel{\widehat{=}} g^*_{a\#m}(S, P) \sqcap g^*_{a\#m}(S, Q) \\
g^*_{a\#m}(S, \mu X.P) &\mathrel{\widehat{=}} \mu X.g^*_{a\#m}(S, P) \\
g^*_{a\#m}(S, P\backslash_{csp}\{a\}) &\mathrel{\widehat{=}} g^*_{a\#m}(S, P)\backslash_{csp} g^*_{a\#m}(S, a) \\
g^*_{a\#m}(S, P \underset{b\#n}{\parallel} Q) &\mathrel{\widehat{=}} g^*_{a\#m\cup b\#n}(S \cup \mathscr{A}(Q), P) \underset{B}{\parallel} g^*_{a\#m\cup b\#n}(S \cup \mathscr{A}(P), Q) \\
B &\mathrel{\widehat{=}} \bigcup\{g^*_{a\#m\cup b\#n}(S \cup \mathscr{A}(Q), b_j) \mid b_j \in \mathscr{A}(P)\} \cup \bigcup\{g^*_{a\#m\cup b\#n}(S \cup \mathscr{A}(P), b_j) \mid b_j \in \mathscr{A}(Q)\}
\end{aligned}
\qquad
\begin{aligned}
g^*_{a\#m}(S, a \rightsquigarrow P) &\mathrel{\widehat{=}} \sum_{b \in g^*_{a\#m}(a)} b \rightsquigarrow g^*_{a\#m}(S, P) \\
g^*_{a\#m}(S, P \,\Box\, Q) &\mathrel{\widehat{=}} g^*_{a\#m}(S, P) \,\Box\, g^*_{a\#m}(S, Q) \\
g^*_{a\#m}(S, X) &\mathrel{\widehat{=}} X
\end{aligned}$$

When $a\#m$ denotes the empty set, we write $g^*_{\#}$ for the corresponding function $g^*_{a\#m}$. Then, the translation of CSPmn into CSP is given by the following.

Definition 16. *Let P be a CSPmn process.* $mn2csp(P) \mathrel{\widehat{=}} g^*_{\#} \circ ix(P)$

The following theorem establishes a labelled operational correspondence (Def. 2), which turns out a strong bisimulation (Def. 3), between CSPmn and CSP.

Theorem 2. *Let P be a CSPmn process. Let I denote a given sequence of natural numbers.*

1. *If* $P \xrightarrow{a} P'$ *then* $\exists I : mn2csp(P) \xrightarrow{a_I} Q$ *and* $Q \equiv mn2csp(P')$
2. *If* $mn2csp(P) \xrightarrow{a_I} Q$ *then* $\exists! P' : P \xrightarrow{a} P'$ *and* $Q \equiv mn2csp(P')$

Proof. When P does not mention $\underset{a\#m}{\|}$, $mn2csp$ behaves like the identity function, hence the theorem holds. By induction, we prove the case for parallel.

(Thrm.2.1.) [Induction step:Parallel]. Let $P_1 \xrightarrow{a} P_1'$. Let $P_2, .., P_n$ denote processes such that $m-1$ among them can perform an a-transition. For ease, we select one such combinations, $P_2..P_m$. The following result applies for all possible combinations. —(Hyp-combine)— Then, by M/N-IndxIfacePar rule (§3),

$$P_1 \underset{a\#m}{\|} .. \underset{a\#m}{\|} P_n \xrightarrow{a} P_1' \underset{a\#m}{\|} P_2' .. \underset{a\#m}{\|} P_m' \underset{a\#m}{\|} P_{m+1} \underset{a\#m}{\|} .. \underset{a\#m}{\|} P_n$$

Assume for each P_i that every occurrence of a in P_i is indexed into a_i. (The following applies even if we separate i into distinct indices, e.g., $i_1, i_2, ..$, as many as there are of instances of a in P_i.) —(Hyp-indx)— Then, by (Hyp-combine), (Hyp-indx), and Def.15, $g^*_{\#}(a) = a_{12..m}$ and:

$$mn2csp(P_1 \underset{a\#m}{\|} .. \underset{a\#m}{\|} P_n) = P_1[a_{12..m}/a] \underset{a_{12..m}}{\|} .. \underset{a_{12..m}}{\|} P_m[a_{12..m}/a] \underset{\{\}}{\|} P_{m+1} \underset{\{\}}{\|} .. \underset{\{\}}{\|} P_n$$

By IndxIfacePar rule (§3) and definition of renaming (Tab.2):

$$P_1[a_{12..m}/a] \underset{a_{12..m}}{\|} .. \underset{a_{12..m}}{\|} P_m[a_{12..m}/a] \underset{\{\}}{\|} P_{m+1} \underset{\{\}}{\|} .. \underset{\{\}}{\|} P_n \xrightarrow{a_{12..m}} P_1'[a_{12..m}/a] \underset{a_{12..m}}{\|} .. \underset{a_{12..m}}{\|} P_m'[a_{12..m}/a] \underset{\{\}}{\|} P_{m+1} \underset{\{\}}{\|} .. \underset{\{\}}{\|} P_n$$

Then, by induction hypothesis.

(Thrm.2.2.) [Induction step: Prallel.] Let $mn2csp(P) \xrightarrow{a_I} Q$. By Par rule, $mn2csp(P) \underset{a_I}{\|} mn2csp(P_2) \xrightarrow{a_I} Q \underset{a_I}{\|} mn2csp(P_2)$, $a_I \notin \mathscr{A}(mn2csp(P_2))$. By induction hypothesis, $\exists! P' : P \xrightarrow{a} P'$ and $Q \equiv mn2csp(P')$. Then, by Par rule, $P \underset{a\#m}{\|} P_2 \xrightarrow{a} P' \underset{a\#m}{\|} P_2$. Moreover, $Q \underset{a_I}{\|} mn2csp(P_2) \equiv mn2csp(P') \underset{a_I}{\|} mn2csp(P_2) = mn2csp(P' \underset{a\#m}{\|} P_2)$, by Def.16. □

As a consequence, when m-among-n CSPmn processes, $\underset{a\#m,j}{\|\|} P_j$, will synchronise on a, m-among-n CSP processes, $\underset{a_{12..m},j}{\|\|} P_j[a_{12..m}/a]$, will synchronise on $a_{12..m}$, where $12..m$ denotes any combination of m potential synchronising processes. We say that *mn2csp* implements m-among-n synchronisation.

Asynchronous Functional Sessions: Cyclic and Concurrent

Bas van den Heuvel Jorge A. Pérez
University of Groningen, The Netherlands
{b.van.den.heuvel, j.a.perez} @ rug.nl

We present Concurrent GV (CGV), a functional calculus with message-passing concurrency governed by session types. With respect to prior calculi, CGV has increased support for concurrent evaluation and for cyclic network topologies. The design of CGV draws on APCP, a session-typed asynchronous π-calculus developed in prior work. Technical contributions are (i) the syntax, semantics, and type system of CGV; (ii) a correct translation of CGV into APCP; (iii) a technique for establishing deadlock-free CGV programs, by resorting to APCP's priority-based type system.

1 Introduction

The goal of this paper is to introduce a new functional calculus with message-passing concurrency governed by linearity and session types. Our work contributes to a research line initiated by Gay and Vasconcelos [8], who proposed a functional calculus with sessions here referred to as λ^{sess}; this line of work has received much recent attention thanks to Wadler's GV calculus [26], which is a variation of λ^{sess}.

Our new calculus is dubbed Concurrent GV (CGV); with respect to previous work, it presents three intertwined novelties: asynchronous (buffered) communication; a highly concurrent reduction strategy; and thread configurations with cyclic topologies. The design of CGV rests upon a solid basis: an operationally correct translation into APCP (Asynchronous Priority-based Classical Processes), a session-typed π-calculus in which asynchronous processes communicate by forming cyclic networks [13].

We discuss the salient features of CGV by example, using a simplified syntax. As in λ^{sess}, communication in CGV is asynchronous: send operations place their messages in buffers, and receive operations read the messages from these buffers. Let us write $\mathsf{send}\ (u,x)$ to denote the output of message u along channel x, and $\mathsf{recv}\ y$ to denote an input on y. The following program expresses the parallel composition ($\|$) of two threads:

$$\begin{array}{l}\mathsf{let}\, x = \mathsf{send}\ (u,x)\, \mathsf{in} \\ \quad \mathsf{let}\, (v,y) = \mathsf{recv}\ y\, \mathsf{in}\, ()\end{array} \quad \Big\| \quad \mathsf{let}\, y = \underline{\mathsf{send}\ (w,y)}\, \mathsf{in}\, ()$$

In variants of λ^{sess} with *synchronous* communication, such as GV and Kokke and Dardha's PGV [17, 18], this program is stuck: the send on y (underlined, on the right) cannot synchronize with the receive on y (on the left): it is blocked by the send on x (on the left), and there is no receive on x. In contrast, in CGV the send on x can be buffered after which the communication on y can take place.

In CGV, reduction is "more concurrent" than usual call-by-value or call-by-name strategies. Consider the following program:

$$\begin{pmatrix}\lambda x\,.\,\mathsf{let}\,(u,y) = \mathsf{recv}\ y\,\mathsf{in} \\ \qquad \mathsf{let}\, x = \mathsf{send}\ (u,x)\,\mathsf{in}()\end{pmatrix} \quad \big(\mathsf{send}\ (v,z)\big)$$

In λ^{sess}, reduction is call-by-value and so the function on x can only be applied on a value. However, the function's parameter (send on z) is not a value, so it needs to be evaluated before the function on x can

V. Castiglioni and C. A. Mezzina (Eds): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics 2022 (EXPRESS/SOS 2022).
EPTCS 368, 2022, pp. 75–94, doi:10.4204/EPTCS.368.5

be applied. Hence, this program can only be evaluated in one order: first the send on z, then the receive on y. In contrast, the semantics of CGV evaluates a function and its parameters *concurrently*: the send on z and the receive on y can be evaluated in any order. Note that asynchrony plays no role here: both buffering a message and synchronous communication entail a reduction in the function's parameter.

The third novelty is cyclic thread configurations: threads can be connected by channels to form cyclic networks. Consider the following program:

$$\begin{array}{l} \mathsf{let}\,(u,x) = \mathsf{recv}\,x\,\mathsf{in} \\ \quad \mathsf{let}\,y = \mathsf{send}\,(u,y)\,\mathsf{in}\,() \end{array} \quad \Big\| \quad \begin{array}{l} \mathsf{let}\,x = \mathsf{send}\,(v,x)\,\mathsf{in} \\ \quad \mathsf{let}\,(w,y) = \mathsf{recv}\,y\,\mathsf{in}\,() \end{array}$$

Here we have two threads connected on channels x and y, thus forming a cyclic thread configuration. Clearly, this program is deadlock-free. In λ^{sess}, the program is well-typed, but there is no deadlock-freedom guarantee: the type system of λ^{sess} admits deadlocked cyclic thread configurations. In GV and Fowler *et al.*'s EGV [7] (an extension of Fowler's AGV [6]) there is a deadlock-freedom guarantee for well-typed programs; however, their type systems only support *tree-shaped* thread configurations—this limitation is studied in [4, 5]. Hence, the program above is not well-typed in GV and EGV.

These novelties are *intertwined*, in the following sense. Asynchronous communication reduces the synchronization points in programs (as output-like operations are non-blocking), therefore increasing concurrent evaluation. In turn, reduced synchronization points can streamline verification techniques for deadlock-freedom based on *priorities* [16, 22, 23, 3], which unlock the analysis of process networks with cyclic topologies. Indeed, in an asynchronous setting only input-like operations require priorities.

We endow CGV with a type system with functional types and session types; we opted for a design in which well-typed terms enjoy subject reduction / type preservation but not deadlock-freedom. To validate our semantic design and attain the three novelties motivated above, we resort to APCP. In our developments, APCP operates as a "low-level" reference programming model. We give a typed translation of CGV into APCP, which satisfies strong correctness properties, in the sense of Gorla [10]. In particular, it enjoys operational correspondence, which provides a significant sanity check to justify our key design decisions in CGV's operational semantics. Interestingly, using our correct translation and the deadlock-freedom guarantees for well-typed processes in APCP, we obtain a technique for transferring the deadlock-freedom property to CGV programs. That is, given a CGV program C, we prove that if the APCP translation of C is typable (and hence, deadlock-free), then C itself is deadlock-free. This result thus delineates a class of deadlock-free CGV programs that includes cyclic thread configurations.

In summary, this paper presents the following technical contributions: (1) CGV, a new functional calculus with session-based asynchronous concurrency; (2) A typed translation of CGV into APCP, which is proven to satisfy well-studied encodability criteria; (3) A transference result for the deadlock-freedom property from APCP to CGV programs. An extended version contains omitted technical details [14].

2 Concurrent GV

2.1 Syntax and Semantics

The main syntactic entities in CGV are *terms*, *runtime terms*, and *configurations*. Intuitively, terms reduce to runtime terms; configurations correspond to the parallel composition of a main thread and several child threads, each executing a runtime term. Buffered messages are part of configurations. We define two reduction relations: one is on terms, which is then subsumed by reduction on configurations.

The syntax of terms (L,M,N) is given and described in Figure 1. We use $x,y,\ldots$ for *variables*; we write *endpoint* to refer to a variable used for session operations (send, receive, select, offer). Let

Terms (L, M, N):

x	(variable)	new	(create new channel)
$()$	(unit value)	$\mathsf{spawn}\ M$	(execute pair M in parallel)
$\lambda x.M$	(abstraction)	$\mathsf{send}\ (M,N)$	(send M along N)
$M\ N$	(application)	$\mathsf{recv}\ M$	(receive along M)
(M,N)	(pair construction)	$\mathsf{select}\ \ell\ M$	(select label ℓ along M)
$\mathsf{let}\,(x,y) = M\,\mathsf{in}\,N$	(pair deconstruction)	$\mathsf{case}\,M\,\mathsf{of}\,\{i : M\}_{i\in I}$	(offer labels in I along M)

Runtime terms $(\mathbb{L}, \mathbb{M}, \mathbb{N})$ and reduction contexts $(\mathscr{R})$:

$$\begin{aligned}\mathbb{L},\mathbb{M},\mathbb{N} ::= {} & \ldots \mid \mathbb{M}\{\!|\mathbb{N}/x|\!\} \mid \mathsf{send}'(\mathbb{M},\mathbb{N}) \\ \mathscr{R} ::= {} & [] \mid \mathscr{R}\ \mathbb{M} \mid \mathsf{spawn}\ \mathscr{R} \mid \mathsf{send}\ \mathscr{R} \mid \mathsf{recv}\ \mathscr{R} \mid \mathsf{let}\,(x,y) = \mathscr{R}\,\mathsf{in}\,\mathbb{M} \\ & \mid \mathsf{select}\ \ell\ \mathscr{R} \mid \mathsf{case}\,\mathscr{R}\,\mathsf{of}\,\{i : \mathbb{M}\}_{i\in I} \mid \mathscr{R}\{\!|\mathbb{M}/x|\!\} \mid \mathbb{M}\{\!|\mathscr{R}/x|\!\} \mid \mathsf{send}'(\mathbb{M},\mathscr{R})\end{aligned}$$

Structural congruence for terms $(\equiv_{\mathtt{M}})$ and term reduction $(\longrightarrow_{\mathtt{M}})$:

SC-SubExt	$x \notin \mathrm{fn}(\mathscr{R}) \Rightarrow (\mathscr{R}[\mathbb{M}])\{\!|\mathbb{N}/x|\!\} \equiv_{\mathtt{M}} \mathscr{R}[\mathbb{M}\{\!|\mathbb{N}/x|\!\}]$
E-Lam	$(\lambda x.\mathbb{M})\ \mathbb{N} \longrightarrow_{\mathtt{M}} \mathbb{M}\{\!|\mathbb{N}/x|\!\}$
E-Pair	$\mathsf{let}\,(x,y) = (\mathbb{M}_1,\mathbb{M}_2)\,\mathsf{in}\,\mathbb{N} \longrightarrow_{\mathtt{M}} \mathbb{N}\{\!|\mathbb{M}_1/x, \mathbb{M}_2/y|\!\}$
E-SubstName	$\mathbb{M}\{\!|y/x|\!\} \longrightarrow_{\mathtt{M}} \mathbb{M}\{y/x\}$
E-NameSubst	$x\{\!|\mathbb{M}/x|\!\} \longrightarrow_{\mathtt{M}} \mathbb{M}$
E-Send	$\mathsf{send}\ (\mathbb{M},\mathbb{N}) \longrightarrow_{\mathtt{M}} \mathsf{send}'(\mathbb{M},\mathbb{N})$
E-Lift	$\mathbb{M} \longrightarrow_{\mathtt{M}} \mathbb{N} \Rightarrow \mathscr{R}[\mathbb{M}] \longrightarrow_{\mathtt{M}} \mathscr{R}[\mathbb{N}]$
E-LiftSC	$\mathbb{M} \equiv_{\mathtt{M}} \mathbb{M}' \wedge \mathbb{M}' \longrightarrow_{\mathtt{M}} \mathbb{N}' \wedge \mathbb{N}' \equiv_{\mathtt{M}} \mathbb{N} \Rightarrow \mathbb{M} \longrightarrow_{\mathtt{M}} \mathbb{N}$

Figure 1: The CGV term language.

$\mathrm{fn}(M)$ denote the free variables of a term. All variables are free unless bound: $\lambda x.M$ binds x in M, and $\mathsf{let}\,(x,y) = M\,\mathsf{in}\,N$ binds x and y in N. We introduce syntactic sugar for applications of abstractions: $\mathsf{let}\,x = M\,\mathsf{in}\,N$ denotes $(\lambda x.N)\ M$. For $(\lambda x.M)\ N$, we assume $x \notin \mathrm{fn}(N)$, and for $\mathsf{let}\,(x,y) = M\,\mathsf{in}\,N$, we assume $x \neq y$ and $x, y \notin \mathrm{fn}(M)$.

Figure 1 also gives the reduction semantics of CGV terms $(\longrightarrow_{\mathtt{M}})$, which relies on runtime terms $(\mathbb{L}, \mathbb{M}, \mathbb{N})$, reduction contexts $(\mathscr{R})$, and structural congruence $(\equiv_{\mathtt{M}})$. Note that this semantics comprises the functional fragment of CGV; we define the concurrent semantics of CGV hereafter.

Runtime terms, whose syntax extends that of terms, guide the evaluation strategy of CGV; we discuss an example evaluation of a term using runtime terms after introducing the reduction rules (Example 2.1). Explicit substitution $\mathbb{M}\{\!|\mathbb{N}/x|\!\}$ enables the concurrent execution of a function and its parameters. The intermediate primitive $\mathsf{send}'(\mathbb{M},\mathbb{N})$ enables $\mathbb{N}$ to reduce to an endpoint; the send primitive takes a pair of terms as an argument, inside which reduction is not permitted (cf. [8]). Reduction contexts define the

Markers (ϕ), messages (m,n), configurations (C,D,E), thread ($\mathscr{F}$) and configuration ($\mathscr{G}$) contexts:

$$\phi ::= \blacklozenge \mid \lozenge \qquad m,n ::= \mathbb{M} \mid \ell$$
$$C,D,E ::= \phi\,\mathbb{M} \mid C \parallel D \mid (\boldsymbol{\nu} x[\vec{m}\rangle y)C \mid C\{\!|\mathbb{M}/x|\!\}$$
$$\mathscr{F} ::= \phi\,\mathscr{R} \mid C\{\!|\mathscr{R}/x|\!\} \qquad \mathscr{G} ::= [] \mid \mathscr{G} \parallel C \mid (\boldsymbol{\nu} x[\vec{m}\rangle y)\mathscr{G} \mid \mathscr{G}\{\!|\mathbb{M}/x|\!\}$$

Structural congruence for configurations ($\equiv_{\mathsf{C}}$):

SC-TermSC	$\mathbb{M} \equiv_{\mathtt{M}} \mathbb{M}' \Rightarrow \phi\,\mathbb{M} \equiv_{\mathsf{C}} \phi\,\mathbb{M}'$
SC-ResSwap	$(\boldsymbol{\nu} x[\varepsilon\rangle y)C \equiv_{\mathsf{C}} (\boldsymbol{\nu} y[\varepsilon\rangle x)C$
SC-ResComm	$(\boldsymbol{\nu} x[\vec{m}\rangle y)(\boldsymbol{\nu} z[\vec{n}\rangle w)C \equiv_{\mathsf{C}} (\boldsymbol{\nu} z[\vec{n}\rangle w)(\boldsymbol{\nu} x[\vec{m}\rangle y)C$
SC-ResExt	$x,y \notin \mathrm{fn}(C) \Rightarrow (\boldsymbol{\nu} x[\vec{m}\rangle y)(C \parallel D) \equiv_{\mathsf{C}} C \parallel (\boldsymbol{\nu} x[\vec{m}\rangle y)D$
SC-ResNil	$x,y \notin \mathrm{fn}(C) \Rightarrow (\boldsymbol{\nu} x[\varepsilon\rangle y)C \equiv_{\mathsf{C}} C$
SC-Send'	$(\boldsymbol{\nu} x[\vec{m}\rangle y)(\hat{\mathscr{F}}[\mathsf{send}'(\mathbb{M},x)] \parallel C) \equiv_{\mathsf{C}} (\boldsymbol{\nu} x[\mathbb{M},\vec{m}\rangle y)(\hat{\mathscr{F}}[x] \parallel C)$
SC-Select	$(\boldsymbol{\nu} x[\vec{m}\rangle y)(\mathscr{F}[\mathsf{select}\,\ell\,x] \parallel C) \equiv_{\mathsf{C}} (\boldsymbol{\nu} x[\ell,\vec{m}\rangle y)(\mathscr{F}[x] \parallel C)$
SC-ParNil	$C \parallel \lozenge() \equiv_{\mathsf{C}} C$
SC-ParComm	$C \parallel D \equiv_{\mathsf{C}} D \parallel C$
SC-ParAssoc	$C \parallel (D \parallel E) \equiv_{\mathsf{C}} (C \parallel D) \parallel E$
SC-ConfSubst	$\phi\,(\mathbb{M}\{\!|\mathbb{N}/x|\!\}) \equiv_{\mathsf{C}} (\phi\,\mathbb{M})\{\!|\mathbb{N}/x|\!\}$
SC-ConfSubstExt	$x \notin \mathrm{fn}(\mathscr{G}) \Rightarrow (\mathscr{G}[C])\{\!|\mathbb{M}/x|\!\} \equiv_{\mathsf{C}} \mathscr{G}[C\{\!|\mathbb{M}/x|\!\}]$

Configuration reduction ($\longrightarrow_{\mathsf{C}}$):

E-New	$\mathscr{F}[\mathsf{new}] \longrightarrow_{\mathsf{C}} (\boldsymbol{\nu} x[\varepsilon\rangle y)(\mathscr{F}[(x,y)])$
E-Spawn	$\hat{\mathscr{F}}[\mathsf{spawn}\ (\mathbb{M},\mathbb{N})] \longrightarrow_{\mathsf{C}} \hat{\mathscr{F}}[\mathbb{N}] \parallel \lozenge\,\mathbb{M}$
E-Recv	$(\boldsymbol{\nu} x[\vec{m},\mathbb{M}\rangle y)(\hat{\mathscr{F}}[\mathsf{recv}\ y] \parallel C) \longrightarrow_{\mathsf{C}} (\boldsymbol{\nu} x[\vec{m}\rangle y)(\hat{\mathscr{F}}[(\mathbb{M},y)] \parallel C)$
E-Case	$j \in I \Rightarrow (\boldsymbol{\nu} x[\vec{m},j\rangle y)(\mathscr{F}[\mathsf{case}\,y\,\mathsf{of}\,\{i : \mathbb{M}_i\}_{i\in I}] \parallel C) \longrightarrow_{\mathsf{C}} (\boldsymbol{\nu} x[\vec{m}\rangle y)(\mathscr{F}[\mathbb{M}_j\ y] \parallel C)$
E-LiftC	$C \longrightarrow_{\mathsf{C}} C' \Rightarrow \mathscr{G}[C] \longrightarrow_{\mathsf{C}} \mathscr{G}[C']$
E-LiftM	$\mathbb{M} \longrightarrow_{\mathtt{M}} \mathbb{M}' \Rightarrow \mathscr{F}[\mathbb{M}] \longrightarrow_{\mathsf{C}} \mathscr{F}[\mathbb{M}']$
E-ConfLiftSC	$C \equiv_{\mathsf{C}} C' \wedge C' \longrightarrow_{\mathsf{C}} D' \wedge D' \equiv_{\mathsf{C}} D \Rightarrow C \longrightarrow_{\mathsf{C}} D$

Figure 2: The CGV configuration language.

non-blocking parts of terms, where subterms may reduce. We write $\mathscr{R}[\mathbb{M}]$ to denote the runtime term obtained by replacing the hole $[]$ in $\mathscr{R}$ by $\mathbb{M}$, and $\mathrm{fn}(\mathscr{R})$ to denote $\mathrm{fn}(\mathscr{R}[()])$; we will use similar notation for other kinds of contexts later.

We discuss the reduction rules. The Structural congruence rule SC-SubExt allows the scope extrusion of explicit substitutions along reduction contexts. Rule E-Lam enforces application, resulting in an explicit substitution. Rule E-Pair unpacks the elements of a pair into two explicit substitutions

(arbitrarily ordered, due to the syntactical assumptions introduced above). Rules E-SUBSTNAME and E-NAMESUBST convert explicit substitutions of or on variables into standard substitutions. Rule E-SEND reduces a send into a send′ primitive. Rules E-LIFT and E-LIFTSC close term reduction under contexts and structural congruence, respectively. We write $\mathbb{M} \longrightarrow_{\mathtt{M}}^{k} \mathbb{N}$ to denote that $\mathbb{M}$ reduces to $\mathbb{N}$ in k steps.

Example 2.1. *We illustrate the evaluation of terms using runtime terms through the following example, which contains a* send *primitive and nested abstractions and applications. In each reduction step, we underline the subterm that reduces and give the applied rule:*

$$
\begin{array}{ll}
\underline{(\lambda x.\,\mathsf{send}\ ((),x))\ ((\lambda y.\,y)\ z)} & \text{(E-LAM)} \\
\longrightarrow_{\mathtt{M}} \underline{(\mathsf{send}\ ((),x))}\,\{\!|\,((\lambda y.\,y)\ z)/x\,|\!\} & \text{(E-SEND)} \\
\longrightarrow_{\mathtt{M}} (\mathsf{send}'((),x))\,\{\!|\,\underline{((\lambda y.\,y)\ z)}/x\,|\!\} & \text{(E-LAM)} \\
\longrightarrow_{\mathtt{M}} (\mathsf{send}'((),x))\,\{\!|\,(y\{\!|\,z/y\,|\!\})/x\,|\!\} & \text{(SC-SUBEXT)} \\
\equiv_{\mathtt{M}} \mathsf{send}'((),\underline{x\{\!|\,(y\{\!|\,z/y\,|\!\})/x\,|\!\}}) & \text{(E-NAMESUBST)} \\
\longrightarrow_{\mathtt{M}} \mathsf{send}'((),\underline{y\{\!|\,z/y\,|\!\}}) & \text{(E-SUBSTNAME)} \\
\longrightarrow_{\mathtt{M}} \mathsf{send}'((),z) &
\end{array}
$$

Notice how the send *primitive needs to reduce to a* send′ *runtime primitive such that the explicit substitution of x can be applied. Also, note that the concurrency of CGV allows many more paths of reduction.*

Note that the concurrent evaluation strategy of CGV may also be defined without explicit substitutions. In principle, this would require additional reduction contexts specific to applications on abstractions and pair deconstruction, as well as variants of Rules E-SUBSTNAME and E-NAMESUBST specific to these contexts. However, it is not clear how to define scope extrusion (Rule SC-SUBEXT) for such a semantics. Hence, we find that using explicit substitutions drastically simplifies the semantics of CGV.

Concurrency in CGV allows the parallel execution of terms that communicate through buffers. The syntax of configurations (C,D,E) is given in Figure 2. The configuration $\phi\,\mathbb{M}$ denotes a *thread*: a concurrently executed term. The thread marker helps to distinguish the *main* thread ($\phi = \blacklozenge$) from *child* threads ($\phi = \lozenge$). The configuration $C \,\|\, D$ denotes parallel composition. The configuration $(\boldsymbol{\nu} x[\vec{m}\rangle y)C$ denotes a *buffered restriction*: it connects the endpoints x and y through a buffer $[\vec{m}\rangle$, binding x and y in C. The buffer's content, $\vec{m}$, is a sequence of messages (terms and labels). Buffers are directed: in $x[\vec{m}\rangle y$, messages can be added to the front of the buffer on x, and they can be taken from the back of the buffer on y. We write $[\varepsilon\rangle$ for the empty buffer. The configuration $C\{\!|\,\mathbb{M}/x\,|\!\}$ lifts explicit substitution to the level of configurations: this allows spawning and sending terms under explicit substitution, such that the substitution can be moved to the context of the spawned or sent term.

The reduction semantics for configurations ($\longrightarrow_{\mathtt{C}}$, also in Figure 2) relies on thread and configuration contexts ($\mathscr{F}$ and $\mathscr{G}$, respectively) and structural congruence ($\equiv_{\mathtt{C}}$). We write $\hat{\mathscr{F}}$ to denote a thread context in which the hole does not occur under explicit substitution, i.e. the context is not constructed using the clause $\mathscr{R}\{\!|\,\mathbb{M}/x\,|\!\}$; this is used in rules for send′, spawn, and recv, effectively forcing the scope extrusion of explicit substitutions when terms are moved between contexts (cf. Example 2.4).

We comment on some of the congruences and reduction rules. Rule SC-RESSWAP allows to swap the direction of an empty buffer; this way, the endpoint that could read from the buffer before the swap can now write to it. Rule SC-RESCOMM allows to interchange buffers, and Rule SC-RESEXT allows to extrude their scope. Rule SC-RESNIL garbage collects buffers of closed sessions. Rule SC-CONFSUBST lifts explicit substitution at the level of terms to the level of threads, and

Rule SC-ConfSubstExt allows the scope extrusion of explicit substitution along configuration contexts. Notably, putting messages in buffers is not a reduction: Rules SC-Send' and SC-Select *equate* sends and selects on an endpoint x with terms and labels in the buffer for x, as asynchronous outputs are computationally equivalent to messages in buffers.

Reduction rule E-New creates a new buffer, leaving a reference to the newly created endpoints in the thread. Rule E-Spawn spawns a child thread (the parameter pair's first element) and continues (as the pair's second element) inside the calling thread. Rule E-Recv retrieves a term from a buffer, resulting in a pair containing the term and a reference to the receiving endpoint. Rule E-Case retrieves a label from a buffer, resulting in a function application of the label's corresponding branch to a reference to the receiving endpoint. There are no reduction rules for closing sessions, as they are closed silently. We write $C \longrightarrow_{\mathsf{C}}^{k} D$ to denote that C reduces to D in k steps. Also, we write $\longrightarrow_{\mathsf{C}}^{+}$ to denote the transitive closure of $\longrightarrow_{\mathsf{C}}$ (i.e., reduction in at least one step).

We illustrate CGV's semantics by giving some examples. The following discusses a cyclic thread configuration which does not deadlock due to asynchrony:

Example 2.2. *Consider configuration C_1 below, in which two threads are spawned and cyclically connected through two channels. One thread first sends on the first channel and then receives on the second, while the other thread first sends on the second channel and then receives on the first. Under synchronous communication, this would determine a configuration that deadlocks; however, under asynchronous communication, this is not the case (cf. the third example in Sec. 1). We detail some interesting reductions:*

$$C_1 = \blacklozenge(\mathsf{let}\,(f,g) = \mathsf{new}\,\mathsf{in}\,\mathsf{let}\,(h,k) = \mathsf{new}\,\mathsf{in}\,\mathsf{spawn}\begin{pmatrix}\mathsf{let}\,f' = (\mathsf{send}\ (u,f))\,\mathsf{in}\\ \quad \mathsf{let}\,(v',h') = (\mathsf{recv}\ h)\,\mathsf{in}\,(),\\ \mathsf{let}\,k' = (\mathsf{send}\ (v,k))\,\mathsf{in}\\ \quad \mathsf{let}\,(u',g') = (\mathsf{recv}\ g)\,\mathsf{in}\,()\end{pmatrix})$$

$$\longrightarrow_{\mathsf{C}}^{8} (\boldsymbol{\nu} x[\varepsilon\rangle y)(\boldsymbol{\nu} w[\varepsilon\rangle z)(\blacklozenge(\mathsf{spawn}\begin{pmatrix}\mathsf{let}\,f' = (\mathsf{send}\ (u,x))\,\mathsf{in} & \mathsf{let}\,k' = (\mathsf{send}\ (v,z))\,\mathsf{in}\\ \quad \mathsf{let}\,(v',h') = (\mathsf{recv}\ w)\,\mathsf{in}\,(), & \quad \mathsf{let}\,(u',g') = (\mathsf{recv}\ y)\,\mathsf{in}\,()\end{pmatrix}) \tag{1}$$

$$\longrightarrow_{\mathsf{C}} (\boldsymbol{\nu} x[\varepsilon\rangle y)(\boldsymbol{\nu} w[\varepsilon\rangle z)(\blacklozenge\begin{pmatrix}\mathsf{let}\,k' = (\mathsf{send}\ (v,z))\,\mathsf{in}\\ \quad \mathsf{let}\,(u',g') = (\mathsf{recv}\ y)\,\mathsf{in}\,()\end{pmatrix} \parallel \Diamond \begin{pmatrix}\mathsf{let}\,f' = (\mathsf{send}\ (u,x))\,\mathsf{in}\\ \quad \mathsf{let}\,(v',h') = (\mathsf{recv}\ w)\,\mathsf{in}\,()\end{pmatrix}) \tag{2}$$

$$\longrightarrow_{\mathsf{C}}^{2} (\boldsymbol{\nu} x[\varepsilon\rangle y)(\boldsymbol{\nu} w[\varepsilon\rangle z)(\blacklozenge\begin{pmatrix}(\mathsf{let}\,(u',g') = (\mathsf{recv}\ y)\,\mathsf{in}\,())\\ \{\!|\mathsf{send}\ (v,z)/k'|\!\}\end{pmatrix} \parallel \Diamond \begin{pmatrix}(\mathsf{let}\,(v',h') = (\mathsf{recv}\ w)\,\mathsf{in}\,())\\ \{\!|\mathsf{send}\ (u,x)/f'|\!\}\end{pmatrix}) \tag{3}$$

$$\longrightarrow_{\mathsf{C}}^{2} (\boldsymbol{\nu} x[\varepsilon\rangle y)(\boldsymbol{\nu} w[\varepsilon\rangle z)(\blacklozenge\begin{pmatrix}(\mathsf{let}\,(u',g') = (\mathsf{recv}\ y)\,\mathsf{in}\,())\\ \{\!|\mathsf{send}'(v,z)/k'|\!\}\end{pmatrix} \parallel \Diamond \begin{pmatrix}(\mathsf{let}\,(v',h') = (\mathsf{recv}\ w)\,\mathsf{in}\,())\\ \{\!|\mathsf{send}'(u,x)/f'|\!\}\end{pmatrix}) \tag{4}$$

$$\equiv (\boldsymbol{\nu} x[u\rangle y)(\boldsymbol{\nu} z[v\rangle w)(\blacklozenge((\mathsf{let}\,(u',g') = (\mathsf{recv}\ y)\,\mathsf{in}\,())\{\!|z/k'|\!\}) \parallel \Diamond((\mathsf{let}\,(v',h') = (\mathsf{recv}\ w)\,\mathsf{in}\,())\{\!|x/f'|\!\})) \tag{5}$$

$$\longrightarrow_{\mathsf{C}}^{2} (\boldsymbol{\nu} x[u\rangle y)(\boldsymbol{\nu} z[v\rangle w)(\blacklozenge(\mathsf{let}\,(u',g') = (\mathsf{recv}\ y)\,\mathsf{in}\,()) \parallel \Diamond(\mathsf{let}\,(v',h') = (\mathsf{recv}\ w)\,\mathsf{in}\,()))$$

$$\longrightarrow_{\mathsf{C}}^{2} (\boldsymbol{\nu} x[\varepsilon\rangle y)(\boldsymbol{\nu} z[\varepsilon\rangle w)(\blacklozenge(\mathsf{let}\,(u',g') = (v,y)\,\mathsf{in}\,()) \parallel \Diamond(\mathsf{let}\,(v',h') = (v,w)\,\mathsf{in}\,())) \longrightarrow_{\mathsf{C}}^{4} \blacklozenge() \tag{6}$$

Intuitively, reduction (1) instantiates two buffers and assigns the endpoints through explicit substitutions. Reduction (2) spawns the left term as a child thread. Reduction (3) turns lets *into explicit substitutions. Reduction (4) turns the* sends *into* send′*s. Structural congruence (5) equates the* send′*s with messages in the buffers. Reduction (6) retrieves the messages from the buffers. Note that many of these steps represent several reductions that may happen in any order.*

The following example illustrates CGV's flexibility for communicating functions over channels:

Example 2.3. *In the following configuration, a buffer and two threads have already been set up (cf. Example 2.2 for an illustration of such an initialization). The main thread sends an interesting term to the child thread: it contains the* send *primitive from which the main thread will subsequently receive from the child thread. We give the configuration's major reductions, with the reducing parts underlined:*

$$
\begin{aligned}
&(\boldsymbol{\nu} x[\varepsilon\rangle y)(\blacklozenge\left(\frac{\mathsf{let}\, x' = \mathsf{send}\,\left(\lambda z.\,\mathsf{send}\,((),z),x\right)\,\mathsf{in}}{\mathsf{let}\,(v,x'') = \mathsf{recv}\,x'\,\mathsf{in}\,v}\right) \parallel \Diamond \begin{pmatrix}\mathsf{let}\,(w,y') = \mathsf{recv}\,y\,\mathsf{in}\\ \mathsf{let}\,y'' = (w\,y')\,\mathsf{in}\,()\end{pmatrix})\\
\longrightarrow_{\mathsf{C}}^{3}\ &(\boldsymbol{\nu} x[\lambda z.\,\mathsf{send}\,((),z)\rangle y)\left(\blacklozenge\left(\mathsf{let}\,(v,x'') = \mathsf{recv}\,x\,\mathsf{in}\,v\right) \parallel \Diamond \begin{pmatrix}\mathsf{let}\,(w,y') = \underline{\mathsf{recv}\,y}\,\mathsf{in}\\ \mathsf{let}\,y'' = (w\,y')\,\mathsf{in}\,()\end{pmatrix}\right)\\
\longrightarrow_{\mathsf{C}}\ &(\boldsymbol{\nu} y[\varepsilon\rangle x)\left(\blacklozenge\left(\mathsf{let}\,(v,x'') = \mathsf{recv}\,x\,\mathsf{in}\,v\right) \parallel \Diamond \left(\frac{\mathsf{let}\,(w,y') = (\lambda z.\,\mathsf{send}\,((),z),y)\,\mathsf{in}}{\mathsf{let}\,y'' = (w\,y')\,\mathsf{in}\,()}\right)\right)\\
\longrightarrow_{\mathsf{C}}\ &(\boldsymbol{\nu} y[\varepsilon\rangle x)\left(\blacklozenge\left(\mathsf{let}\,(v,x'') = \mathsf{recv}\,x\,\mathsf{in}\,v\right) \parallel \Diamond \left(\underline{\left(\mathsf{let}\,y'' = (w\,y')\,\mathsf{in}\,()\right)\{\!\{\left(\lambda z.\,\mathsf{send}\,((),z)\right)/w,y/y'\}\!\}}\right)\right)\\
\longrightarrow_{\mathsf{C}}^{2}\ &(\boldsymbol{\nu} y[\varepsilon\rangle x)\left(\blacklozenge\left(\mathsf{let}\,(v,x'') = \mathsf{recv}\,x\,\mathsf{in}\,v\right) \parallel \Diamond \left(\mathsf{let}\,y'' = \underline{\left(\left(\lambda z.\,\mathsf{send}\,((),z)\right)\,y\right)}\,\mathsf{in}\,()\right)\right)\\
\longrightarrow_{\mathsf{C}}^{2}\ &(\boldsymbol{\nu} y[\varepsilon\rangle x)\left(\blacklozenge\left(\mathsf{let}\,(v,x'') = \mathsf{recv}\,x\,\mathsf{in}\,v\right) \parallel \Diamond \left(\underline{\mathsf{let}\,y'' = \mathsf{send}\,((),y)\,\mathsf{in}\,()}\right)\right)\\
\longrightarrow_{\mathsf{C}}^{3}\ &(\boldsymbol{\nu} y[()\rangle x)\left(\blacklozenge\left(\underline{\mathsf{let}\,(v,x'') = \mathsf{recv}\,x\,\mathsf{in}\,v}\right)\right) \longrightarrow_{\mathsf{C}}^{4} \blacklozenge\,()
\end{aligned}
$$

The following example illustrates why the restricted thread context $\hat{\mathscr{F}}$ is used:

Example 2.4. *Consider the configuration* $C = (\boldsymbol{\nu} x[\varepsilon\rangle y)\left(\blacklozenge\left(\left(\mathsf{send}'(z,x)\right)\{\!\{v/z\}\!\}\right) \parallel D\right)$. *Suppose Structural congruence rule* SC-SEND' *were defined on unrestricted thread contexts; then the rule applies under the explicit substitution of* z: $C \equiv_{\mathsf{C}} (\boldsymbol{\nu} x[z\rangle y)\left(\blacklozenge\left(x\{\!\{v/z\}\!\}\right) \parallel D\right)$. *Here,* C *and the right-hand-side are inconsistent with each other: in* C*, the variable* z *is bound by the explicit substitution, whereas* z *is free on the right-hand-side. With the restricted thread contexts we are forced to first extrude the scope of the explicit substitution before applying Rule* SC-SEND', *making sure that* z *remains bound:*

$$
C \equiv_{\mathsf{C}} \left((\boldsymbol{\nu} x[\varepsilon\rangle y)\left(\blacklozenge\left(\mathsf{send}'(z,x)\right) \parallel D\right)\right)\{\!\{v/z\}\!\} \equiv_{\mathsf{C}} \left((\boldsymbol{\nu} x[z\rangle y)(\blacklozenge x \parallel D)\right)\{\!\{v/z\}\!\}.
$$

2.2 Type System

We define a type system for CGV, with functional types for functions and pairs and session types for communication. The syntax and meaning of functional types (T, U) and session types (S) are as follows:

$$
\begin{array}{llllllllll}
T,U ::= & T \times U & \text{(pair)} & \mid T \multimap U & \text{(function)} & \mid \mathbf{1} & \text{(unit)} & \mid S & \text{(session)} \\
S ::= & !T.S & \text{(output)} & \mid ?T.S & \text{(input)} & \mid \oplus\{i : T\}_{i\in I} & \text{(select)} & \mid \&\{i : T\}_{i\in I} & \text{(case)} \quad \mid \mathsf{end}
\end{array}
$$

Session type duality ($\overline{S}$) is defined as usual; note that only the continuations, and not the messages, of output and input types are dualized.

$$
\overline{!T.S} = ?T.\overline{S} \quad \overline{?T.S} = !T.\overline{S} \quad \overline{\oplus\{i : S_i\}_{i\in I}} = \&\{i : \overline{S_i}\}_{i\in I} \quad \overline{\&\{i : S_i\}_{i\in I}} = \oplus\{i : \overline{S_i}\}_{i\in I} \quad \overline{\mathsf{end}} = \mathsf{end}
$$

Typing judgments use typing environments (Γ, Δ, Λ) consisting of types assigned to variables ($x : T$). We write $\emptyset$ to denote the empty environment; in writing 'Γ, Δ', we assume that the variables in Γ and Δ

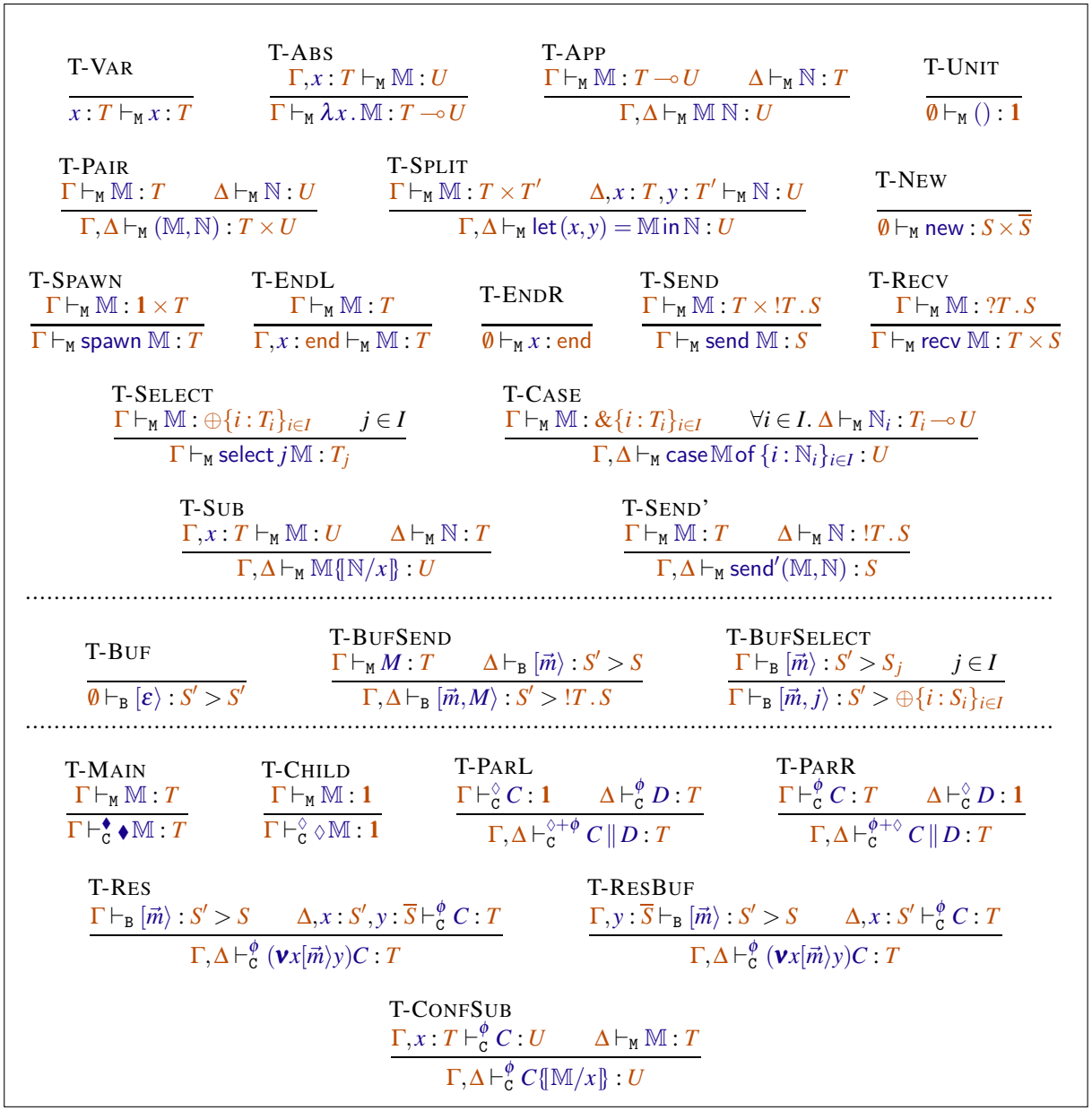


Figure 3: Typing rules for terms (top), buffers (center), and configurations (bottom).

are pairwise distinct. Figure 3 (top) gives the type system for (runtime) terms. Judgments are denoted $\Gamma \vdash_{\mathtt{M}} \mathbb{M} : T$ and have a *use-provide* reading: term $\mathbb{M}$ *uses* the variables in Γ to *provide* a behavior of type T (cf. Caires and Pfenning [1]). When a term provides type T, we often say that the term is of type T.

Typing rules T-Var, T-Abs, T-App, T-Unit, T-Pair, and T-Split are standard. Rule T-New types a pair of dual session types $S \times \overline{S}$. Rule T-Spawn types spawning a $\mathbf{1}$-typed term as a child thread, continuing as a term of type T. Rules T-EndL and T-EndR type finished sessions. Rule T-Send (resp. T-Recv) uses a term of type $!T.S$ (resp. $?T.S$) to type a send (resp. receive) of a term of type T, continuing as type S. Rule T-Select uses a term of type $\oplus\{i : T_i\}_{i \in I}$ to type selecting a label

$j \in I$, continuing as type T_j. Rule T-CASE uses a term of type $\&\{i : T_i\}_{i \in I}$ to type branching on labels $i \in I$, continuing as type U—each branch is typed $T_i \multimap U$. Rule T-SUB types an explicit substitution. Rule T-SEND' types sending directly, not requiring a pair but two separate terms.

Figure 3 (bottom) gives the typing rules for configurations. The typing judgments here are annotated with a thread marker: $\Gamma \vdash^{\phi}_{\mathsf{C}} C : T$. The thread marker serves to keep track of whether the typed configuration contains the main thread or not (i.e. $\phi = \blacklozenge$ if so, and $\phi = \lozenge$ otherwise). When typing the parallel composition of two configurations, we thus have to combine the thread markers of their judgments. This combination of thread markers ($\phi + \phi'$) is defined as follows:

$$\blacklozenge + \lozenge = \blacklozenge \qquad \lozenge + \blacklozenge = \blacklozenge \qquad \lozenge + \lozenge = \lozenge \qquad (\blacklozenge + \blacklozenge \text{ is undefined})$$

Typing rules T-MAIN and T-CHILD turn a typed term into a thread, where child threads may only be of type $\mathbf{1}$. Rules T-PARL and T-PARR compose configurations: one configuration must be of type $\mathbf{1}$ and have thread marker $\lozenge$ (i.e., it does not contain a main thread), providing the other configuration's type. Rule T-RES types buffered restriction, with output endpoint x and input endpoint y used in the configuration. It is possible to send the endpoint y on x, so there is also a Rule T-RESBUF where y is used in the buffer. Unlike with usual typing rules for restriction, the types S' of x and $\overline{S}$ of y do not necessarily have to be duals. This is because the restriction's buffer may already contain messages sent on x but not yet received on y, such that the restricted configuration only needs to use x according to a continuation of S. To ensure that S' is indeed a continuation of S in accordance with the messages in the buffer, we have additional typing rules for buffers, which we explain hereafter. Finally, Rule T-CONFSUB types an explicit substitution on the level of configurations.

For typing buffers, in Figure 3 (center), we have judgments of the form: $\Gamma \vdash_{\mathsf{B}} [\vec{m}\rangle : S' > S$. The judgment denotes that S' is a continuation of S, in accordance with the messages $\vec{m}$, which use the variables in Γ. The idea of the typing rules is that, starting with an empty buffer at the top of the typing derivation (Rule T-BUF) where $S' = S$, Rules T-BUFSEND and T-BUFSELECT add messages to the end of the buffer. Rule T-BUFSEND then prefixes S with an output of the sent term's type, and Rule T-BUFSELECT prefixes S with a selection such that the sent label's continuation is S.

Example 2.5. *Figure 4 (top) shows the typing derivation of a configuration reduced from C_1 in Example 2.2 (following an alternative path after Reduction (5)). Figure 4 (bottom) shows the typing of the configuration $(\boldsymbol{\nu} x[M, \ell, ()\rangle y)C$, which has some messages in a buffer; notice how the type of x in C is a continuation of the dual of the type of y.*

In the configuration $(\boldsymbol{\nu} x[\mathsf{let}\,(z,y) = \mathsf{recv}\,y\,\mathsf{in}\,y\rangle y)C$, the endoint y is inside the buffer connecting it with x; to type it, we need Rule T-RESBUF *(omitting the derivation of the buffer):*

$$\frac{y : ?\mathsf{end}\,.\,\mathsf{end} \vdash_{\mathsf{B}} [\mathsf{let}\,(z,y) = \mathsf{recv}\,y\,\mathsf{in}\,y\rangle : \mathsf{end} > !\mathsf{end}\,.\,\mathsf{end} \qquad \Gamma, x : \mathsf{end} \vdash^{\phi}_{\mathsf{C}} C : U}{\Gamma \vdash^{\phi}_{\mathsf{C}} (\boldsymbol{\nu} x[\mathsf{let}\,(z,y) = \mathsf{recv}\,y\,\mathsf{in}\,y\rangle y)C : U}$$

Note that such buffers will always deadlock: the message in the buffer can never be received.

Type Preservation Well-typed CGV terms and configurations satisfy protocol fidelity and communication safety. These properties follow from type preservation: typing is consistent across structural congruence and reduction. In both cases the proof is by induction on the derivation of the congruence and reduction, respectively; we include full proofs in the extended version of this paper [14].

Theorem 2.6. *If $\Gamma \vdash^{\phi}_{\mathsf{C}} C : T$ and $C \equiv_{\mathsf{C}} D$ or $C \longrightarrow_{\mathsf{C}} D$, then $\Gamma \vdash^{\phi}_{\mathsf{C}} D : T$.*

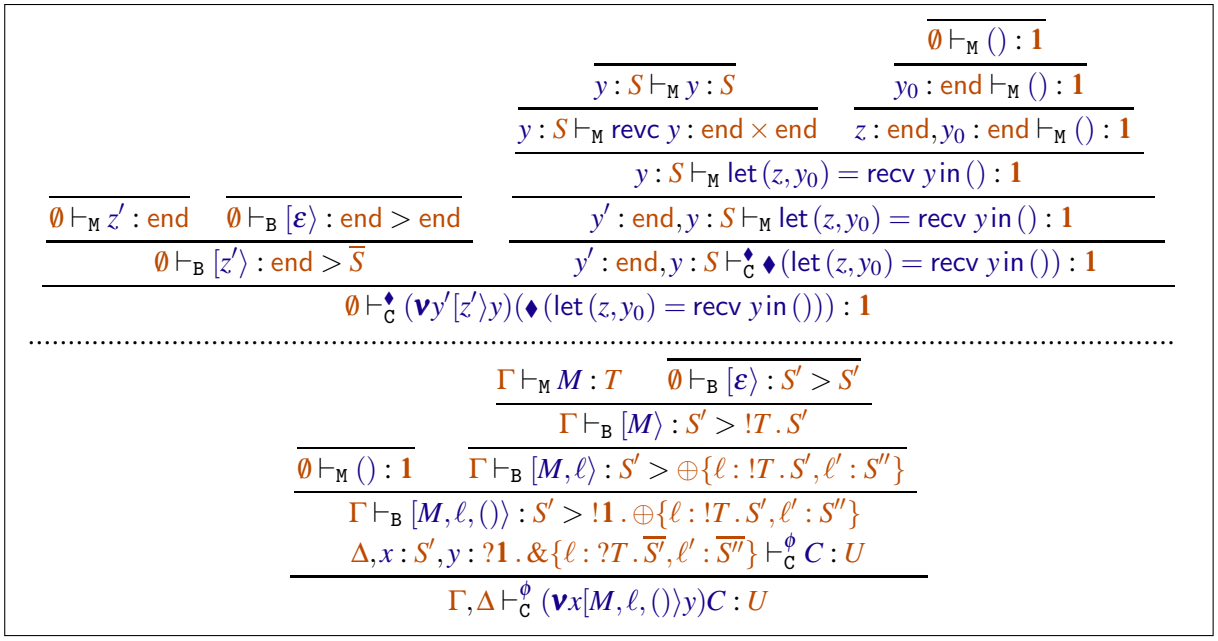


Figure 4: Derivation of configurations (cf. Example 2.5): (top) reduced from the initial one in Example 2.2 ($S = ?\mathsf{end}\,.\,\mathsf{end}$); (bottom) a buffer containing several messages.

3 APCP (Asynchronous Priority-based Classical Processes)

APCP [13] is a linear type system for π-calculus processes that communicate asynchronously (i.e., the output of messages is non-blocking) on connected channel endpoints. The type system assigns to endpoints types that specify two-party protocols, in the style of binary session types [15]. In APCP, well-typed processes may be cyclically connected: types rely on *priority* annotations, which enable cyclic connections while ruling out circular dependencies between sessions. Properties of well-typed APCP processes are *type preservation* (Theorem 3.4) and *deadlock-freedom* (Theorem 3.5).

Syntax and Semantics We write $x, y, z, \ldots$ to denote *endpoints* (or *names*), and write $\tilde{x}, \tilde{y}, \tilde{z}, \ldots$ to denote sequences of endpoints. Also, we write $i, j, k, \ldots$ to denote *labels* and $I, J, K, \ldots$ to denote sets of labels.

Figure 5 (top) gives the syntax and meaning of processes. In APCP, all endpoints are used strictly linearly: each endpoint can be used for exactly one communication only. However, we want to assign session types to endpoints, so we have to be able to implement sequences of communications. Therefore, each communication action carries an additional *continuation endpoint* to continue the session on.

The output action $x[y, z]$ sends a message endpoint y and a continuation endpoint z along x. The input prefix $x(y, z)\,.\,P$ blocks until a message and a continuation endpoint are received on x, binding y and z in P. The selection action $x[z] \triangleleft i$ sends a label i and a continuation endpoint z along x. The branching prefix $x(z) \triangleright \{i : P_i\}_{i \in I}$ blocks until it receives a label $i \in I$ and a continuation endpoint z on x, binding z in each P_i. Restriction $(\boldsymbol{\nu} xy)P$ binds x and y in P to form a channel for communication. The process $P \mid Q$ denotes parallel composition. The process $\mathbf{0}$ denotes inaction. The forwarder process $x \leftrightarrow y$ is a primitive copycat process that links together x and y.

Endpoints are free unless they are bound somehow. We write $\mathrm{fn}(P)$ for the set of free names of P.

Process syntax:

$$\begin{array}{llllll} P,Q ::= x[y,z] & \text{(output)} & \mid\ x(y,z)\,.\,P & \text{(input)} & & \\ \mid\ x[z] \triangleleft i & \text{(selection)} & \mid\ x(z) \triangleright \{i : P\}_{i\in I} & \text{(branching)} & \mid\ (\boldsymbol{\nu} xy)P & \text{(restriction)} \\ \mid\ P \mid Q & \text{(parallel)} & \mid\ \mathbf{0} & \text{(inaction)} & \mid\ x \leftrightarrow y & \text{(forwarder)} \end{array}$$

Structural congruence:

$$\begin{array}{lll} P \equiv P' \quad (\text{if } P \equiv_\alpha P') & P \mid Q \equiv Q \mid P & x \leftrightarrow y \equiv y \leftrightarrow x \\ P \mid (Q \mid R) \equiv (P \mid Q) \mid R & P \mid \mathbf{0} \equiv P & (\boldsymbol{\nu} xy)x \leftrightarrow y \equiv \mathbf{0} \\ P \mid (\boldsymbol{\nu} xy)Q \equiv (\boldsymbol{\nu} xy)(P \mid Q) \quad (\text{if } x,y \notin \text{fn}(P)) & (\boldsymbol{\nu} xy)\mathbf{0} \equiv \mathbf{0} & \\ (\boldsymbol{\nu} xy)(\boldsymbol{\nu} zw)P \equiv (\boldsymbol{\nu} zw)(\boldsymbol{\nu} xy)P & (\boldsymbol{\nu} xy)P \equiv (\boldsymbol{\nu} yx)P & \end{array}$$

Reduction:

$$\frac{z,y \neq x}{(\boldsymbol{\nu} yz)(x \leftrightarrow y \mid P) \longrightarrow P\{x/z\}}\ {\to_{\text{ID}}} \qquad \frac{}{(\boldsymbol{\nu} xy)(x[a,b] \mid y(v,z)\,.\,P) \longrightarrow P\{a/v,b/z\}}\ {\to_{\otimes\parr}}$$

$$\frac{j \in I}{(\boldsymbol{\nu} xy)(x[b] \triangleleft j \mid y(z) \triangleright \{i : P_i\}_{i\in I}) \longrightarrow P_j\{b/z\}}\ {\to_{\oplus\&}} \qquad \frac{P \equiv P' \quad P' \longrightarrow Q' \quad Q' \equiv Q}{P \longrightarrow Q}\ {\to_{\equiv}}$$

$$\frac{P \longrightarrow Q}{(\boldsymbol{\nu} xy)P \longrightarrow (\boldsymbol{\nu} xy)Q}\ {\to_{\nu}} \qquad \frac{P \longrightarrow Q}{P \mid R \longrightarrow Q \mid R}\ {\to_{\mid}}$$

Figure 5: Definition of APCP's process language.

Also, we write $P\{x/y\}$ to denote the capture-avoiding substitution of the free occurrences of y in P for x. We write sequences of substitutions $P\{x_1/y_1\}\ldots\{x_n/y_n\}$ as $P\{x_1/y_1,\ldots,x_n/y_n\}$.

The reduction relation for processes ($P \longrightarrow Q$) formalizes how complementary actions on connected endpoints may synchronize. As usual for π-calculi, reduction relies on *structural congruence* ($P \equiv Q$), which relates processes with minor syntactic differences; it is the smallest congruence on the syntax of processes (Fig. 5 (top)) satisfying the axioms in Figure 5 (center).

We define the reduction relation $P \longrightarrow Q$ by the axioms and closure rules in Figure 5 (bottom). Rule $\to_{\text{ID}}$ implements the forwarder as a substitution. Rule $\to_{\otimes\parr}$ synchronizes an output and an input on connected endpoints and substitutes the message and continuation endpoints. Rule $\to_{\oplus\&}$ synchronizes a selection and a branch: the received label determines the continuation process, substituting the continuation endpoint appropriately. Rules $\to_{\equiv}$, $\to_{\nu}$, and $\to_{\mid}$ close reduction under congruence, restriction, and parallel composition, respectively. We write $\longrightarrow^*$ for the reflexive, transitive closure of $\longrightarrow$.

The Type System APCP types processes by assigning binary session types to channel endpoints. Following Curry-Howard interpretations, we present session types as linear logic propositions (cf. Caires *et al.* [2] and Wadler [26]) extended with *priority* annotations. Intuitively, actions typed with lower priority cannot be blocked by those with higher priority.

We write $\mathsf{o}, \kappa, \pi, \rho, \ldots$ to denote priorities, and ω to denote the ultimate priority that is greater than all other priorities and cannot be increased further. That is, $\forall \mathsf{o} \in \mathbb{N}.\ \omega > \mathsf{o}$ and $\forall \mathsf{o} \in \mathbb{N}.\ \omega + \mathsf{o} = \omega$.

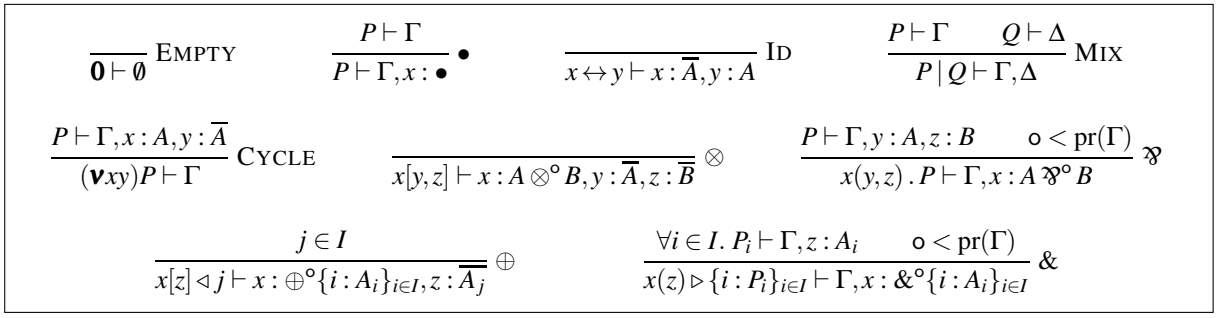


Figure 6: The typing rules of APCP.

Definition 3.1. *The following grammar defines the syntax of* session types A, B. *Let* $\circ \in \mathbb{N}$.

$$A, B ::= A \otimes^{\circ} B \quad \textit{(output)} \mid A \parr^{\circ} B \quad \textit{(input)} \mid \oplus^{\circ}\{i : A\}_{i \in I} \quad \textit{(select)} \mid \&^{\circ}\{i : A\}_{i \in I} \quad \textit{(branch)} \mid \bullet \quad \textit{(end)}$$

Note that type $\bullet$ does not require a priority.

Duality, the cornerstone of session types and linear logic, ensures that the two endpoints of a channel have matching actions. Furthermore, dual types must have matching priority annotations.

Definition 3.2. *The* dual *of session type A, denoted* $\overline{A}$*, is defined inductively as follows:*

$$\overline{A \otimes^{\circ} B} := \overline{A} \parr^{\circ} \overline{B} \qquad \overline{\oplus^{\circ}\{i : A_i\}_{i \in I}} := \&^{\circ}\{i : \overline{A_i}\}_{i \in I} \qquad \overline{\bullet} := \bullet$$
$$\overline{A \parr^{\circ} B} := \overline{A} \otimes^{\circ} \overline{B} \qquad \overline{\&^{\circ}\{i : A_i\}_{i \in I}} := \oplus^{\circ}\{i : \overline{A_i}\}_{i \in I}$$

The priority of a type is determined by the priority of the type's outermost connective:

Definition 3.3. *For session type A,* $\mathrm{pr}(A)$ *denotes its* priority*:*

$$\mathrm{pr}(A \otimes^{\circ} B) := \mathrm{pr}(A \parr^{\circ} B) := \mathrm{pr}(\oplus^{\circ}\{i : A_i\}_{i \in I}) := \mathrm{pr}(\&^{\circ}\{i : A_i\}_{i \in I}) := \circ \qquad \mathrm{pr}(\bullet) := \omega$$

The priority of $\bullet$ is ω: it denotes a "final" action of protocols without blocking behavior. Although associated with non-blocking behavior, $\otimes$ and $\oplus$ do have a non-constant priority: they are connected to $\parr$ and &, respectively, which denote blocking actions.

The typing rules of APCP ensure that actions with lower priority are not blocked by those with higher priority (cf. Dardha and Gay [3]). To this end, typing rules enforce the following laws:

1. An action with priority $\circ$ must be prefixed only by inputs and branches with priority strictly smaller than $\circ$—this law does not hold for output and selection, as they are not prefixes;
2. dual actions leading to a synchronization must have equal priorities (cf. Def. 3.2).

Judgments are of the form $P \vdash \Gamma$, where P is a process and Γ is a context that assigns types to endpoints $(x : A)$. A judgment $P \vdash \Gamma$ then means that P can be typed in accordance with the type assignments for names recorded in Γ. The context Γ obeys *exchange*: assignments may be silently reordered. Γ is *linear*, disallowing *weakening* (i.e., all assignments must be used) and *contraction* (i.e., assignments may not be duplicated). The empty context is written $\emptyset$. In writing $\Gamma, x : A$ we assume that $x \notin \mathrm{dom}(\Gamma)$. We write $\mathrm{pr}(\Gamma)$ to denote the least priority of all types in Γ (cf. Def. 3.3).

Figure 6 gives the typing rules. Rule EMPTY types an inactive process with no endpoints. Rule $\bullet$ silently removes a closed endpoint from the typing context. Rule ID types forwarding between endpoints

$$
\begin{array}{rl|rl}
[\![T \times U]\!] = ([\![T]\!] \parr \bullet) \otimes ([\![U]\!] \parr \bullet) & & [\![!T.S]\!] = (\overline{[\![T]\!]} \otimes \bullet) \parr [\![S]\!] & [\![\oplus\{i : T_i\}_{i\in I}]\!] = \&\{i : [\![T_i]\!]\}_{i\in I} \\
[\![T \multimap U]\!] = (\overline{[\![T]\!]} \otimes \bullet) \parr [\![U]\!] & & [\![?T.S]\!] = ([\![T]\!] \parr \bullet) \otimes [\![S]\!] & [\![\&\{i : T_i\}_{i\in I}]\!] = \oplus\{i : [\![T_i]\!]\}_{i\in I} \\
[\![\mathbf{1}]\!] = \bullet & & [\![\mathsf{end}]\!] = \bullet &
\end{array}
$$

Figure 7: Translation of CGV types into session types.

of dual type. Rule MIX types the parallel composition of two processes that do not share assignments on the same endpoints. Rule CYCLE types a restriction, where the two restricted endpoints must be of dual type. Rule $\otimes$ types an output action; this rule does not have premises to provide a continuation process, leaving the free endpoints to be bound to a continuation process using MIX and CYCLE. Similarly, Rule $\oplus$ types an unbound selection action. Priority checks are confined to Rules $\parr$ and &, which type input and branching prefixes, respectively. In both cases, the used endpoint's priority must be lower than the priorities of the other types in the continuation's typing context, thus enforcing Law 1 above.

Well-typed processes satisfy protocol fidelity, communication safety, and deadlock-freedom. The first two properties follow from *type preservation*. Here we only state these results; see [13] for details.

Theorem 3.4 (Type Preservation)**.** *If $P \vdash \Gamma$ and $P \equiv Q$ or $P \longrightarrow Q$, then $Q \vdash \Gamma$.*

Theorem 3.5 (Deadlock-freedom)**.** *If $P \vdash \emptyset$, then either $P \equiv \mathbf{0}$ or $P \longrightarrow Q$ for some Q.*

4 Translating CGV into APCP

4.1 The Translation

In this section, we translate CGV into APCP. We translate entire typing derivations, following, e.g., Wadler [26]. Given the structure of CGV and its type system, the translation is defined in parts: for (run-time) terms, for configurations, and for buffers. The translation is defined on well-typed configurations which may be deadlocked, so our translation does not consider priority requirements. As we will see, typability in APCP will enable us to identify deadlock-free configurations in CGV (cf. Sec. 4.3).

The translation is informed by the semantics of CGV. It is crucial that subterms may only reduce when they occur in reduction contexts. For example, M_1 and M_2 may not reduce if they appear in a pair (M_1, M_2). The translation must thus ensure that subterms are blocked when they do not occur in reduction contexts. Translations such as Wadler's hinge on blocking outputs and inputs; for example, the pair (M_1, M_2) is translated as an output that blocks the translations of M_1 and M_2. However, outputs in APCP are non-blocking and so we use additional inputs to disable the reduction of subterms. For example, the translation of (M_1, M_2) adds extra inputs to block the translations of M_1 and M_2.

Figure 7 gives the translation of CGV types into APCP types ($[\![T]\!]$), which already captures the operation of the translation: our translation is similar to the one by Wadler, but includes the aforementioned additional inputs. It may seem odd that this translation dualizes CGV session types (e.g., an output '!' becomes an input '$\parr$'). To understand this, consider that a variable x typed $!T.S$ represents access to a session which expects the user to send a term of type T and continue as S, but not the output itself. Hence, to translate an output on x into APCP, we need to connect the translation of x to an actual output. Since this actual output would be typed with $\otimes$, this means that the translation of x would need to be dually typed, i.e., typed with $\parr$. A more technical explanation is that the translation moves from two-sided CGV judgments to one-sided APCP judgments, which requires dualization (see, e.g., [9, 12]).

Importantly, the translation preserves duality of session types (by induction on their structure):

Proposition 4.1. *Given a CGV session type S, $\overline{[\![S]\!]} = [\![\overline{S}]\!]$.*

We extend the translation of types to typing environments, defined as expected. Similarly, we extend duality to typing environments: $\overline{\Gamma}$ denotes Γ with each type dualized. In this section, we give simplified presentations of the translations, showing only the conclusions of the source and target derivations; we include the translations with full derivations in the extended version of this paper [14].

A remark on notation. Some translated terms include annotated restrictions $(\overset{\leftrightarrow}{\boldsymbol{\nu}}xy)$. These so-called *forwarder-enabled* restrictions can be ignored in this subsection, but will be useful later when proving soundness (one of the correctness properties of the translation; cf. Section 4.2).

We define the translation of (the typing rules of) terms. Since a term has a provided type, the translation takes as a parameter a name on which the translation provides this type. Figure 8 gives the translation of terms, denoted $[\![\Gamma \vdash_{\mathtt{M}} \mathbb{M} : T]\!]z$, where the type T is provided on z. By abuse of notation, we write $[\![\mathbb{M}]\!]z$ to denote the process translation of the term $\mathbb{M}$, and similary for configurations and buffers. Notice the aforementioned additional inputs to block behavior of subterms in rules such as Rule T-Pair. Before moving to buffers and configurations, we illustrate the translation of terms by an example:

Example 4.2. *Consider the following subterm from Example 2.3: $(\lambda z\,.\,\mathsf{send}\ ((),z))\ y$. We gradually discuss how this term translates to APCP, and how the translation is set up to mimick the term's behavior.*

$$[\![(\lambda z\,.\,\mathsf{send}\ ((),z))\ y]\!]q = (\boldsymbol{\nu} ab)\big([\![\lambda z\,.\,\mathsf{send}\ ((),z)]\!]a \mid (\boldsymbol{\nu} cd)(b[c,q] \mid d(e,\_)\,.\,[\![y]\!]e)\big)$$

The function application translates the function on a, which is connected to b. The output on b serves to activate the function, which will subsequently activate the functions parameter ($[\![y]\!]e = y \leftrightarrow e$) by means of an output that will be received on d.

$$[\![\lambda z\,.\,\mathsf{send}\ ((),z)]\!]a = a(f,g)\,.\,(\overset{\leftrightarrow}{\boldsymbol{\nu}} hz)((\boldsymbol{\nu}\_\_)f[h,\_] \mid [\![\mathsf{send}\ ((),z)]\!]g)$$

The translation of the function is indeed blocked until it receives on a. It then outputs on f to activate the function's parameter (which receives on d), while the function's body appears in parallel.

$$[\![\mathsf{send}\ ((),z)]\!]g = (\boldsymbol{\nu} kl)\Big([\![((),z)]\!]k \mid l(m,n)\,.\,(\boldsymbol{\nu} op)\big((\boldsymbol{\nu}\_\_)n[o,\_] \mid (\boldsymbol{\nu} rs)(p[m,r] \mid s \leftrightarrow g)\big)\Big)$$

The translation of the send *primitive connects the translation of the pair $((),z)$ on k to an input on l, receiving endpoints for the output term (m) and the output endpoint (n). Once activated by the input on l, the term representing the output endpoint is activated by means of an output on n. In parallel, the actual output (on p) sends the endpoint of the output term (m) and a fresh endpoint (r) representing the continuation channel after the message has been placed in a buffer (the forwarder $s \leftrightarrow g$).*

$$[\![((),z)]\!]k = (\boldsymbol{\nu} tu)(\boldsymbol{\nu} vw)(k[t,v] \mid u(a',\_)\,.\,[\![()]\!]a' \mid w(b',\_)\,.\,[\![z]\!]b')$$

The translation of the pair outputs on k two endpoints for the two terms it contains (to be received by whatever intends to use the pair in the context, e.g., the send *primitive on l). The translations of the two terms inside the pair ($[\![()]\!]a' = \mathbf{0}$ and $[\![z]\!]b' = z \leftrightarrow b'$) are both guarded by an input, preventing the terms from reducing until the context explicitly activates them by means of outputs.*

Analogously to the reductions from Example 2.3—$(\lambda z\,.\,\mathsf{send}\ ((),z))\ y \longrightarrow^3_{\mathtt{M}} \mathsf{send}'((),y)$—we have

$$[\![(\lambda z\,.\,\mathsf{send}\ ((),z))\ y]\!]q \longrightarrow^5 [\![\mathsf{send}'((),y)]\!]q.$$

T-VAR $[\![x : T \vdash_{\mathtt{M}} x : T]\!]z = x \leftrightarrow z \vdash x : \overline{[\![T]\!]}, z : [\![T]\!]$ T-UNIT $[\![\emptyset \vdash_{\mathtt{M}} () : \mathbf{1}]\!]z = \mathbf{0} \vdash z : \bullet$

T-ABS $[\![\Gamma \vdash_{\mathtt{M}} \lambda x.M : T \multimap U]\!]z = z(a,b).(\overleftrightarrow{\boldsymbol{\nu}}cx)((\boldsymbol{\nu}ef)a[c,e] \mid [\![M]\!]b) \vdash \overline{[\![\Gamma]\!]}, z : (\overline{[\![T]\!]} \otimes \bullet) \parr [\![U]\!]$

T-APP $[\![\Gamma,\Delta \vdash_{\mathtt{M}} M\,N : U]\!]z = (\boldsymbol{\nu}ab)([\![M]\!]a \mid (\boldsymbol{\nu}cd)(b[c,z] \mid d(e,f).[\![N]\!]e)) \vdash \overline{[\![\Gamma]\!]}, \overline{[\![\Delta]\!]}, z : [\![U]\!]$

T-PAIR $\left[\!\!\left[\begin{array}{r}\Gamma,\Delta \vdash_{\mathtt{M}} (M,N)\\ : T \times U\end{array}\right]\!\!\right]z = \begin{array}{l}(\boldsymbol{\nu}ab)(\boldsymbol{\nu}cd)(z[a,c] \mid b(e,f).[\![M]\!]e \mid d(g,h).[\![N]\!]g)\\ \quad \vdash \overline{[\![\Gamma]\!]}, \overline{[\![\Delta]\!]}, z : ([\![T]\!] \parr \bullet) \otimes ([\![U]\!] \parr \bullet)\end{array}$

T-SPLIT $\left[\!\!\left[\begin{array}{r}\Gamma,\Delta \vdash_{\mathtt{M}} \mathsf{let}\,(x,y)\\ = M\,\mathsf{in}\,N : U\end{array}\right]\!\!\right]z = \begin{array}{l}(\boldsymbol{\nu}ab)([\![M]\!]a \mid b(c,d).(\overleftrightarrow{\boldsymbol{\nu}}ex)(\overleftrightarrow{\boldsymbol{\nu}}fy)(\\ \quad (\boldsymbol{\nu}gh)c[e,g] \mid (\boldsymbol{\nu}kl)d[f,k] \mid [\![N]\!]z)) \vdash \overline{[\![\Gamma]\!]}, \overline{[\![\Delta]\!]}, z : [\![U]\!]\end{array}$

T-NEW $[\![\emptyset \vdash_{\mathtt{M}} \mathsf{new} : S \times \overline{S}]\!]z = \begin{array}{l}(\boldsymbol{\nu}ab)((\boldsymbol{\nu}cd)a[c,d] \mid b(e,f).(\boldsymbol{\nu}xy)[\![(x,y)]\!]z)\\ \quad \vdash z : ([\![S]\!] \parr \bullet) \otimes ([\![\overline{S}]\!] \parr \bullet)\end{array}$

T-SPAWN $[\![\Gamma \vdash_{\mathtt{M}} \mathsf{spawn}\,M : T]\!]z = (\boldsymbol{\nu}ab)([\![M]\!]a \mid b(c,d).((\boldsymbol{\nu}ef)c[e,f] \mid (\boldsymbol{\nu}gh)d[z,g])) \vdash \overline{[\![\Gamma]\!]}, z : [\![T]\!]$

T-ENDL $[\![\Gamma, x : \mathsf{end} \vdash_{\mathtt{M}} M : T]\!]z = [\![M]\!]z \vdash \overline{[\![\Gamma]\!]}, x : \bullet, z : [\![T]\!]$ T-ENDR $[\![\emptyset \vdash_{\mathtt{M}} x : \mathsf{end}]\!]z = \mathbf{0} \vdash z : \bullet$

T-SEND $[\![\Gamma \vdash_{\mathtt{M}} \mathsf{send}\,M : S]\!]z = \begin{array}{l}(\boldsymbol{\nu}ab)([\![M]\!]a \mid b(c,d).(\boldsymbol{\nu}ef)((\boldsymbol{\nu}gh)d[e,g]\\ \quad \mid (\boldsymbol{\nu}kl)(f[c,k] \mid l \leftrightarrow z))) \vdash \overline{[\![\Gamma]\!]}, z : [\![S]\!]\end{array}$

T-RECV $[\![\Gamma \vdash_{\mathtt{M}} \mathsf{recv}\,M : T \times S]\!]z = \begin{array}{l}(\boldsymbol{\nu}ab)([\![M]\!]a \mid b(c,d).(\boldsymbol{\nu}ef)(z[c,e] \mid f(g,h).d \leftrightarrow g))\\ \quad \vdash \overline{[\![\Gamma]\!]}, z : ([\![T]\!] \parr \bullet) \otimes ([\![S]\!] \parr \bullet)\end{array}$

T-SELECT $[\![\Gamma \vdash_{\mathtt{M}} \mathsf{select}\,j\,M : T_j]\!]z = (\boldsymbol{\nu}ab)([\![M]\!]a \mid (\boldsymbol{\nu}cd)(b[c] \triangleleft j \mid d \leftrightarrow z)) \vdash \overline{[\![\Gamma]\!]}, z : [\![T_j]\!]$

T-CASE $\left[\!\!\left[\begin{array}{r}\Gamma,\Delta \vdash_{\mathtt{M}} \mathsf{case}\,M\\ \mathsf{of}\,\{i : N_i\}_{i\in I} : U\end{array}\right]\!\!\right]z = (\boldsymbol{\nu}ab)([\![M]\!]a \mid b(c) \triangleright \{i : [\![N_i\ c]\!]z\}_{i\in I}) \vdash \overline{[\![\Gamma]\!]}, \overline{[\![\Delta]\!]}, z : [\![U]\!]$

T-SUB $[\![\Gamma,\Delta \vdash_{\mathtt{M}} \mathbb{M}\{\!|\mathbb{N}/x|\!\} : U]\!]z = (\overleftrightarrow{\boldsymbol{\nu}}xa)([\![\mathbb{M}]\!]z \mid [\![\mathbb{N}]\!]a) \vdash \overline{[\![\Gamma]\!]}, \overline{[\![\Delta]\!]}, z : [\![U]\!]$

T-SEND' $\left[\!\!\left[\begin{array}{r}\Gamma,\Delta \vdash_{\mathtt{M}} \mathsf{send}'\\ (M,\mathbb{N}) : S\end{array}\right]\!\!\right]z = \begin{array}{l}(\boldsymbol{\nu}ab)(a(c,d).[\![M]\!]c \mid (\boldsymbol{\nu}ef)([\![\mathbb{N}]\!]e \mid (\boldsymbol{\nu}gh)(f[b,g] \mid h \leftrightarrow z)))\\ \quad \vdash \overline{[\![\Gamma]\!]}, \overline{[\![\Delta]\!]}, z : [\![S]\!]\end{array}$

Figure 8: Translation of (runtime) term typing rules. See [14] for typing derivations.

Figure 9 (top) gives the translation of configurations, denoted $[\![\Gamma \vdash^{\phi}_{\mathtt{C}} C : T]\!]z$. We omit the translation of Rule T-PARR. Noteworthy are the translations of buffered restrictions: the translation of $(\boldsymbol{\nu}x[\vec{m}\rangle y)C$ relies on the translation of $[\vec{m}\rangle$, which is given the translation of C as its continuation.

The translation of buffers requires care: each message in the buffer is translated as an output in APCP, where the output of the following messages is on the former output's continuation endpoint. Once there are no more messages in the buffer, the translation uses a typed APCP process—a parameter of the translation—to provide the behavior of the continuation of the lastmost output. The translation has no requirements for the continuation process and its typing, except for the type of the buffer's endpoint. With this in mind, Figure 9 (bottom) gives the translation of the typing rules of buffers, denoted $[\![\Gamma \vdash_{\mathtt{B}} [\vec{m}\rangle : S' > S]\!]_x^{P \vdash^* \Lambda, x:\overline{[\![S']\!]}}$, where x is the endpoint on which the buffer outputs, and P is the continuation of the buffer's last message. Note that we never use the typing rules for buffers by themselves: they always accompany the typing of endpoint restriction, of which the translation properly instantiates the continuation process.

Because CGV configurations may deadlock, the type preservation result of our translation holds up

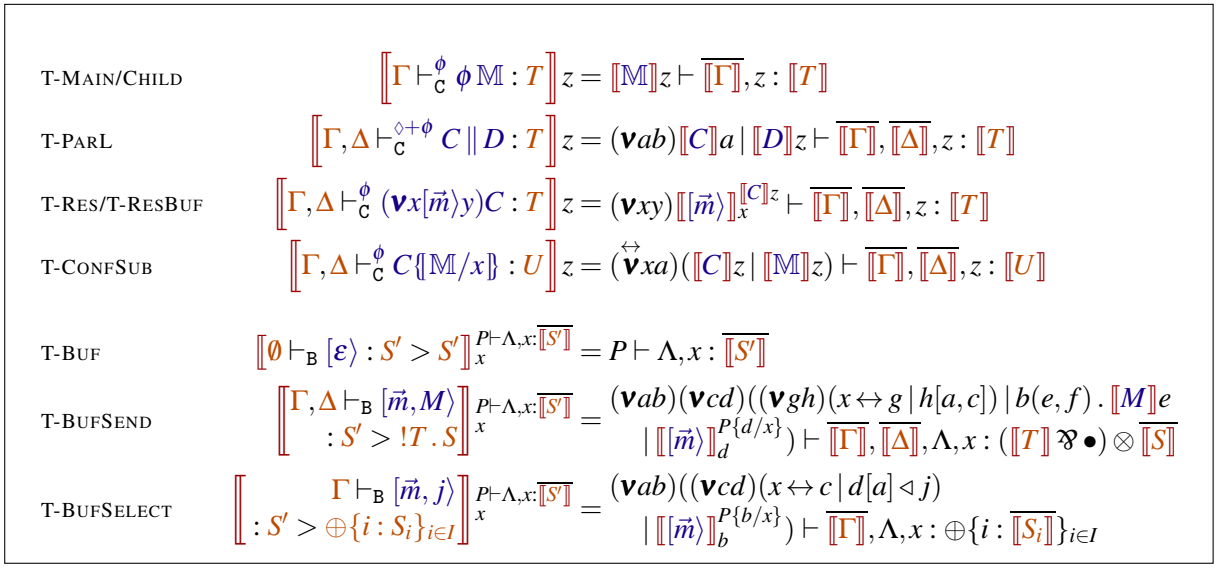


Figure 9: Translation of configuration and buffer typing rules. See [14] for typing derivations.

to priority requirements. To formalize this, we have the following definition:

Definition 4.3. *Let P be a process. We write $P \vdash^* \Gamma$ to denote that P is well-typed according to the typing rules in Figure 6 where Rules $\parr$ and & are modified by erasing priority checks.*

Hence, if $P \vdash \Gamma$ then $P \vdash^* \Gamma$ but the converse does not hold. Our translation correctly preserves the typing of terms, configurations, and buffers:

Theorem 4.4 (Type Preservation for the Translation)**.**

- $\llbracket \Gamma \vdash_{\mathtt{M}} \mathbb{M} : T \rrbracket z = \llbracket \mathbb{M} \rrbracket z \vdash^* \overline{\llbracket \Gamma \rrbracket}, z : \llbracket T \rrbracket$
- $\llbracket \Gamma \vdash^{\phi}_{\mathtt{C}} C : T \rrbracket z = \llbracket C \rrbracket z \vdash^* \overline{\llbracket \Gamma \rrbracket}, z : \llbracket T \rrbracket$
- $\llbracket \Gamma \vdash_{\mathtt{B}} [\vec{m}\rangle : S' > S \rrbracket_x^{P \vdash^* \Lambda, x : \overline{\llbracket S' \rrbracket}} = \llbracket [\vec{m}\rangle \rrbracket_x^P \vdash^* \overline{\llbracket \Gamma \rrbracket}, \Lambda, x : \llbracket S \rrbracket$

Example 4.5. *Consider again the configuration $(\boldsymbol{\nu} x[M, \ell, ()\rangle y)C$. We illustrate the translation of buffers into APCP by giving the translation of this configuration (writing $\langle x\rangle[a,b]$ to denote the forwarded output $(\boldsymbol{\nu} cd)(x \leftrightarrow c \mid d[a,b])$):*

$$\llbracket (\boldsymbol{\nu} x[M, \ell, ()\rangle y)C \rrbracket z = (\boldsymbol{\nu} xy)\llbracket [M, \ell, ()\rangle \rrbracket_x^{\llbracket C \rrbracket z} = (\boldsymbol{\nu} xy)\ (\boldsymbol{\nu} ab)(\boldsymbol{\nu} cx')(\langle x\rangle[a,c] \mid b(d,\_)\,.\,\mathbf{0} \mid (\boldsymbol{\nu} ex'')(\langle x'\rangle[e] \triangleleft \ell \mid (\boldsymbol{\nu} fg)(\boldsymbol{\nu} hx''')(\langle x''\rangle[f,h] \mid g(k,\_)\,.\,\llbracket M \rrbracket k \mid \llbracket C \rrbracket z\{x'''/x\})))$$

Notice how the (forwarded) outputs are sequenced by continuation endpoints, and how the translation of C uses the last continuation endpoint x''' to interact with the buffer.

4.2 Operational Correctness

Following Gorla [10], we focus on *operational correspondence*: a translated configuration can reproduce all of the source configuration's reductions (completeness; Theorem 4.6), and any of the translated configuration's reductions can be traced back to reductions of the source configuration (soundness; Theorem 4.7). With the soundness result, our translation is stronger than related prior translations [20, 24, 19].

Our completeness result states that the reductions of a well-typed configuration can be mimicked by its translation in zero or more steps.

Theorem 4.6 (Completeness). *Given* $\Gamma \vdash^{\phi}_{\mathsf{C}} C : T$, *if* $C \longrightarrow_{\mathsf{C}} D$, *then* $[\![C]\!]z \longrightarrow^{*} [\![D]\!]z$.

Proof (Sketch). By induction on the derivation of the configuration's reduction. In each case, we infer the shape of the configuration from the reduction and well-typedness. We then consider the translation of the configuration, and show that the resulting process reduces in zero or more steps to the translation of the reduced configuration. See the extended version of this paper [14] for a full proof. □

Soundness states that any sequence of reductions from the translation of a well-typed configuration eventually leads to the translation of another configuration, which the initial configuration also reduces to. Asynchrony in APCP requires us to be careful, specifically concerning the semantics of variables in CGV. Variables can only cause reductions under specific circumstances. On the other hand, variables translate to forwarders in APCP, which reduce as soon as they are bound by restriction. This semantics for forwarders turns out to be too eager for soundness. As a result, soundness only holds for an alternative, so-called *lazy semantics* for APCP, denoted $\longrightarrow_{\mathsf{L}}$, in which forwarders may only cause reductions under specific circumstances. It is here that the forwarder-enabled restrictions $(\overleftrightarrow{\boldsymbol{\nu}}xy)$ anticipated in Section 4.1 come into play. As we will see in Section 4.3, this alternative semantics does not prevent us from identifying a class of deadlock-free CGV configurations through the translation into APCP. Due to space limitations, the definitions of the lazy semantics only appears in the extended version of this paper [14].

Theorem 4.7 (Soundness). *Given* $\Gamma \vdash^{\phi}_{\mathsf{C}} C : T$, *if* $[\![C]\!]z \longrightarrow^{*}_{\mathsf{L}} Q$, *then* $C \longrightarrow^{*}_{\mathsf{C}} D$ *and* $Q \longrightarrow^{*}_{\mathsf{L}} [\![D]\!]z$ *for some* D.

Proof (Sketch). By induction on the structure of C. In each case, we additionally apply induction on the number k of steps $[\![C]\!]z \longrightarrow^{k}_{\mathsf{L}} Q$. We then consider which reductions might occur from $[\![C]\!]z$ to Q. Considering the structure of C, we then isolate a sequence of k' possible steps, such that $[\![C]\!]z \longrightarrow^{k'}_{\mathsf{L}} [\![D']\!]z$ for some D' where $C \longrightarrow_{\mathsf{C}} D'$. Since $[\![D']\!]z \longrightarrow^{k-k'}_{\mathsf{L}} Q$, it then follows from the induction hypothesis that there exists D such that $D' \longrightarrow^{*}_{\mathsf{C}} D$ and $[\![D']\!]z \longrightarrow^{*}_{\mathsf{L}} [\![D]\!]z$.

Key here is the independence of reductions in APCP: if two or more reductions are enabled from a (well-typed) process, they must originate from independent parts of the process, and so they do not interfere with each other. This essentially means that the order in which independent reductions occur does not affect the resulting process. Hence, we can pick "desirable" sequences of reductions, postponing other possible reductions. See the extended version of this paper [14] for a full proof of soundness. □

From the proof above we can deduce that if the translation takes at least one step, then so does the source:

Corollary 4.8. *Given* $\Gamma \vdash^{\phi}_{\mathsf{C}} C : T$, *if* $[\![C]\!]z \longrightarrow^{+}_{\mathsf{L}} Q$, *then* $C \longrightarrow^{+}_{\mathsf{C}} D$ *and* $Q \longrightarrow^{*}_{\mathsf{L}} [\![D]\!]z$ *for some* D.

4.3 Transferring Deadlock-freedom from APCP to CGV

In APCP, well-typed processes typable under empty contexts ($P \vdash \emptyset$) are deadlock-free. By appealing to the operational correctness of our translation, we transfer this result to CGV configurations. Each deadlock-free configuration in CGV obtained via transference satisfies two requirements:

- The configuration is typable $\emptyset \vdash^{\blacklozenge}_{\mathsf{C}} C : \mathbf{1}$: it needs no external resources and has no external behavior.
- The typed translation of the configuration satisfies APCP's priority requirements: it is well-typed under '$\vdash$', not only under '$\vdash^{*}$' (cf. Def. 4.3).

We rely on soundness (Theorem 4.7) to transfer deadlock-freedom to configurations. However, APCP's deadlock-freedom (Theorem 3.5) considers standard semantics ($\longrightarrow$), whereas soundness considers the lazy semantics ($\longrightarrow_{\mathsf{L}}$). Therefore, we first must show that if the translation of a configuration

	λ^{sess} [8]	GV [26]	EGV [7]	PGV [17, 18]	**CGV (this paper)**
Communication	Asynch.	Synch.	Asynch.	Synch.	Asynch.
Cyclic Topologies	Yes	No	No	Yes	Yes
Deadlock-Freedom	No	Yes (typing)	Yes (typing)	Yes (typing)	Yes (via APCP)

Table 1: The features of CGV compared to its predecessors.

satisfying the requirements above reduces under $\longrightarrow$, it also reduces under $\longrightarrow_{\mathtt{L}}$; this is Theorem 4.9 below. The deadlock-freedom of these configurations (Theorem 4.10) then follows from Theorems 3.5 and 4.9. See the extended version of this paper [14] for detailed proofs of these results.

Theorem 4.9. *Given* $\emptyset \vdash^{\blacklozenge}_{\mathtt{C}} C : \mathbf{1}$, *if* $[\![C]\!]z \vdash \Gamma$ *for some* Γ *and* $[\![C]\!]z \longrightarrow Q$, *then* $[\![C]\!]z \longrightarrow_{\mathtt{L}} Q'$, *for some* Q'.

Proof (Sketch). By inspecting the derivation of $[\![C]\!]z \longrightarrow Q$. If the reduction is not derived from $\rightarrow_{\mathrm{ID}}$, it can be directly replicated under $\longrightarrow_{\mathtt{L}}$. Otherwise, we analyze the possible shapes of C and show that a different reduction under $\longrightarrow_{\mathtt{L}}$ is possible. □

Theorem 4.10 (Deadlock-freedom for CGV)**.** *Given* $\emptyset \vdash^{\blacklozenge}_{\mathtt{C}} C : \mathbf{1}$, *if* $[\![C]\!]z \vdash \Gamma$ *for some* Γ, *then* $C \equiv \blacklozenge()$ *or* $C \longrightarrow_{\mathtt{C}} D$ *for some* D.

Proof (Sketch). By assumption and Theorem 4.4, $[\![C]\!]z \vdash z : \bullet$. Then $(\boldsymbol{\nu} z\_)[\![C]\!]z \vdash \emptyset$. By Theorem 3.5, (i) $(\boldsymbol{\nu} z\_)[\![C]\!]z \equiv \mathbf{0}$ or (ii) $(\boldsymbol{\nu} z\_)[\![C]\!]z \longrightarrow Q$ for some Q. In case (i) it follows from the well-typedness and translation of C that $C \equiv_{\mathtt{C}} \blacklozenge()$. In case (ii) we deduce that the reduction of $(\boldsymbol{\nu} z\_)[\![C]\!]z$ cannot involve the endpoint z. Hence, $[\![C]\!]z \longrightarrow Q_0$ for some Q_0. By Theorem 4.9, then $[\![C]\!]z \longrightarrow_{\mathtt{L}} Q'$ for some Q'. Then, by Corollary 4.8, there exists D' such that $C \longrightarrow^{+}_{\mathtt{C}} D'$. Hence, $C \longrightarrow_{\mathtt{C}} D$ for some D, proving the thesis. □

As an example, using Theorem 4.10 we can show that C_1 from Example 2.2 is deadlock-free; see [14].

5 Conclusion

We have presented CGV, a new functional language with asynchronous session-typed communication. As illustrated in Section 1, CGV is strictly more expressive than its predecessors, thanks to a highly asynchronous semantics (compared to GV and PGV), its support for cyclic thread configurations (compared to EGV), and the ability to send whole terms and not just values (compared to all the mentioned calculi). Table 1 summarizes the features of CGV compared to its predecessors.

An operationally correct translation into APCP solidifies the design of CGV, and enables identifying a class of deadlock-free CGV programs. Interestingly, the asynchronous semantics of CGV is reminiscent of *future/promise* programming paradigms (see, e.g., [11, 21, 25]), which have been little studied in the context of session-typed communication.

The alternative to establishing deadlock-freedom in CGV via translation into APCP would be to enhance CGV's type system with priorities (in the spirit of, e.g., work by Padovani and Novara [23]). Another useful addition concerns recursion / recursive types. We leave these extensions to future work.

Acknowledgments Thanks to Simon Fowler and the anonymous reviewers for their helpful feedback. We gratefully acknowledge the support of the Dutch Research Council (NWO) under project No. 016.Vidi.189.046 (Unifying Correctness for Communicating Software).

References

[1] Luís Caires & Frank Pfenning (2010): *Session Types as Intuitionistic Linear Propositions*. In Paul Gastin & François Laroussinie, editors: *CONCUR 2010 - Concurrency Theory*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 222–236, doi:10.1007/978-3-642-15375-4_16.

[2] Luís Caires, Frank Pfenning & Bernardo Toninho (2016): *Linear Logic Propositions as Session Types*. *Mathematical Structures in Computer Science* 26(3), pp. 367–423, doi:10.1017/S0960129514000218.

[3] Ornela Dardha & Simon J. Gay (2018): *A New Linear Logic for Deadlock-Free Session-Typed Processes*. In Christel Baier & Ugo Dal Lago, editors: *Foundations of Software Science and Computation Structures*, Lecture Notes in Computer Science, Springer International Publishing, pp. 91–109, doi:10.1007/978-3-319-89366-2_5.

[4] Ornela Dardha & Jorge A. Pérez (2015): *Comparing Deadlock-Free Session Typed Processes*. *Electronic Proceedings in Theoretical Computer Science* 190, pp. 1–15, doi:10.4204/EPTCS.190.1. arXiv:1508.06707.

[5] Ornela Dardha & Jorge A. Pérez (2022): *Comparing Type Systems for Deadlock Freedom*. *Journal of Logical and Algebraic Methods in Programming* 124, p. 100717, doi:10.1016/j.jlamp.2021.100717.

[6] Simon Fowler (2019): *Typed Concurrent Functional Programming with Channels, Actors, and Sessions*. Ph.D. thesis, University of Edinburgh.

[7] Simon Fowler, Sam Lindley, J. Garrett Morris & Sára Decova (2019): *Exceptional Asynchronous Session Types: Session Types without Tiers*. *Proceedings of the ACM on Programming Languages*, doi:10.1145/3290341.

[8] Simon J. Gay & Vasco T. Vasconcelos (2010): *Linear Type Theory for Asynchronous Session Types*. *Journal of Functional Programming* 20(1), pp. 19–50, doi:10.1017/S0956796809990268.

[9] Jean-Yves Girard (1993): *On the Unity of Logic*. *Annals of Pure and Applied Logic* 59(3), pp. 201–217, doi:10.1016/0168-0072(93)90093-S.

[10] Daniele Gorla (2010): *Towards a Unified Approach to Encodability and Separation Results for Process Calculi*. *Information and Computation* 208(9), pp. 1031–1053, doi:10.1016/j.ic.2010.05.002.

[11] Robert H. Halstead (1985): *MULTILISP: A Language for Concurrent Symbolic Computation*. *ACM Transactions on Programming Languages and Systems* 7(4), pp. 501–538, doi:10.1145/4472.4478.

[12] Bas van den Heuvel & Jorge A. Pérez (2020): *Session Type Systems Based on Linear Logic: Classical versus Intuitionistic*. *Electronic Proceedings in Theoretical Computer Science* 314, pp. 1–11, doi:10.4204/EPTCS.314.1. arXiv:2004.01320.

[13] Bas van den Heuvel & Jorge A. Pérez (2021): *Deadlock Freedom for Asynchronous and Cyclic Process Networks (Extended Version)*. *arXiv:2111.13091 [cs]*. arXiv:2111.13091. A short version appears in the Proceedings of ICE'21: arXiv:2110.00146.

[14] Bas van den Heuvel & Jorge A. Pérez (2022): *Asynchronous Functional Sessions: Cyclic and Concurrent (Extended Version)*, doi:10.48550/arXiv.2208.07644. arXiv:2208.07644.

[15] Kohei Honda (1993): *Types for Dyadic Interaction*. In Eike Best, editor: *CONCUR'93*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 509–523, doi:10.1007/3-540-57208-2_35.

[16] Naoki Kobayashi (2006): *A New Type System for Deadlock-Free Processes*. In Christel Baier & Holger Hermanns, editors: *CONCUR 2006 – Concurrency Theory*, Lecture Notes in Computer Science, Springer Berlin Heidelberg, pp. 233–247, doi:10.1007/11817949_16.

[17] Wen Kokke & Ornela Dardha (2021): *Prioritise the Best Variation*. In Kirstin Peters & Tim A. C. Willemse, editors: *Formal Techniques for Distributed Objects, Components, and Systems*, Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 100–119, doi:10.1007/978-3-030-78089-0_6.

[18] Wen Kokke & Ornela Dardha (2021): *Prioritise the Best Variation*, doi:10.48550/arXiv.2103.14466. arXiv:2103.14466.

[19] Sam Lindley & J. Garrett Morris (2015): *A Semantics for Propositions as Sessions*. In Jan Vitek, editor: *Programming Languages and Systems*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 560–584, doi:10.1007/978-3-662-46669-8_23.

[20] Robin Milner (1989): *Communication and Concurrency*. Prentice Hall International Series in Computer Science, Prentice Hall, New York, USA.

[21] Gerald K. Ostheimer & Antony J. T. Davie (1993): *Pi-Calculus Characterizations of Some Practical Lambda-Calculus Reduction Strategies*. Technical Report CS/93/14, Department of Computing Sciences, University of St Andrews.

[22] Luca Padovani (2014): *Deadlock and Lock Freedom in the Linear π-Calculus*. In: *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, ACM, New York, NY, USA, pp. 72:1–72:10, doi:10.1145/2603088.2603116.

[23] Luca Padovani & Luca Novara (2015): *Types for Deadlock-Free Higher-Order Programs*. In Susanne Graf & Mahesh Viswanathan, editors: *Formal Techniques for Distributed Objects, Components, and Systems*, Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 3–18, doi:10.1007/978-3-319-19195-9_1.

[24] Davide Sangiorgi & David Walker (2003): *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press.

[25] G. Tremblay & B. Malenfant (2000): *Lenient Evaluation and Parallelism*. *Computer Languages* 26(1), pp. 27–41, doi:10.1016/S0096-0551(01)00007-8.

[26] Philip Wadler (2012): *Propositions As Sessions*. In: *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, ACM, New York, NY, USA, pp. 273–286, doi:10.1145/2364527.2364568.

Encodability and Separation for a Reflective Higher-Order Calculus*

Stian Lybech
Reykjavík University
Reykjavík, Iceland
stian21@ru.is

The ρ-calculus (Reflective Higher-Order Calculus) of Meredith and Radestock is a π-calculus-like language with some unusual features, notably, structured names, runtime generation of free names, and the lack of an operator for scoping visibility of names. These features pose some interesting difficulties for proofs of encodability and separation results. We describe two errors in a previously published attempt to encode the π-calculus in the ρ-calculus by Meredith and Radestock. Then we give a new encoding and prove its correctness, using a set of encodability criteria close to those proposed by Gorla, and discuss the adaptations necessary to work with a calculus with runtime generation of structured names. Lastly we prove a separation result, showing that the ρ-calculus cannot be encoded in the π-calculus.

1 Introduction

Process calculi are formalisms for modelling and reasoning about concurrent and distributed computations; a prominent example is the π-calculus of Milner, Parrow and Walker [13, 12]. These languages commonly begin by assuming a *countably infinite set of atomic names* $\mathcal{N}$, ranged over by x, y, z. This is not an unreasonable assumption for most purposes, but it does leave open the question of how this set of names should actually be interpreted, e.g. if we were to create an implementation of the π-calculus or one of its variants [20, 16, 5].

A similar issue arises with the scoping operator $(\nu x)P$, which is used to declare a new name x with visibility limited to P. Here the question becomes how we should choose this new name x, such that it is actually ensured to be unique. For a process modelling a program running on a single computer, this can easily be solved, e.g. with a counter; but if the process models a *distributed* system, with programs running on distinct computers, the solution is less obvious. These issues are not directly handled in the π-calculus model, but only become apparent when we consider a more practical implementation of the set of names.

A radically different approach is taken in the Reflective Higher-Order (RHO or ρ) calculus proposed by Meredith and Radestock in [11]. These authors instead begin by positing that the set of names is built by a syntax, similar to the syntax for processes, and thus generated from a *finite* set of elements. One could imagine different possibilities for this syntax, but Meredith and Radestock here make the unusual choice of letting names be 'quoted' processes, written $\ulcorner P \urcorner$. Thus, if P is a process, then $\ulcorner P \urcorner$ is a name. This creates a mutually recursive definition, since processes also contain names. The full syntax of the ρ-calculus is then

$$P \in \mathscr{P}_\rho ::= \mathbf{0} \;\mid\; P_1 \mid P_2 \;\mid\; x\langle\!|P|\!\rangle \;\mid\; x(y).P \;\mid\; \urcorner x \ulcorner$$
$$x, y \in \ulcorner \mathscr{P}_\rho \urcorner ::= \ulcorner P \urcorner$$

*This work was supported by the Icelandic Research Fund Grant No. 218202-05(1-3).

V. Castiglioni and C. A. Mezzina (Eds): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics 2022 (EXPRESS/SOS 2022).
EPTCS 368, 2022, pp. 95–112, doi:10.4204/EPTCS.368.6

Three of the constructs are as in the π-calculus: The *nil* process, $\mathbf{0}$, is the inactive process; The *parallel* construct, $P_1 \mid P_2$, is the parallel composition of processes P_1 and P_2; and the *input* construct, $x(y).P$, is a blocking operation, awaiting a communication on the channel x of some name, which, upon reception, will be bound to y in the continuation P.

The two remaining constructs are particular to the ρ-calculus: The *lift* construct $x\langle\!|P|\!\rangle$ quotes the process P, thereby creating the name $\ulcorner P \urcorner$, and outputs it on x; thus name generation is handled explicitly in the ρ-calculus, rather than implicitly by a π-calculus style ν-operator. This is the second peculiarity of this calculus, since the newly generated name will be *free* in the continuation of the corresponding input, and therefore also *observable* if substituted for the subject of an input or lift. As we shall later see, this feature is crucial for showing a separation result w.r.t. the π-calculus.

Lastly, the *drop* construct $\urcorner x \ulcorner$ removes the quotes of the name to run the process within them, thereby enabling higher-order behaviour (i.e. process mobility). This construct is thus similar to a process variable X in e.g. HOπ [17, 18], and is also the reason for the 'reflective' epithet in the name of this calculus. It derives from Smith [19], who defined reflection as the ability of a program to turn code into data, compute with it, modify it, and turn it back into running code, which in the ρ-calculus is captured by the combination of the lift and drop constructs, and the duality of names and processes.

Although superficially quite similar to the π-calculus, these features suggest that the ρ-calculus is actually rather different. As argued above, the use of structured terms as names, and explicit name generation, seem more realistic from an implementation perspective, as it places the problems of choosing the next name, and of ensuring freshness, within the language itself, rather than simply assuming that these features just work behind the scenes. However, providing a *solution* to these problems is not trivial, as we shall see below. For example, in [11] Meredith and Radestock also propose an encoding of the asynchronous, choice-free fragment of the π-calculus into the ρ-calculus, reviewed in section 3, but as we shall show in section 4, this encoding contains two fatal errors, invalidating their correctness result.

In what follows, we shall instead propose a different encoding of the π-calculus into the ρ-calculus and formally prove its correctness w.r.t. a number of encodability criteria closely related to those proposed by Gorla in [8], but with some adaptations necessitated by the aforementioned peculiar features of the ρ-calculus (Propositions 1-5). Using the same criteria we then derive a separation result, showing that the converse of this statement does not hold: there cannot be an encoding of the ρ-calculus into the π-calculus satisfying the same criteria (Theorem 1).[1] This result is quite surprising, and it suggests that we cannot always just reduce higher-order behaviour to the first-order paradigm, as Sangiorgi was able to do with HOπ in [18]. This is because higher-order behaviour in the ρ-calculus is not just an extension on top of an already computationally complete language, as it is the case with HOπ which extends the 'first-order' π-calculus, but rather appears as a special case of the more general phenomenon of reflection, where processes (code) are communicated without modification.

2 The Reflective Higher-Order Calculus

We begin by presenting the ρ-calculus following Meredith and Radestock in [11]. As we have already seen the syntax above, we shall here focus on the semantics, which we shall give in terms of a reduction system. Firstly, we shall need a notion of *structural congruence* on processes, written $\equiv$. We shall postpone its precise definition slightly, but the intuition is that $P_1 \equiv P_2$ denotes that P_1 and P_2 are the same process, up to some insignificant structural change, such as reordering of components in parallel composition or a change of bound names (α-conversion).

[1] Full proofs of most results are available in a technical report [10].

Now, since names are quoted processes, this notion of structural congruence is extended to the set of names: the *name equivalence* relation, written $\equiv_{\mathscr{N}}$, is defined as the least equivalence on names closed under the following rules:

$$[\text{N-STRUCT}]\frac{P_1 \equiv P_2}{\ulcorner P_1 \urcorner \equiv_{\mathscr{N}} \ulcorner P_2 \urcorner} \qquad [\text{N-DROP}]\frac{x_1 \equiv_{\mathscr{N}} x_2}{\ulcorner\urcorner x_1 \ulcorner\urcorner \equiv_{\mathscr{N}} x_2}$$

The point of [N-STRUCT] is that if the processes within quotes have the same structure (up to structural congruence), then the quoted processes should also represent the same name. Furthermore, by [N-DROP], we allow nested levels of quotes and drops to 'cancel out.'

Next, we shall need the notions of free and bound names, $\text{fn}(P)$ and $\text{bn}(P)$, which are defined in the usual (syntactic) way, with input being the only formal binder in the language. Thus $\text{bn}(x(y).P) = \{y\} \cup \text{bn}(P)$, and all other names are free. We write $\text{n}(P) \triangleq \text{fn}(P) \cup \text{bn}(P)$ for all the names in P, and we also write $x\#P$ to mean that x is *fresh* for P. However, with *structured* names, it is no longer enough that $x \notin \text{n}(P)$; x must also not be *name equivalent* to any name in P. Thus we say $x\#P \triangleq \forall n \in \text{n}(P)\ .\ x \not\equiv_{\mathscr{N}} n$. Lastly, we write $P\{x/y\}$ for the safe substitution of x for y within P. However, given our considerations about $\equiv_{\mathscr{N}}$ above, $P\{x/y\}$ will not only replace y, but also any name that is *name equivalent* to y. Note also, in particular, that substitution does *not* recur into processes under quotes. Thus $\ulcorner P \urcorner\{x/y\} = \ulcorner P \urcorner$ for all names y where $y \not\equiv_{\mathscr{N}} \ulcorner P \urcorner$, and $\ulcorner P \urcorner\{x/y\} = x$ otherwise.

We shall now return to the definition of structural congruence: it is defined as the usual least congruence on processes, containing α-equivalence and the abelian monoid rules for parallel composition with $\mathbf{0}$ as the unit element. However, with structured terms as names, the congruence rules take on a slightly unusual form, since we now also need to compare names. For example, to conclude $x_1(y_1).P_1 \equiv x_2(y_2).P_2$ we would need the following rule in structural congruence:

$$[\text{S-IN}]\frac{x_1 \equiv_{\mathscr{N}} x_2 \quad P_1\{z/y_1\} \equiv P_2\{z/y_2\}}{x_1(y_1).P_1 \equiv x_2(y_2).P_2}\ (z\#P_1, P_2)$$

This yields another mutual recursion between structural congruence and name equivalence.

With these concepts in place, we can at last give the reduction rules for our semantics as follows:

$$[\rho\text{-PAR}]\frac{P_1 \to P_1'}{P_1 \mid P_2 \to P_1' \mid P_2} \qquad [\rho\text{-STRUCT}]\frac{P_1 \equiv P_1' \quad P_1' \to P_2' \quad P_2' \equiv P_2}{P_1 \to P_2}$$

$$[\rho\text{-COM}]\frac{x_1 \equiv_{\mathscr{N}} x_2}{x_1(y).P_1 \mid x_2\langle\!|P_2|\!\rangle \to P_1\{\ulcorner P_2 \urcorner/y\}}$$

The [ρ-PAR] and [ρ-STRUCT] rules are standard (as in e.g. the π-calculus); the former lets us conclude a reduction of one component in a parallel composition, whilst the latter allows us to rewrite the process, using structural congruence $\equiv$, such that its form can match the conclusion of one of the other rules.

The [ρ-COM] rule is also *almost* standard: The process P_2 is quoted and sent out over x_2, and the matching input receives it as the name $\ulcorner P_2 \urcorner$ and substitutes it for y in the continuation P_1. However, since names in the ρ-calculus have structure, we must be able to explicitly conclude the equivalence $x_1 \equiv_{\mathscr{N}} x_2$ between the two subjects in a communication. This is thus different from calculi with atomic names where exact syntactic equality is (usually implicitly) required between subjects.

One last detail concerns substitution: In structural congruence, including α-equivalence, $P\{x/y\}$ is defined as the usual capture-avoiding substitution of names for names. However, the substitution used in the *semantics* is slightly different, as it is also used to handle the $\urcorner x \ulcorner$ construct, which was not given a reduction rule above. The semantic substitution also contains the clause $\urcorner x \ulcorner\{\ulcorner P \urcorner/y\} = P$ if $x \equiv_{\mathscr{N}} y$, thus replacing the *process* $\urcorner x \ulcorner$ with P; and $\urcorner x \ulcorner\{\ulcorner P \urcorner/y\} = \urcorner x \ulcorner$ if $x \not\equiv_{\mathscr{N}} y$. This is the only way in which a $\urcorner x \ulcorner$ is ever executed, and it implies that the drop of a *free* name is a deadlock, as it can never be touched by a substitution at runtime.

3 The encoding of Meredith and Radestock

In [11], Meredith and Radestock proposed an encoding of the asynchronous, choice-free π-calculus, taking full abstraction w.r.t. weak, barbed bisimilarity as their correctness criterion. Unfortunately, that encoding is *not* correct, as we shall now show. The counter-examples are instructive, as they highlight some of the difficulties inherent in working with a calculus without the assumption of an infinite set of atomic names and explicit scoping operators.

First, we recall the syntax and semantics of the asynchronous choice-free π-calculus, as given e.g. in [14]. Note that some of the constructs and concepts are similar to those found in the ρ-calculus. We shall therefore reuse some of the symbols and rely on context to distinguish whether a π-calculus or ρ-calculus construct is meant. The syntax is:

$$P \in \mathscr{P}_\pi ::= \mathbf{0} \;\big|\; P_1 \mid P_2 \;\big|\; x(y).P \;\big|\; \bar{x}\langle z\rangle \;\big|\; (\nu x)P \;\big|\; !P$$

The semantics is given in terms of a reduction system with the rules

$$[\pi\text{-COM}]\frac{}{x(y).P \mid \bar{x}\langle z\rangle \to P\{z/y\}} \qquad [\pi\text{-RES}]\frac{P \to P'}{(\nu x)P \to (\nu x)P'}$$

and with rules for parallel composition and structural congruence similar to those in the ρ-calculus (rules [ρ-PAR] and [ρ-STRUCT] above). Structural congruence $\equiv$ over $\mathscr{P}_\pi$ contains the same rules as in the ρ-calculus, but with syntactic equality replacing name equivalence, and also the following rules for scoping and replication:

$$(\nu x)\mathbf{0} \equiv \mathbf{0} \qquad !P \equiv P \mid !P$$
$$(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P \qquad (\nu x)P_1 \mid P_2 \equiv (\nu x)(P_1 \mid P_2) \text{ if } x \notin \mathrm{fn}(P_2)$$

Now for the encoding, assume a function $\varphi : \mathscr{N} \to \ulcorner \mathscr{P}_\rho \urcorner$ from π-calculus atomic names to ρ-calculus names. Since the set of π-calculus names is countably infinite, it can for example be mapped to the set of natural numbers. The function φ could then be regarded as an enumeration of names (or a successor function), starting e.g. from $\ulcorner \mathbf{0} \urcorner$ for the name x_0, and then letting the name x_{i+1} be defined in terms of the name x_i as for example $x_{i+1} \triangleq \ulcorner x_i \langle\!| \mathbf{0} |\!\rangle \urcorner$. In the sequel, we shall say that $\ulcorner x \langle\!| \mathbf{0} |\!\rangle \urcorner$ is a *left increment* of x, written $+x$. Then we can generate a countably infinite sequence of names $x_0, x_1, x_2, \ldots$, starting from any name $x = x_0$, as $+x = x_1, ++x = +x_1 = x_2, \ldots$ and so on. This shows that the set of π-calculus names can be implemented as ρ-names, as, by the definition of name equivalence and structural congruence, we have that $x \not\equiv_{\mathscr{N}} \ulcorner x \langle\!| \mathbf{0} |\!\rangle \urcorner$.

Correspondingly we can define $x+ \triangleq \ulcorner x(\ulcorner \mathbf{0} \urcorner).\mathbf{0} \urcorner$ as a *right increment* of x, which gives us another countably infinite sequence. Another option is *name composition* $x \cdot y \triangleq \ulcorner x \langle\!| \mathbf{0} |\!\rangle \mid y(\ulcorner \mathbf{0} \urcorner).\mathbf{0} \urcorner$, which yields yet another sequence with $x^2 = x \cdot x, x^3 = x^2 \cdot x, x^4 = x^3 \cdot x, \ldots$ and so on.

These are all examples of *static quoting* techniques for consistent name generation, and each could be used to implement the function φ. Given such techniques, Meredith and Radestock then begin by assuming that all π-calculus names are already implemented as ρ-names. Their translation function $[\![P]\!]_{n_0,p_0}$ requires two names as parameters, which must be chosen such that they are distinct from all the names in P, and furthermore that no name *within* P can ever be *generated* from n_0 or p_0 by means of the aforementioned methods of static name generation. One way of ensuring this is by letting

$$n_0 = \ulcorner \prod_{x \in \mathrm{fn}(P)} x \langle\!| \mathbf{0} |\!\rangle \urcorner \qquad \text{and} \qquad p_0 = \ulcorner \prod_{x \in \mathrm{fn}(P)} x(\ulcorner \mathbf{0} \urcorner).\mathbf{0} \urcorner$$

where $\prod$ denotes generalised parallel composition.

The translation function also uses two short-hands: $D(x) \triangleq x(y).(\ulcorner y \urcorner \mid x \langle\!| \ulcorner y \urcorner |\!\rangle)$ is a copying process used to implement replication; and $x\langle y\rangle \triangleq x \langle\!| \ulcorner y \urcorner |\!\rangle$ simulates output in the π-calculus, since by [N-DROP]

we have that $\ulcorner\urcorner y \ulcorner\urcorner \equiv_{\mathscr{N}} y$. The translation $[\![P]\!] = [\![P]\!]_{n_0,p_0}$ [11, p. 13] is then given by the following recursive equations:[2]

$$\begin{aligned}
[\![\mathbf{0}]\!]_{n,p} &= \mathbf{0} & [\![P_1 \mid P_2]\!]_{n,p} &= [\![P_1]\!]_{+n,+p} \mid [\![P_2]\!]_{n+,p+} \\
[\![\bar{x}\langle y\rangle]\!]_{n,p} &= x\langle y\rangle & [\![(\nu x)P]\!]_{n,p} &= p(x).[\![P]\!]_{+n,+p} \mid p\langle n\rangle \\
[\![x(y).P]\!]_{n,p} &= x(y).[\![P]\!]_{n,p} \\
[\![!P]\!]_{n,p} &= n\cdot p\,\langle\!|n{+}(n).p{+}(p).([\![P]\!]_{n,p} \mid D(n\cdot p) \mid n{+}\langle\!|n\langle n\rangle|\!\rangle \mid p{+}\langle\!|p\langle p\rangle|\!\rangle)|\!\rangle \\
&\quad \mid D(n\cdot p) \mid n{+}\langle {+}n\rangle \mid p{+}\langle {+}p\rangle
\end{aligned}$$

A central element in this translation is the encoding of replication, $[\![!P]\!]_{n,p}$, so we shall give some further details about its underlying intuitions. Firstly, with higher-order process mobility, we can create a diverging process simply as $x\langle\!|D(x)|\!\rangle \mid D(x)$. This construction is reminiscent of the λ-calculus Ω-combinator $(\lambda x.xx)\lambda x.xx$: $D(x)$ will run the process it receives on x whilst simultaneously making it available again on x, so by sending it a copy of $D(x)$ itself, we obtain a process that continuously copies itself. Then, by embedding another process P in this construct, $x\langle\!|P \mid D(x)|\!\rangle \mid D(x)$, we obtain a process that will create arbitrarily many copies of P at runtime. Thus we can implement unguarded replication by using just a single name x. However, this name x must not be used by any other process, lest it might interfere with the replication. This is achieved in the above encoding by composing the two name parameters, n and p, to obtain a new name $n\cdot p$.

Secondly, if $[\![P]\!]_{n,p}$ were simply copied in this fashion, any usage of the parameters n and p within the translation of P would also be copied, which thus could create a name clash. Therefore, the inner process is prefixed with two inputs that *bind* n and p within the continuation. In parallel, we then have two other processes, $n{+}\langle{+}n\rangle$ and $p{+}\langle{+}p\rangle$, that output the new names $+n$ and $+p$, which will be substituted for n and p. These processes are also copied, and in the next round of replication they will instead create the names $\ulcorner{+}n\langle{+}n\rangle\urcorner$ and $\ulcorner{+}p\langle{+}p\rangle\urcorner$, and so on, thereby implementing a *runtime* form of name generation, similar to our static quoting technique.

For the purpose of defining a notion of behavioural equivalence that is comparable to that of other calculi that do feature a ν-operator, Meredith and Radestock define a *name-restricted observation predicate* $\downarrow^{\mathscr{N}}$ for the ρ-calculus, parametrised with a set of names $\mathscr{N}$. The idea is to only allow observation of names in this set. We follow their definition, but also allow the observation predicate to distinguish between input x, and output $\bar{x}$:[3]

$$[\rho\text{-BOUT}]\frac{x_1 \equiv_{\mathscr{N}} x_2 \quad x_1 \in \mathscr{N}}{x_1\langle\!|P|\!\rangle \downarrow^{\mathscr{N}} \overline{x_2}} \qquad [\rho\text{-BIN}]\frac{x_1 \equiv_{\mathscr{N}} x_2 \quad x_1 \in \mathscr{N}}{x_1(y).P \downarrow^{\mathscr{N}} x_2} \qquad [\rho\text{-BPAR}]\frac{P_1 \downarrow^{\mathscr{N}} \widehat{x} \quad \vee \quad P_2 \downarrow^{\mathscr{N}} \widehat{x}}{P_1 \mid P_2 \downarrow^{\mathscr{N}} \widehat{x}}$$

where $\widehat{x}$ ranges over $x, \bar{x}$. An *$\mathscr{N}$-restricted barbed bisimulation* is then a symmetric, binary relation $\mathscr{R}^{\mathscr{N}}$ on processes, parametrised with a set of names $\mathscr{N}$, such that $(P_1,P_2) \in \mathscr{R}^{\mathscr{N}}$ implies:

- If $P_1 \to P_1'$ then there exists a P_2' such that $P_2 \to P_2'$ and $(P_1',P_2') \in \mathscr{R}^{\mathscr{N}}$.
- If $P_1 \downarrow^{\mathscr{N}} \widehat{x}$ then $P_2 \downarrow^{\mathscr{N}} \widehat{x}$.

We say that P_1 is *$\mathscr{N}$-restricted barbed bisimilar* to P_2, written $\sim^{\mathscr{N}}$, if there exists an $\mathscr{N}$-restricted barbed bisimulation $\mathscr{R}^{\mathscr{N}}$ such that $(P_1,P_2) \in \mathscr{R}^{\mathscr{N}}$. The corresponding 'weak' observation predicate is then written

$$P \Downarrow^{\mathscr{N}} \widehat{x} \triangleq \exists P' \,.\, P \to^* P' \wedge P' \downarrow^{\mathscr{N}} \widehat{x}$$

[2]The translation has been adapted to use our notation for name increments, which we find more intuitive than x^l and x^r, which is used in the original presentation. We also use $x\langle y\rangle$ rather than $x[y]$ for output, which is more in line with standard π-calculus notation.

[3]The added distinction between input and output observations is only for use in our later development of a correct encoding, and does not invalidate our claim that the encoding by Meredith and Radestock is incorrect, since our counter-examples shall only rely on observing outputs.

where $\rightarrow^*$ is the reflexive and transitive closure of $\rightarrow$, and by replacing $P_2 \downarrow^{\mathscr{N}} \widehat{x}$ with $P_2 \Downarrow^{\mathscr{N}} \widehat{x}$, and $P_2 \rightarrow P_2'$ with $P_2 \rightarrow^* P_2'$ in the above definition, we obtain the corresponding notion of a *weak* $\mathscr{N}$-restricted barbed bisimulation. We say that P_1 is *weakly $\mathscr{N}$-restricted barbed bisimilar* to P_2, written $\approx^{\mathscr{N}}$, if there exists a weak $\mathscr{N}$-restricted barbed bisimulation $\mathscr{R}^{\mathscr{N}}$ that relates them.

The corresponding observation predicate for the π-calculus is built by the following rules for observation on output, restriction and replication

$$[\pi\text{-BOUT}]\frac{x \in \mathscr{N}}{\overline{x}\langle y\rangle \downarrow^{\mathscr{N}} \overline{x}} \qquad [\pi\text{-BRES}]\frac{P \downarrow^{\mathscr{N}} \widehat{x}}{(\nu z)P \downarrow^{\mathscr{N}} \widehat{x}}\ (x \neq z) \qquad [\pi\text{-BREP}]\frac{P \downarrow^{\mathscr{N}} \widehat{x}}{!P \downarrow^{\mathscr{N}} \widehat{x}}$$

and with rules similar to $[\rho\text{-BPAR}]$ and $[\rho\text{-BIN}]$ in the ρ-calculus for observation on parallel composition and input, with strict syntactic equality replacing name equivalence in the premise of the latter rule. The notions of a weak observation predicate, and (strong resp. weak) $\mathscr{N}$-restricted barbed bisimulation and bisimilarity for the π-calculus are then defined as in the ρ-calculus. We write $P \downarrow \widehat{x}$, $P \Downarrow \widehat{x}$, $P_1 \sim P_2$ and $P_1 \approx P_2$ when $\mathscr{N}$ is the set of all names, corresponding to no restriction on the names we can observe. This yields the familiar notions of (strong resp. weak) barbed bisimilarity in the π-calculus (as defined in e.g. [12]).

Given these notions of behavioural equivalence, Meredith and Radestock then state the following as a theorem [11, p. 14, Theorem 5.3], but without providing a proof:

$$P_1 \approx P_2 \iff [\![P_1]\!] \approx^{\mathrm{fn}(P_1)\cup\mathrm{fn}(P_2)} [\![P_2]\!] \tag{1}$$

with observation in the ρ-calculus restricted to $\mathrm{fn}(P_1) \cup \mathrm{fn}(P_2)$, i.e. the free names in P_1 and P_2, implemented as ρ-names.[4]

4 The errors

We shall now see why the claim stated in 1 does not hold. Firstly, consider the following π-calculus processes:

$$P_1 \triangleq !(\nu z)\overline{u}\langle z\rangle \qquad \text{and} \qquad P_2 \triangleq (\nu z)\,!\overline{u}\langle z\rangle$$

Clearly, they represent different behaviours: P_2 will continuously send out the *same* fresh name z on u, whilst P_1 will send out *different* fresh names, as we can see by applying α-conversion after unfolding the replication (see [10, p. 11] for details). We can also easily construct a testing context C such that they can be distinguished by the (π-calculus) $\Downarrow \overline{x}$ predicate, for example

$$C \triangleq [\,] \mid u(n_1).u(n_2).(\overline{n_1} \mid n_2.\overline{x})$$

where the objects for the input/output of $\overline{n_1}, n_2$ and $\overline{x}$ are ignored, as this only requires pure synchronisation. Clearly, if the two names received on u are the same, then n_1 and n_2 will be the same name, so they can synchronise and we will therefore be able to observe $\overline{x}$ after 3 reduction steps. And conversely, if the two names are distinct, then we will not observe $\overline{x}$. Thus $C[P_1] \not\Downarrow \overline{x}$ whilst $C[P_2] \Downarrow \overline{x}$ as argued above.

Now we make a slight adjustment to the two terms. By composing an arbitrary process Q with the inner output process $\overline{u}\langle z\rangle$ we obtain the following:

$$P_1' \triangleq !((\nu z)\overline{u}\langle z\rangle \mid Q) \quad \text{and} \quad P_2' \triangleq (\nu z)\,!(\overline{u}\langle z\rangle \mid Q)$$

[4] Note that the original presentation [11, p. 14, Theorem 5.3] only has $P_1 \approx P_2 \iff [\![P_1]\!] \approx^{\mathrm{fn}(P_1)} [\![P_2]\!]$, but we regard this as a simple omission, since it trivially would not hold for the implication from right to left: Take for example $P_1 \triangleq x\langle z\rangle$ and $P_2 \triangleq x\langle z\rangle \mid w\langle z\rangle$. Then we have that $\mathrm{fn}(P_1) = \{x\}$, and indeed $[\![P_1]\!] \approx^{\{x\}} [\![P_2]\!]$ since for $i \in \{1,2\}$ we have that $[\![P_i]\!] \not\rightarrow$ and $[\![P_i]\!] \downarrow^{\{x\}} \overline{x}$; but obviously $P_1 \not\approx P_2$, since $P_2 \downarrow \overline{w}$ but $P_1 \not\downarrow \overline{w}$.

The actual behaviour of Q is irrelevant; it is there solely to induce the parameter pair (n,p) to be split into a 'left pair' $(+n,+p)$ and a 'right pair' $(n+,p+)$ that are passed to the translations of the left (resp. right) parts of the parallel composition. Note also that this changes nothing w.r.t. observability of $\bar{x}$: we still have that $C[P_1'] \not\Downarrow \bar{x}$ and $C[P_2'] \Downarrow \bar{x}$.

We shall now perform the actual translation. To make it more readable, we tabulate the names generated by static quoting during the translation and rename them as follows:

$$\begin{array}{cccccc} n\cdot p = a & p+ = c & +p = e & (+p)+ = g & ++n = i \\ n+ = b & +n = d & (+n)+ = f & (+n)\cdot(+p) = h & ++p = j \end{array}$$

Note that *none* of these names will be observable by the $\Downarrow^{\mathrm{fn}(P_1)\cup\mathrm{fn}(P_2)}$ predicate, because they are generated by the translation, and hence are not in the set $\mathrm{fn}(P_1)\cup\mathrm{fn}(P_2)$ of free names of P_1 and P_2. Now, here is the translation:

$$\begin{aligned} [\![P_1']\!]_{n,p} &= a\Big\langle\Big| b(n).c(p).\Big(e(z).u\langle z\rangle \mid e\langle d\rangle \mid [\![Q]\!]_{f,g} \mid D(a) \mid b\langle\!\langle n\langle n\rangle\rangle\!\rangle \mid c\langle\!\langle p\langle p\rangle\rangle\!\rangle\Big)\Big|\Big\rangle \\ &\quad \mid D(a) \mid b\langle d\rangle \mid c\langle e\rangle \\ [\![P_2']\!]_{n,p} &= p(z).h\Big\langle\Big| f(d).g(e).\Big(u\langle z\rangle \mid [\![Q]\!]_{f,g} \mid D(h) \mid f\langle\!\langle d\langle d\rangle\rangle\!\rangle \mid g\langle\!\langle e\langle e\rangle\rangle\!\rangle\Big)\Big|\Big\rangle \\ &\quad \mid D(h) \mid f\langle i\rangle \mid g\langle j\rangle \mid p\langle n\rangle \end{aligned}$$

By performing the reductions, we see (not surprisingly) that $[\![P_2']\!]_{n,p}$ firstly performs the communication on p, which causes z to be replaced by n, and the process afterwards expands into arbitrarily many instances of $u\langle n\rangle$ (see [10, p. 12] for a reduction sequence). On the other hand, the translated process $[\![P_1']\!]_{n,p}$ will immediately go through the replication steps, thereby creating arbitrarily many instances of the process $e(z).u\langle z\rangle \mid e\langle d\rangle$ corresponding to the translation of $(\nu z)\bar{u}\langle z\rangle$. This process obviously reduces to $u\langle d\rangle$ in one step. However, precisely because of the aforementioned split of (n,p) over the translation of parallel composition, the name d will *not* be updated by the replication context. This process will therefore *also* repeatedly output the *same* name d on u, and the (translated) form of our testing context can therefore no longer distinguish the processes.

Both $[\![P_1']\!]$ and $[\![P_2']\!]$ thus reduce to arbitrarily many copies of either $u\langle d\rangle$ (for P_1') or $u\langle n\rangle$ (for P_2'), and u is the only name we can observe, as all the other names are created by the translation. This then gives us our desired counter-example: by also translating the testing context we obtain a pair of processes where

$$C[P_1'] \not\approx C[P_2'] \qquad \text{but} \qquad [\![C[P_1']]\!] \approx^{\mathrm{fn}(C[P_1'])\cup\mathrm{fn}(C[P_2'])} [\![C[P_2']]\!]$$

in contradiction of the implication from right to left in the claim stated in 1.

The detailed analysis above gives us a clear idea of the root cause of the problem: The translation of replication creates a context with the purpose of ensuring that the names (n,p) used within it will repeatedly be substituted with new, fresh names $(+n,+p)$ dynamically built from the previous names, and these act as sources of new names for any occurrence of $(\nu z)P$ within a replicated process. The point is precisely to ensure that each instance of a replicated ν operator will generate a unique new name, and the parameters (n,p) on the translation function act as 'handles' to access this resource; they are the names that have *most recently* been replicated.

The problem arises because this property of being the 'most recently replicated names' is not preserved by the translation of parallel composition: It splits the pair into a left and a right pair, used in the translation of the left and right parallel components:

$$[\![P_1 \mid P_2]\!]_{n,p} = [\![P_1]\!]_{+n,+p} \mid [\![P_2]\!]_{n+,p+}$$

Thus, the access to the most recently replicated names is lost in the translation of the inner processes, because, as we noted above, substitution does *not* recur into processes under quotes. Therefore, when the replication context increments (n, p) at runtime, this update cannot touch the n and p embedded in the *statically* incremented names $(+n, +p)$ and $(n+, p+)$ which the translation function generates for the translation of parallel composition. This is why we added an arbitrary Q to create a parallel composition in our counter-example above.

However, the error above is not the only one in the claim by Meredith and Radestock: whilst its root cause was the splitting of names over the translation of parallel composition, we can also create another example that is more directly related to the interplay between $(\nu x)P$ and replication. Consider the following processes:

$$P_1 \triangleq !(\nu z)\overline{u}\langle z\rangle \qquad \text{and} \qquad P_2 \triangleq !(\nu q)(\nu z)\overline{u}\langle z\rangle$$

Note that P_1 and P_2 are structurally congruent, since the new name q is never used. Thus $P_1 \approx P_2$ also holds. Yet when we translate those terms, the name incrementation in the translation of a term of the form $(\nu x)P$ means that we again lose access to the most recently replicated names from the translation of replication. This can be easily seen if we perform the translation stepwise, using the same tabulated list of names as before. For both processes, the translation of replication is the same:

$$[\![!P]\!]_{n,p} = a\langle\!|b(n).c(p).([\![P]\!]_{n,p} \mid D(a) \mid b\langle\!|n\langle n\rangle|\!\rangle \mid c\langle\!|p\langle p\rangle|\!\rangle)|\!\rangle \mid D(a) \mid b\langle d\rangle \mid c\langle e\rangle$$

Now let $P_1' \triangleq (\nu z)\overline{u}\langle z\rangle$ and $P_2' \triangleq (\nu q)(\nu z)\overline{u}\langle z\rangle$ and replace $[\![P]\!]_{n,p}$ above with $[\![P_1']\!]_{n,p}$ and $[\![P_2']\!]_{n,p}$ respectively. The translations of the inner processes yield:

$$[\![(\nu z)\overline{u}\langle z\rangle]\!]_{n,p} = p(z).u\langle z\rangle \mid p\langle n\rangle$$
$$[\![(\nu q)(\nu z)\overline{u}\langle z\rangle]\!]_{n,p} = p(q).(e(z).u\langle z\rangle \mid e\langle d\rangle) \mid p\langle n\rangle$$

which reduce to $u\langle n\rangle$ and $u\langle d\rangle$ respectively. The names n, p are bound in the replication context and will therefore be updated whenever the process replicates. However, in the case of P_2, these names are *statically* incremented in the translation of (νq) to yield the names $+n = d$ and $+p = e$, and these two names will therefore *not* be updated at runtime, just as in the previous counter-example. Consequently, in the case of P_2 the names sent out on u will *not* be distinct; they will all be the name $+n = d$. We can therefore use the same testing context C as in the previous example and proceed as before to generate another contradiction of the claim in 1; this time by *distinguishing* the translated terms, although we have $C[P_1] \approx C[P_2]$ in the π-calculus. In summary, neither of the implications in the claim stated in 1 hold.

5 Our criteria for encodability

Both of the previous examples illustrate the difficulties involved in reasoning about a parametrised translation. Usually, the parameters represent a property or invariant that is assumed to be preserved throughout the translation, and a proof of correctness of the translation must therefore also include a proof that this invariant or property is indeed preserved. For example, in the present case, the invariant assumed to hold for the parameters is precisely that they always refer to the most recently replicated names. However, this assumption is never formally stated in the original ρ-calculus paper [11], and as the examples above show, it does not hold either. Thus, a naive attempt to show correctness of the translation by induction in the clauses of the translation function may therefore seemingly go through, if the parameters are not considered. This is doubly problematic in the present case, because the observation predicate used

in the bisimulation relation over ρ-calculus terms is parametrised so that we do *not* observe the names created by the translation function.

Full abstraction, of which the claim in 1 is an instance, may also not be the most informative correctness criterion, as argued by Gorla and Nestmann [9]; for example, it does not necessarily prevent the translation from introducing divergence. Also, as we are here more interested in showing that the π-calculus is 'implementable' in the ρ-calculus than in transferring equations between the source and target language, we shall instead follow the approach of such authors as Gorla [8], Carbone and Maffeis [4] and others, and state a number of criteria for what we consider a valid encoding, where we also take the presence of parameters into account:

Definition 1 (Language)**.** *A* language *$\mathscr{L}$ is a tuple $\mathscr{L} \triangleq (\mathscr{P}, \mathscr{N}, \to, \simeq)$, where $\mathscr{P}$ is a set of terms, $\mathscr{N}$ is a set of names, $\to \subseteq \mathscr{P} \times \mathscr{P}$ is the reduction relation, with $\to^*$ denoting the reflexive and transitive closure of $\to$, and $\simeq \subseteq \mathscr{P} \times \mathscr{P}$ is a notion of behavioural equivalence.*

We say a term $P \in \mathscr{P}$ *diverges*, written $P \to^\omega$, if P has an infinite reduction sequence. We use $\sigma : \mathscr{N} \to \mathscr{N}$ to denote a substitution function in $\mathscr{L}$. For encodings, we need the notion of a *source* and a *target* language, and we shall generally use the convention of subscripting s (for source) and t (for target) to a language $\mathscr{L}$ or its components, including substitutions, and we let $S \in \mathscr{P}_s$ and $T \in \mathscr{P}_t$.

Definition 2 (Encoding)**.** *An* encoding *of $\mathscr{L}_s$ into $\mathscr{L}_t$ is a tuple $(\llbracket\ \rrbracket_N, \varphi, \delta)$, where $\llbracket\ \rrbracket_N : \mathscr{P}_s \to \mathscr{P}_t$ is a* translation function*, parametrised with a finite list of names $N \in \mathscr{N}_t^{\,k}$; and $\varphi : \mathscr{N}_s \to \mathscr{N}_t$ is a* renaming policy*, mapping names in the source language into names in the target language; and $\delta : \mathscr{N}_t^{\,k} \to \mathscr{N}_t^{\,k}$ is a* name derivation *function, mapping k-ary tuples of target names to tuples of equal arity for some k.*

The name derivation function δ allows us to express that the list of name parameters N may evolve in some predictable way during the course of translation. This seems necessary in particular when we are working with a language with structured terms as names. In some cases we may also need to derive multiple tuples of names from the same input tuple; thus to comply with the requirement that δ is a single function, we could e.g. envision using an extra, designated name as argument to control the derivation method used by δ. However, to abstract away from such details, we say that a tuple of names N_2 is *derivable* from some tuple of names N_1, written $N_1 \rightsquigarrow N_2$, if $\delta(N_1) = N_2$, and likewise that $N_1 \rightsquigarrow n$ if $n \in N_2$. Note that we abuse the notation slightly and treat the lists as sets when the position of each individual component does not matter.

Definition 3 (Valid encoding)**.** *We shall regard an encoding as* valid*, if it satisfies at least the following criteria:*

1. *Compositionality: $\llbracket S_1 \mid \ldots \mid S_n \rrbracket_N = C \mid \llbracket S_1 \rrbracket_{N_1} \mid \ldots \mid \llbracket S_n \rrbracket_{N_n}$ where C is an optional coordinating context and $\mathrm{fn}(C) \subseteq \varphi(\mathrm{fn}(S_1 \mid \ldots \mid S_n)) \cup N$, and for each $i \in \{1, \ldots, n\}$ we have that $N \rightsquigarrow N_i$.*
2. *Substitution invariance: $\llbracket S\sigma_s \rrbracket_N \simeq \llbracket S \rrbracket_N \sigma_t$ for each σ_s, where $\varphi(\sigma_s(x)) = \sigma_t(\varphi(x))$.*
3. *Operational correspondence: $S \to^* S' \iff \exists T' \,.\, \llbracket S \rrbracket_N \to^* T' \wedge T' \simeq \llbracket S' \rrbracket_{N'}$ and $N \rightsquigarrow N'$.*
4. *Observational correspondence: We require that $N \cap \varphi(\mathscr{M}) = \emptyset$ for any set of observable names $\mathscr{M}$. Then $P \downarrow^{\mathscr{M}} \widehat{x} \iff \llbracket P \rrbracket_N \Downarrow^{\varphi(\mathscr{M})} \varphi(\widehat{x})$.*
5. *Divergence reflection: $\llbracket P \rrbracket_N \to^\omega \implies P \to^\omega$.*
6. *Parameter independence: $\llbracket P \rrbracket_{N_1} \simeq \llbracket P \rrbracket_{N_2}$ for each finite N_1, N_2.*

These criteria are very close to those proposed by Gorla [8], except that we have chosen observational correspondence, rather than the less specific success testing; i.e. $P \to^* \downarrow \checkmark$ implies $\llbracket P \rrbracket_N \to^* \downarrow \checkmark$. This

can easily be obtained, simply by choosing a specific name x and then defining $\checkmark$ as a process with x in subject position, as we did in our counter-examples above.

Furthermore, as we are here allowing parameters to appear on the translation, we have also added the criterion of parameter independence, which does not appear in [8]. This is just to ensure that the behaviour of the translated terms will not depend on the exact choice of the parameters. Likewise, we have also added name restriction to the observation predicate for observational correspondence $\Downarrow^{\mathscr{M}}$, and we require that $N \cap \mathscr{M} = \emptyset$; i.e. that the parameters should not be observable. This seems a natural requirement, since we also require that $N \subseteq \mathscr{N}_t$; i.e. that the parameters belong to the target language. They should therefore not be observable on the source terms.

6 A correct encoding

As the previous examples have illustrated, the main difficulty in creating an encoding of the π-calculus in the ρ-calculus, is how to achieve a robust source of fresh names at runtime that are guaranteed never to cause a name clash. One way is to use a dedicated process for this purpose. Consider the following process, where $D(x)$ is defined as in section 3:

$$!N(x,z,v,s) \triangleq D(x) \mid x\Big\langle\Big|z(a).v(r).\Big(D(x) \mid r\langle\!|\ulcorner a \urcorner|\!\rangle \mid z\langle\!|a\langle\!|\mathbf{0}|\!\rangle|\!\rangle\Big)\Big|\Big\rangle \mid z\langle\!|\ulcorner s \urcorner|\!\rangle$$

This process is a *name server*; it consistently generates names corresponding to consecutive left-increments of the initial name s and outputs them on the 'return address' r received on v. We refer to the above form as the *initial state* of the name server and note that after two reductions it evolves to the form

$$v(r).\Big(D(x) \mid r\langle\!|\ulcorner s \urcorner|\!\rangle \mid z\langle\!|\ulcorner\ulcorner s \urcorner\urcorner\langle\!|\mathbf{0}|\!\rangle|\!\rangle\Big) \mid x\Big\langle\Big|z(a).v(r).\Big(D(x) \mid r\langle\!|\ulcorner a \urcorner|\!\rangle \mid z\langle\!|a\langle\!|\mathbf{0}|\!\rangle|\!\rangle\Big)\Big|\Big\rangle$$

which we refer to as its *ready state*, where it blocks, awaiting a request for a new name on v. The first request will return $\ulcorner\urcorner s \urcorner\urcorner$; a second request will return $\ulcorner\ulcorner\urcorner s \urcorner\urcorner\langle\!|\mathbf{0}|\!\rangle\urcorner = +s$, and so on.

We can verify that the names will all be distinct by considering the *quote depth* of a name (resp. process) defined thus:

$$\mathrm{QD}(\ulcorner P \urcorner) = \begin{cases} \mathrm{QD}(x) & \text{if } P \equiv \urcorner x \ulcorner \\ 1 + \mathrm{QD}(P) & \text{otherwise} \end{cases} \qquad \mathrm{QD}(P) = \begin{cases} \max\{\,\mathrm{QD}(x) \mid x \in \mathrm{fn}(P)\,\} & \text{if } \mathrm{fn}(P) \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

The quote depth of a name x_1 corresponds to the maximum number of calls to [N-STRUCT] used to conclude name equivalence $x_1 \equiv_{\mathscr{V}} x_2$ for some name x_2. Thus, a necessary (but not sufficient) condition for two names to be name equivalent is that they have the same quote depth. Names are therefore automatically stratified based on their quote depth:

Lemma 1 (Stratification)**.** $x_1 \equiv_{\mathscr{V}} x_2 \implies \mathrm{QD}(x_1) = \mathrm{QD}(x_2)$.

We can also partition names into *namespaces* in the following way: let $\mathscr{N}_{[\,]}$ be a collection of name contexts, ranged over by N, with one or more holes occurring in the position of free names. If s is a name, then so is $N[s]$ for some $N \in \mathscr{N}_{[\,]}$. We write $\mathscr{N}_{[s]} \triangleq \{\,N[s] \mid N \in \mathscr{N}_{[\,]}\,\}$, and we say that $\mathscr{N}_{[s]}$ is a *namespace rooted at* s. Clearly, if $\mathrm{QD}(N) = n$ (counting $\mathrm{QD}([\,]) = 0$), and $\mathrm{QD}(s) = i$ and $\mathrm{QD}(s') = j$, then $\mathrm{QD}(N[s]) = n + i$ and $\mathrm{QD}(N[s']) = n + j$.

Using the concepts of name contexts, we can describe our aforementioned three static quoting techniques as three distinct name space 'templates,' built by the following grammars:

$$\begin{array}{lll} {}^{+}N \in {}^{+}\mathscr{N}_{[\,]} ::= [\,] & \mid & \ulcorner {}^{+}N\langle\!|\mathbf{0}|\!\rangle\urcorner \\ N^{+} \in \mathscr{N}^{+}_{[\,]} ::= [\,] & \mid & \ulcorner N^{+}(\ulcorner\mathbf{0}\urcorner).\mathbf{0}\urcorner \\ N^{\circ} \in \mathscr{N}^{\circ}_{[\,]} ::= [\,] & \mid & \ulcorner N^{\circ}\langle\!|\mathbf{0}|\!\rangle \mid N^{\circ}(\ulcorner\mathbf{0}\urcorner).\mathbf{0}\urcorner \end{array}$$

We shall use these namespace templates to implement the name derivation function δ. Thus, if we let $\widehat{N}$ denote any of the name contexts ${}^{+}N, N^{+}, N^{\circ}$ then $s \rightsquigarrow s'$ if there exists a name context $\widehat{N}$ such that $s' \equiv_{\mathscr{N}} \widehat{N}[s]$. This assures us that even if two namespaces use the same structure, e.g. ${}^{+}\mathscr{N}_{[\,]}$, all their names will still be distinct *if* their roots are not name equivalent, and neither is derivable from the other.

In case of the name server, we see that it generates the namespace ${}^{+}\mathscr{N}_{[s]}$, i.e. the namespace of left-increments rooted at s, where s is a parameter. Thus if $s_1 \not\equiv_{\mathscr{N}} s_2$ and neither is derivable from the other, then $!N(x,z,v,s_1)$ and $!N(x,z,v,s_2)$ will generate similarly structured namespaces, ${}^{+}\mathscr{N}_{[s_1]}$ and ${}^{+}\mathscr{N}_{[s_2]}$, but consisting of different sets of names. Yet we can easily construct a mapping ${}^{+}\mathscr{N}_{[s_1]} \mapsto {}^{+}\mathscr{N}_{[s_2]}$ simply by replacing s_1 with s_2 within each name ${}^{+}N[s_1] \in {}^{+}\mathscr{N}_{[s_1]}$. This will be important in the proof for parameter independence below.

Based on these considerations we can now construct our encoding. We let the encoding be defined as $[\![P]\!] \triangleq [\![P]\!]_{n,v} \mid !N(x,z,v,s)$, where we assume we can choose the names n,v,x,z,s such that they are distinct from all free names in P and $n,v,x,z \notin {}^{+}\mathscr{N}_{[s]}$. As in the encoding by Meredith and Radestock, we shall assume that all π-calculus names are implemented as ρ-names, and thus we shall generally omit explicit reference to φ in the following. We shall also limit ourselves to the π-calculus fragment with only input-guarded replication, to ensure that the encoding does not introduce divergence, unlike the encoding by Meredith and Radestock which replicates eagerly and therefore always diverges.[5] This can be achieved by prefixing the object of the lift with an input construct, i.e. $n\langle\!|x(y).(D(n) \mid P)|\!\rangle$, since

$$D(n) \mid n\langle\!|x(y).(D(n) \mid P)|\!\rangle \to x(y).(D(n) \mid P) \mid n\langle\!|x(y).(D(n) \mid P)|\!\rangle$$

and the process then blocks until it receives a communication on x. Given these considerations, the translation function $[\![\ ]\!]_{n,v}$ is then given by the following equations:

$$\begin{array}{ll}
[\![\mathbf{0}]\!]_{n,v} = \mathbf{0} & [\![\bar{x}\langle z\rangle]\!]_{n,v} = x\langle z\rangle \\
[\![P_1 \mid P_2]\!]_{n,v} = [\![P_1]\!]_{+n,v} \mid [\![P_2]\!]_{n+,v} & [\![(\nu x)P]\!]_{n,v} = v\langle n\rangle \mid n(x).[\![P]\!]_{n\cdot n,v} \\
[\![x(y).P]\!]_{n,v} = x(y).[\![P]\!]_{n,v} & [\![\,!x(y).P]\!]_{n,v} = D(n) \mid n\langle\!|x(y).(D(n) \mid [\![P]\!]_{n\cdot n,v})|\!\rangle
\end{array}$$

The idea is that we simplify the 'bookkeeping' involved in runtime name generation by isolating it to a single, contextual process. This prevents errors of the first kind in the encoding by Meredith and Radestock, which resulted from processes losing access to the most recently replicated names. Here, the name v is used by all processes to contact the name server, and since it is never updated this access can never be lost. Conversely, the name n, which is used for the 'return address,' as well as for replication, is *always* updated incrementally, during the translation. It is never bound or reused, unlike in the translation by Meredith and Radestock, where the replication context used $+n, +p$ but also bound n, p and passed them to the inner translation of P, which resulted in the second kind of error. We say that a name is *unique for the translation* if it is never generated more than once by the translation function, and this is the invariant that should hold for the parameter n:

Lemma 2 (Uniqueness)**.** *For each clause $[\![C[P]\,]\!]_{n,v} = [\![C]\!]_{n,v}[[\![P]\!]_{n',v}]$, where $n \rightsquigarrow n'$, and $[\![C]\!]_{n,v}$ contains a set of names $N' = \{n_1, \ldots, n_k\}$ such that $n \rightsquigarrow N'$, it holds that if n is unique for the translation, then so are $n_1, \ldots, n_k$ and n'.*

This can easily be shown by examining the clauses of the translation function, assuming n is unique. For every usage of n in a clause, we always either increase the quote depth of the parameter we pass to the inner call to the translation, or we shift the parameter into a new namespace by composition. Furthermore, the behaviour of the translated process does not depend on the structure of the name parameter n, as long as n is unique:

[5] This is only a slight limitation, as we can use input-guarded replication to encode full replication. Note also that having only input-guarded replication would not have prevented any of the errors described in section 4.

Proposition 1 (Independence of parameters)**.** *If $n,n',s,s' \# \mathrm{n}(P)$ and all are unique for the translation, then $[\![P]\!]_{n,v} \mid !N(x,z,v,s) \sim^{\mathrm{fn}(P)} [\![P]\!]_{n',v} \mid !N(x,z,v,s')$.*

This follows from the fact that the translation only generates finitely many names, say, of the structure $\mathscr{N}_{\lceil\,\rceil}$, so we can construct a finite substitution $\sigma_t : \mathscr{N}_{\lceil n\rceil} \to \mathscr{N}_{\lceil n'\rceil}$ and simply apply it to $[\![P]\!]_{n,v}$ to obtain $[\![P]\!]_{n',v}$. Then as we know that $n,n',s,s' \# \mathrm{n}(P)$, and by construction $x \notin \mathrm{fn}(P)$ for each $x \in \mathscr{N}_{\lceil s\rceil} \cup \mathscr{N}_{\lceil s'\rceil}$, none of these names can be observed by the $\downarrow^{\mathrm{fn}(P)}$ predicate, so they cannot be used to distinguish the two processes. A similar argument can then be made for the name server and the two namespaces ${}^{+}\mathscr{N}_{\lceil s\rceil}$ and ${}^{+}\mathscr{N}_{\lceil s'\rceil}$ generated by it at runtime.

Next, we formulate a (mostly) standard result relating substitution in the two calculi:

Proposition 2 (Substitution)**.** *Let $\sigma_s \triangleq \{u/w\}$ denote substitution in the π-calculus, and let $\sigma_t \triangleq \{u/w\}$ denote substitution in the ρ-calculus. Then $[\![P\sigma_s]\!]_{n,v} = [\![P]\!]_{n,v}\sigma_t$ if $u,w \# P,n,v,\mathscr{N}_{\lceil n\rceil}$.*

This is proved by induction in the clauses of the translation. The condition $u,w\#P,n,v,\mathscr{N}_{\lceil n\rceil}$ ensures that the substitution cannot touch any of the names created by the translation, which is reasonable, since the substitutions we care about should derive from communications in the π-calculus, and not from some of the 'internal' reductions in the ρ-calculus that are used to simulate replication or requests for new names.

Our next result establishes that our translation preserves observability of subjects, as long as we restrict observations to the set of free names in P:

Proposition 3 (Weak observational correspondence)**.** *Let $\Downarrow^{\mathscr{N}}$ be the least predicate such that $[\![S]\!] \Downarrow^{\mathscr{N}} \widehat{n}$ holds if either of the following conditions are satisfied:*

1. *if $S = S_1 \mid S_2$ and $[\![S_1]\!] \Downarrow^{\mathscr{N}} \widehat{n} \vee [\![S_2]\!] \Downarrow^{\mathscr{N}} \widehat{n}$*
2. *if $S \neq S_1 \mid S_2$ and $[\![S]\!]_{n,v} \mid !N(x,z,v,s) \to^* T' \wedge T' \downarrow^{\mathscr{N}} \widehat{n}$*

Then for any x, $P \downarrow^{\mathrm{fn}(P)} \widehat{x} \iff [\![P]\!] \Downarrow^{\mathrm{fn}(P)} \widehat{x}$.

This is proved by induction in the clauses of the translation. Note that we purposefully restrict the weak observation predicate to only allow reductions involved in replication and requesting a fresh name from the name server; i.e. by splitting it directly over parallel compositions rather than allowing them to first interact. This is necessary for proving the implication from right to left in Proposition 3, since reductions might otherwise expose more names that are not immediately observable in the source terms. This restriction can be lifted if we replace $\downarrow$ by $\Downarrow$ in the π-calculus, but we prefer this slightly more complicated formulation to illustrate that observability is strictly preserved, in the sense that any auxiliary steps required in the ρ-calculus are 'internal,' deriving either from a replication step, a request for a new name, or from the name server as it moves from its initial state to its ready state, and neither of these are observable by the $\downarrow^{\mathrm{fn}(P)}$ predicate.

Next we show that the translation preserves the semantic meaning of the source program:

Proposition 4 (Operational correspondence)**.** $P \to^* P' \iff [\![P]\!] \to^* \approx^{\mathrm{fn}(P)} [\![P']\!]$.

This proof can be split into two parts. For the forward direction (completeness) we can actually show the stronger statement that $P \to P' \implies [\![P]\!] \to \to^* \sim^{\mathrm{fn}(P)} [\![P']\!]$ by induction in the reduction semantics of the π-calculus, as every reduction in the π-calculus is matched by one or more steps in the ρ-calculus. The proof often relies on Proposition 2 for the cases of communication, replication and $(\nu x)P$, and on Proposition 1 when we translate the reduct of the π-calculus term as this often induces a slightly different form on the parameters.

For the other direction (soundness) we can only prove the weaker form $[\![P]\!] \to^* T' \implies \exists P'.P \to^* P' \wedge T' \approx^{\mathrm{fn}(P)} [\![P']\!]$, due to the extra reductions deriving from the name server, replication, or requests for

new names. Thus we proceed by induction in the reduction sequence, and often again making use of Proposition 1.

Having only the weaker form of completeness, with $\rightarrow^*$ instead of $\rightarrow$, of course means that this statement in itself is not enough to verify that the translation does not introduce divergence. We therefore prove this separately:

Proposition 5 (Divergence reflection)**.** $[\![P]\!] \rightarrow^\omega \implies P \rightarrow^\omega$.

We show this by induction in the clauses of the translation function. The matter is made easier by the fact that a reduction sequence related to the name server, requests for new names, or unfolding replication, is always of finite length: the name server takes two steps to evolve from its initial state to its ready state, where it blocks until it receives a request; serving a request requires two steps, and then two further steps to return to its ready state; and input-guarded replication takes a single step to unfold once, after which it blocks until it receives an input.

7 A separation result

The ρ-calculus can encode the π-calculus, as we saw in the previous section. However, the converse does not hold. Under some general assumptions about the behavioural equivalence $\simeq$ used in the target language, we can show that there cannot be an encoding of the ρ-calculus into the π-calculus that satisfies our validity criteria from Definition 3. This result relies on a simple observation about substitution in the π-calculus, namely that reduction is preserved under substitution:

Lemma 3. *Let $\sigma_t = \{x/n\}$ be a substitution in the π-calculus, with $n \in \mathrm{fn}(P)$ and $x\#P$. Then $P \rightarrow P' \implies P\sigma_t \rightarrow P'\sigma_t$.*

This can easily be shown by induction in the semantic rules, and then with an extra induction in structural congruence for the [π-STRUCT] rule.

Next, we consider our requirements for the notion of behavioural equivalence: First of all, $\simeq$ should obviously be an equivalence relation. Secondly, it should in some sense preserve the semantics of the processes it equates: as we are here working in a reduction system, it should at least preserve reductions and observability, and it should be preserved under substitution:

Definition 4 (Behavioural equivalence requirements)**.** *We require that $\simeq$ be at least an equivalence relation over π-terms satisfying the following:*

1. $P_1 \simeq P_2 \wedge P_1 \rightarrow^* P_1' \implies \exists P_2' \,.\, P_2 \rightarrow^* P_2' \wedge P_1' \simeq P_2'$
2. $P_1 \simeq P_2 \wedge P_1 \Downarrow \widehat{x} \implies P_2 \Downarrow \widehat{x}$
3. $P_1 \simeq P_2 \implies P_1\sigma_t \simeq P_2\sigma_t$

The requirements suggest that $\simeq$ should be at least weak, barbed congruence, which does not seem too demanding. However, we prefer to keep the formulation general, without committing to one specific notion of behavioural equivalence, to emphasise that other, stronger choices are also possible. The following result will then hold for any such choice:

Theorem 1 (Separation)**.** *If $\simeq$ satisfies the requirements of Definition 4, then there is no encoding of the ρ-calculus into the π-calculus satisfying the criteria of Definition 3.*

Proof. Assume to the contrary that there exists a translation $[\![\ ]\!]_N : \mathscr{P}_\rho \rightarrow \mathscr{P}_\pi$ satisfying the criteria of Definition 3. We show that this leads to a contradiction. Firstly, let $u \triangleq \ulcorner\urcorner x_1 \urcorner \mid \urcorner x_2 \ulcorner\urcorner$, and consider the processes P and P' where

$$P \triangleq P_1 \mid P_2 \qquad P_1 \triangleq a\langle\!|\ulcorner x_1 \urcorner \mid \urcorner x_2 \ulcorner|\!\rangle \qquad P_2 \triangleq a(n).n\langle\!|\mathbf{0}|\!\rangle \qquad P' \triangleq u\langle\!|\mathbf{0}|\!\rangle$$

Thus $P = a\langle\!|\ulcorner x_1 \urcorner \mid \ulcorner x_2 \urcorner|\!\rangle \mid a(n).n\langle\!|\mathbf{0}|\!\rangle$ and clearly $P \not\downarrow u$ and $u \notin \mathrm{fn}(P)$, but $P \to \ulcorner\ulcorner x_1 \urcorner \mid \ulcorner x_2 \urcorner\ulcorner\langle\!|\mathbf{0}|\!\rangle = u\langle\!|\mathbf{0}|\!\rangle = P'$ and $P' \downarrow u$.

Consider now the substitution $\sigma_t \triangleq \{m/\varphi(u)\}$ for some fresh name m, i.e. $m \neq \varphi(u)$ and with $m \notin \mathrm{fn}([\![P]\!]_N)$. $P \to P'$ gives, by criterion 3 (operational completeness), that $[\![P]\!]_N \to^* T'$ and $T' \simeq [\![P']\!]_{N'}$ for some T' and N' derivable from N. By criterion 2 (substitution invariance), $\sigma_t(\varphi(u)) = m$ implies $\exists\sigma_s.\varphi(\sigma_s(u)) = m$, so we can combine σ_s with the observability predicate. By criterion 4 (observational correspondence), since $P' \not\downarrow \sigma_s(u)$, we therefore also have that $[\![P']\!]_N \not\Downarrow m$. This establishes that

$$[\![P]\!]_N \to^* T' \wedge T' \simeq [\![P']\!]_{N'} \wedge [\![P']\!]_{N'} \not\Downarrow m$$

as expected. By requirement 2 in Definition 4, since $[\![P']\!]_{N'} \simeq T'$, it must therefore also be the case that $T' \not\Downarrow m$, and hence that $[\![P]\!]_N \not\Downarrow m$.

Now consider the term $[\![P]\!]_N\sigma_t$: Lemma 3 yields $[\![P]\!]_N\sigma_t \to^* T'\sigma_t \simeq [\![P']\!]_{N'}\sigma_t$, and by criterion 2 (substitution invariance) $[\![P']\!]_{N'}\sigma_t \simeq [\![P'\sigma_s]\!]$. As we know that $P' \downarrow u$, this implies that $P'\sigma_s \downarrow \sigma_s(u)$, which again implies that $[\![P'\sigma_s]\!]_{N'} \Downarrow \sigma_t(\varphi(u))$, which implies $[\![P']\!]_{N'}\sigma_t \Downarrow m$. This establishes that

$$[\![P]\!]_N\sigma_t \to^* T'\sigma_t \wedge T'\sigma_t \simeq [\![P']\!]_{N'}\sigma_t \wedge [\![P']\!]_{N'}\sigma_t \Downarrow m$$

again, as expected. By requirement 2 in Definition 4, since $[\![P']\!]_{N'}\sigma_t \simeq T'\sigma_t$, it must therefore also be the case that $T'\sigma_t \Downarrow m$, and hence that $[\![P]\!]_N\sigma_t \Downarrow m$.

However, consider now the effect of applying the substitution $[\![P]\!]_N\sigma_t$. By criterion 1 (compositionality), we have that

$$[\![P]\!]_N\sigma_t = C\sigma_t \mid [\![P_1]\!]_{N_1}\sigma_t \mid [\![P_2]\!]_{N_2}\sigma_t = C \mid [\![P_1]\!]_{N_1}\sigma_t \mid [\![P_2]\!]_{N_2}\sigma_t$$

where we can eliminate the substitution from C, since $\varphi(u) \notin N \cup N_1 \cup N_2$, as this immediately would violate criterion 6 (parameter independence); and as we know that $u \notin \mathrm{fn}(P)$, we therefore also know that $\varphi(u) \notin \mathrm{fn}(C)$, since C at most can contain a subset of the (φ-translated) free names of the process and the parameters. Thus the substitution has no effect on C.

Now consider the two subterms $[\![P_1]\!]_{N_1}\sigma_t$ and $[\![P_2]\!]_{N_2}\sigma_t$. By criterion 2, $[\![P_1]\!]_{N_1}\sigma_t \simeq [\![P_1\sigma_s]\!]_{N_1}$ and $[\![P_2]\!]_{N_2}\sigma_t \simeq [\![P_2\sigma_s]\!]_{N_2}$, but when we apply the substitution, we get that

$$P_1\sigma_s = (a\langle\!|\ulcorner x_1 \urcorner \mid \ulcorner x_2 \urcorner|\!\rangle)\,\sigma_s = a\langle\!|\ulcorner x_1 \urcorner \mid \ulcorner x_2 \urcorner|\!\rangle = P_1 \qquad P_2\sigma_s = (a(n).n\langle\!|\mathbf{0}|\!\rangle)\,\sigma_s = a(n).n\langle\!|\mathbf{0}|\!\rangle = P_2$$

since obviously $u \notin \mathrm{fn}(P_1)$ and $u \notin \mathrm{fn}(P_2)$, so the substitution has no effect on any of the subterms. Thus

$$C \mid [\![P_1\sigma_s]\!]_{N_1} \mid [\![P_2\sigma_s]\!]_{N_2} = C \mid [\![P_1]\!]_{N_1} \mid [\![P_2]\!]_{N_2}$$

and hence $[\![P\sigma_s]\!]_N = [\![P]\!]_N$. By criterion 2 (substitution invariance) $[\![P\sigma_s]\!]_N \simeq [\![P]\!]_N\sigma_t$, and thus we have that $[\![P]\!]_N\sigma_t \simeq [\![P]\!]_N$. This then yields the desired contradiction, since, as established above, $[\![P]\!]_N\sigma_t \Downarrow m$ but $[\![P]\!]_N \not\Downarrow m$, whilst by requirement 2 of Definition 4 it must hold that $[\![P]\!]_N\sigma_t \simeq [\![P]\!]_N \wedge [\![P]\!]_N\sigma_t \Downarrow m \implies [\![P]\!]_N \Downarrow m$. □

The above proof exploits the reflective capability of the ρ-calculus to create new, *free* names at runtime, which are therefore also observable and substitutable. Thus, a substitution can affect the reduct of a process, without affecting the process itself, if the reduction step creates a new name. This cannot be mimicked in the π-calculus, where names have no structure and cannot be composed at runtime. Any new *free* name appearing at runtime can therefore only come from the translation parameters, since it cannot come from the source term; but this would then violate the criterion of parameter independence,

since we would then have to choose the parameters such that they correspond to the names that will be created at runtime.

This result does not directly depend on the higher-order characteristics of the ρ-calculus, and adding higher-order behaviour to the π-calculus would not suffice to enable it to encode the ρ-calculus. In [18], Sangiorgi gave an encoding of the Higher-Order π-calculus, HOπ, in the π-calculus. His encoding also satisfies our criteria from Definition 3, and we therefore also have the following result:

Corollary 1. *There is no encoding of the ρ-calculus into* HOπ *satisfying the criteria of Definition 3, when $\simeq$ satisfies the requirement in Definition 4.*

Indeed, if such an encoding existed, we could compose it with the encoding of HOπ into the π-calculus, to obtain an encoding of the ρ-calculus into the π-calculus, in contradiction of Theorem 1. This also indicates that the key feature of the ρ-calculus which cannot be represented in the π-calculus, is not its higher-order characteristics per se, but rather its capability for reflection, which gives it higher-order characteristics as a by-product.

8 Related works

The issues of encodability and assessing the relative expressiveness of various process calculi has been considered by several authors; in particular, Gorla [8] proposed a framework for reasoning about encodability and separation w.r.t. a set of criteria that also served as inspiration for the criteria used in the present paper. Towards the end of the paper, Gorla also discusses some of the difficulties involved in formulating a *general* framework for encodability in the presence of parameters, which particularly pertain to the question of which language the names belong to (the source or the target). In the present case, the answer is clearly the target language, which is further underscored by our restrictions on observability and compositionality; i.e. that the parameters should not be observable in the source term; and that, for each recursive call to the translation function, the parameters should be derivable from the initial set. Furthermore, we have added the criterion of parameter independence. We believe that such a criterion will generally be necessary for encodings that allow the set of parameters to 'evolve' or be updated in some structured way during the course of the translation, which seems particularly likely when we are working with structured names or terms. More recently, van Glabbeek [7] has also proposed a definition of a valid encoding, which he derives from a notion of a semantic equivalence or preorder, rather than basing it on a list of commonly agreed-upon criteria (as we have done in the present paper, following Gorla). However, this work also does not consider parametrised translations.

Also related is the work by Carbone and Maffeis [4] on expressivity of polyadic synchronisation. Their ${}^e\pi$-calculus substitutes names for names (as in the π-calculus), but allows n-ary vectors of names $x_1 \cdot \ldots \cdot x_n$ of arbitrary length $n \geq 0$ to appear in *subject* position of input/output prefixes, and subjects must then match on all n names to yield a reduction. Thus name vectors can be altered at runtime, but they cannot grow in length as in the ρ-calculus. However, we could conceive of a (purposefully ill-sorted) variant of ${}^e\pi$ that would allow entire vectors of names to be substituted for single names, thereby allowing new vectors of increasing length to be composed at runtime. We do not know if such a calculus could encode the ρ-calculus, but we suspect that it might, if equipped with an appropriate notion of name equivalence.

Another approach to using structured terms as names is given by Bengtson et al. [2, 3] and Parrow et al. [15] in their work on Ψ-calculi, which is based on the theory of nominal sets and datatypes by Gabbay and Pitts [6]. Ψ-calculi allow both subjects and objects to be terms from an arbitrary nominal datatype,

and with substitution of terms for names. This enables runtime composition of terms, and, notably, the ρ-calculus *can* be instantiated as a (higher-order) Ψ-calculus, as the present author and others have shown in [1].

9 Conclusion

The original ρ-calculus paper [11] by Meredith and Radestock raises some interesting questions about the nature of names in process calculi. By including name generation in the language, it forces any process to give an explicit account of the source of any fresh names required during its execution, whilst this is entirely implicit in the π-calculus with the $(\nu x)P$ operator. This adds a degree of realism to the ρ-calculus, which may be relevant from an implementation perspective, but also requires some extra care when we wish to reason about it formally. For example, Meredith and Radestock attempted to show that the π-calculus can be interpreted in the ρ-calculus, but their encoding did not properly account for the invariant that must hold for the names used as parameters in their encoding; i.e. that the parameters always refer to the most recently replicated names, leading to two errors that invalidate their correctness result. The purpose of the present paper has been to describe these errors and then give a new encoding of the π-calculus, for which we have shown correctness w.r.t. a set of criteria for encodability close to those proposed by Gorla [8]. The main difference is that we here use a parametrised translation, and we therefore had to take parameters into account in our criteria. This seems unavoidable when we are working with a calculus with structured names like the ρ-calculus, where all names are global and cannot be declared at runtime.

Our encoding works, modulo the criteria in Definition 3; yet it may not be an entirely satisfactory solution in at least one regard: the name server acts as a single, central source of fresh names. If we consider the scenarios one might wish to *model* in the π-calculus, having such a single central process might be acceptable for e.g. models of programs running on a single computer, or models of client-server systems with a star topology. However, for distributed systems with a different network topology, the translation would not yield an adequate representation. Thus, the encoding may preserve the semantics of a program, but not necessarily the intuitions underlying its structure. We could instead conceive of a more elaborate encoding, where e.g. each replication also instantiates its own copy of a name server to service the replicated processes. This would be closer to the intention in the encoding by Meredith and Radestock; but as we have seen, one would then have to be careful to ensure that each replica of the name server will generate a distinct namespace to avoid the possibility of a name clash. This could be achieved by letting each replica first request fresh names for all its parameters, including the namespace root s which must then be composed or otherwise shifted into a new namespace. Yet this creates a scaffolding problem, where, in order to instantiate a new source of fresh names, one must first have a source of fresh names. It does not remove the need for an initial, 'top level' instance of the name server. These considerations illustrate some of the difficulties involved in working with, and reasoning about, structured names with global visibility. None of these problems are present in the π-calculus, yet any implementation of a π-calculus program would need to include a solution to the problem of obtaining fresh names. In the words of Meredith and Radestock [11], the π-calculus does not provide a 'theory of names.'

We have also shown that the π-calculus cannot encode the ρ-calculus in a way that satisfies the same criteria, modulo some requirements on the notion of behavioural equivalence $\simeq$ used in Definition 4. The key to this separation result seems precisely to be the ability of the ρ-calculus to create new *free* names at runtime, which cannot be mimicked in the π-calculus. This ability is a consequence of *reflection* in the

ρ-calculus, which also gives it higher-order characteristics as a by-product. In a process-calculus setting where computation is modelled as communication, higher-order behaviour appears as just a special case of reflection, where processes (code) are transmitted without modification. Thus, the separation result is also interesting in light of a remark by Sangiorgi regarding the encodability of HOπ into the π-calculus. He notes that this *"[...] proves that the first-order paradigm, being by far simpler, should be taken as basic. Such a conclusion takes away the interest in the opposite direction, namely the representability of the π-calculus within a language using purely communications of agents ..."* [17, p. 8]. But as we have seen, this does not seem to hold in the more general case where higher-order characteristics derive from the capability of reflection. The ρ-calculus purely uses communication of agents (processes), because names and processes are the same thing.

Acknowledgements

The author wishes to thank Hans Hüttel, Bjarke B. Bojesen and Alex R. Bendixen for many discussions of the ρ-calculus, and Luca Aceto and the anonymous reviewers for their numerous and invaluable comments on earlier drafts of this paper.

References

[1] Alex Rønning Bendixen, Bjarke Bredow Bojesen, Hans Hüttel & Stian Lybech (2022): *A Generic Type System for Higher-Order Ψ-calculi*. *Electronic Proceedings in Theoretical Computer Science* 368, Open Publishing Association, pp. 43–59, doi:10.4204/EPTCS.368.3.

[2] Jesper Bengtson, Magnus Johansson, Joachim Parrow & Björn Victor (2009): *Psi-calculi: Mobile processes, nominal data, and logic*. In: *2009 24th Annual IEEE Symposium on Logic In Computer Science*, IEEE, pp. 39–48, doi:10.1016/S1571-0661(05)80361-5.

[3] Jesper Bengtson, Magnus Johansson, Joachim Parrow & Björn Victor (2011): *Psi-calculi: a framework for mobile processes with nominal data and logic*. *Logical Methods in Computer Science* Volume 7, Issue 1, doi:10.2168/LMCS-7(1:11)2011. Available at `https://lmcs.episciences.org/696`.

[4] Marco Carbone & Sergio Maffeis (2003): *On the Expressive Power of Polyadic Synchronisation in Pi-Calculus*. *Nordic Journal of Computing* 10(2), pp. 70–98, doi:10.1016/S1571-0661(05)80361-5.

[5] Cédric Fournet & Georges Gonthier (2000): *The Join Calculus: A Language for Distributed Mobile Programming*. In: *International Summer School on Applied Semantics*, Springer, pp. 268–332, doi:10.1007/3-540-45699-6_6.

[6] Murdoch Gabbay & Andrew Pitts (2002): *A New Approach to Abstract Syntax with Variable Binding*. *Formal Asp. Comput.* 13, pp. 341–363, doi:10.1007/s001650200016.

[7] Rob van Glabbeek (2018): *A theory of encodings and expressiveness*. In: *International Conference on Foundations of Software Science and Computation Structures*, Springer, Cham, pp. 183–202, doi:10.1007/978-3-319-89366-2_10.

[8] Daniele Gorla (2010): *Towards a unified approach to encodability and separation results for process calculi*. *Information and Computation* 208(9), pp. 1031–1053, doi:10.1016/j.ic.2010.05.002.

[9] Daniele Gorla & Uwe Nestmann (2014): *Full abstraction for expressiveness: history, myths and facts*. *Mathematical Structures in Computer Science* 26, pp. 639 – 654, doi:10.1017/S0960129514000279.

[10] Stian Lybech (2022): *Encodability and Separation for a Reflective Higher-Order Calculus*. Technical Report, Department of Computer Science, Reykjavík University. Available at `http://icetcs.ru.is/stian/2022/reflection_encodability2022techreport.pdf`.

[11] L.G. Meredith & Matthias Radestock (2005): *A Reflective Higher-order Calculus*. *Electronic Notes in Theoretical Computer Science* 141(5), pp. 49 – 67, doi:10.1016/j.entcs.2005.05.016. Proceedings of the Workshop on the Foundations of Interactive Computation (FInCo 2005).

[12] Robin Milner (1993): *The Polyadic π-Calculus: a Tutorial*. In: *Logic and Algebra of Specification*, Springer Berlin Heidelberg, pp. 203–246, doi:10.1007/978-3-642-58041-3_6.

[13] Robin Milner, Joachim Parrow & David Walker (1992): *A calculus of mobile processes, I*. *Information and Computation* 100(1), pp. 1–40, doi:10.1016/0890-5401(92)90008-4.

[14] Joachim Parrow (2001): *An introduction to the π-calculus*. In: *Handbook of Process Algebra*, Elsevier, pp. 479–543, doi:10.1016/B978-044482830-9/50026-6.

[15] Joachim Parrow, Johannes Borgström, Palle Raabjerg & Johannes Åman Pohjola (2014): *Higher-order psi-calculi*. *Mathematical Structures in Computer Science* 24(2), doi:10.1017/S0960129513000170.

[16] Benjamin C. Pierce & David N. Turner (2000): *Pict: a programming language based on the Pi-Calculus*. In: *Proof, Language, and Interaction*, pp. 455–494, doi:10.5555/345868.345924.

[17] Davide Sangiorgi (1993): *Expressing mobility in process algebras: first-order and higher-order paradigms*. Ph.D. thesis, University of Edinburgh. Available at `http://hdl.handle.net/1842/6569`.

[18] Davide Sangiorgi (1993): *From π-calculus to higher-order π-calculus — and back*. In M. C. Gaudel & J. P. Jouannaud, editors: *TAPSOFT'93: Theory and Practice of Software Development*, Springer Berlin Heidelberg, pp. 151–166, doi:10.1007/3-540-56610-4_62.

[19] Brian Cantwell Smith (1982): *Procedural Reflection in Programming Languages*. Ph.D. thesis, Massachusetts Institute of Technology. Available at `http://hdl.handle.net/1721.1/15961`.

[20] David N. Turner (1996): *The Polymorphic Pi-calculus: Theory and Implementation*. Ph.D. thesis, University of Edinburgh, UK. Available at `https://hdl.handle.net/1842/395`.

On the Expressiveness of Mixed Choice Sessions

Kirstin Peters
Universität Augsburg
Germany
kirstin.peters@uni-a.de

Nobuko Yoshida
Imperial College London
UK
n.yoshida@imperial.ac.uk

Session types provide a flexible programming style for structuring interaction, and are used to guarantee a safe and consistent composition of distributed processes. Traditional session types include only one-directional input (external) and output (internal) guarded choices. This prevents the session-processes to explore the full expressive power of the π-calculus where the mixed choices are proved more expressive than the (non-mixed) guarded choices. To account this issue, recently Casal, Mordido, and Vasconcelos proposed the binary session types with mixed choices (CMV^+). This paper carries a surprising, unfortunate result on CMV^+: in spite of an inclusion of unrestricted channels with mixed choice, CMV^+'s mixed choice is rather separate and not mixed. We prove this negative result using two methodologies (using either the leader election problem or a synchronisation pattern as distinguishing feature), showing that there exists no good encoding from the π-calculus into CMV^+, preserving distribution. We then close their open problem on the encoding from CMV^+ into CMV (without mixed choice), proving its soundness and thereby that the encoding is good up to coupled similarity.

1 Introduction

Starting with the landmark result by Palamidessi in [21] and followed up by results such as [20, 22, 10, 26, 28, 29] it was shown that the key to the expressive power of the full π-calculus in comparison to its sub-calculi such as e.g. the asynchronous π-calculus is *mixed choice*.

Mixed choice in the π-calculus is a choice construct that allows to choose between inputs and outputs. In contrast, e.g. *separate choices* are constructed from either only inputs or only outputs. The additional expressive power of mixed choice relies on its ability to rule out alternative options of the opposite nature, i.e., a term can rule out its possibility to perform an input by doing an output, whereas without mixed choice inputs can rule out alternative inputs only and outputs may rule out only alternative outputs.

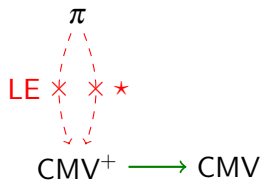
To compare calculi with different variants of choice, we try to build an encoding or show that no such encoding exists [3, 23]. The existence of an encoding that satisfies relevant criteria shows that the target language is expressive enough to emulate the behaviours in the source language. Gorla [10] and others [23, 31] introduced and classified a set of general criteria for encodability which are syntax-agnostic [10, 31]: they are now commonly used for claiming expressiveness of a given calculus, defining important features which a "good encoding" should satisfy. These include *compositionality* (homomorphism), *name invariance* (bijectional renaming), sound and complete *operational correspondence* (the source and target can simulate each other), *divergence reflection* (the target diverges only if the source diverges), *observability* (barb-sensitiveness), and *distributability* preservation (the target has the same degree of distribution as the source). Conversely, a *separation result*, i.e., the proof of the absence of an encoding with certain criteria, shows that the source language can represent behaviours that can*not* be expressed in the target. This paper gives a fresh look at expressiveness of typed π-calculi, focusing on choice constructs of *session types*.

V. Castiglioni and C. A. Mezzina (Eds): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics 2022 (EXPRESS/SOS 2022).
EPTCS 368, 2022, pp. 113–130, doi:10.4204/EPTCS.368.7

Session types [13, 37] specify and constrain the communication behaviour as a protocol between components in a system. A session type system excludes any non-conforming behaviour, statically preventing type and communication errors (i.e., mismatch of choice labels). Several languages now have session-type support via libraries and tools [36, 1]. As the origin of session types is Linear Logic [11], traditional session types include only one-directional input (external) and output (internal) guarded choices. To explore the full expressiveness of mixed choice from the π-calculus, recently Casal, Mordido, and Vasconcelos proposed the binary session types with mixed choices called *mixed sessions* [6]. We denote their calculus by CMV^+. Mixed sessions include a mixture of branchings (labelled input choices) and selections (labelled output choices) at the same *linear* channel or *unrestricted* channel. This extension gives us many useful and typable *structured* concurrent programming idioms which consist of both unrestricted and linear non-deterministic choice behaviours. We show that in spite of its practical relevance, mixed sessions in CMV^+ are *strictly less expressive* than mixed choice in the π-calculus even with an unrestricted usage of choice channels.

This result surprised us. We would have expected that using mixed choice with an unrestricted choice channel results into a choice construct comparable to choice in the π-calculus. But, as we show in the following, mixed choice in CMV^+ cannot express essential features of mixed choice in the π-calculus. First we observe that mixed sessions are not expressive enough to solve leader election in symmetric networks. Remember that it was leader election in symmetric networks that was used to show that mixed choice is more expressive than separate choice in the π-calculus (see [21]). Second we observe that mixed sessions cannot express the synchronisation pattern $\star$. Synchronisation patterns were introduced in [31] to capture the amount of synchronisation that can be expressed in distributed systems. The synchronisation pattern $\star$ was identified in [31] as capturing exactly the amount of synchronisation introduced with mixed choice in the π-calculus. Finally, we have a closer look at the encoding from CMV^+ into CMV presented in [6]. CMV is the variant of session types that is extended in [6] with a mixed-choice-construct in order to obtain CMV^+, i.e., CMV has traditional branching and selection but not their mix. As it is the case for many variants of session types, CMV can express separate choice but has no construct for mixed choice. By analysing this encoding, we underpin our claim that mixed choice in CMV^+ is not more expressive than separate choice in the π-calculus.

Our contributions are summarised in the picture on the right. In § 3 we prove that there exists no good encoding from the π-calculus (with mixed choice) into CMV^+, where we use the leader election problem by Palamidessi in [21] (LE) as distinguishing feature (the first - -×- →). In § 4 we reprove this result using the *synchronisation pattern* $\star$ from [31] instead as distinguishing feature (the second - -×- →). Then we prove soundness of the encoding presented in [6] closing their open problem in § 5 (⟶). By this encoding source terms in CMV^+ and their literal translations in CMV are related by *coupled similarity* [25], i.e., CMV^+ is encoded into CMV up to coupled similarity. From the separation results in § 3 and § 4 and the encoding into session types with separate choice in § 5 we conclude that *mixed sessions in [6] can express only separate choice*.



To make our paper readable, we focus on presenting our results with intuitive, self-contained explanations. In particular, we simplify the languages CMV^+ and CMV for the discussion in this paper and omit their type systems. However, the proofs are carried out on the original definitions of the languages from [6]. We include complete proofs of all the statements in this paper and the technical details of the notions from the literature that we use in [34].

2 Technical Preliminaries: Mixed Sessions and Encodability Criteria

This section gives a summary of the π-calculus, CMV^+, CMV, and encodability criteria.

Assume a countably-infinite set $\mathscr{N}$ of *names*. For the π-calculus we additionally assume a set $\{\overline{y} \mid y \in \mathscr{N}\}$ of *co-names*. Let $\tau \notin \mathscr{N} \cup \{\overline{y} \mid y \in \mathscr{N}\}$.

The *syntax* of a process calculus is usually defined by a context-free grammar defining operators. We use $P, Q, \ldots$ to range over process terms. The arguments of a term P that are again process terms are called *subterms* of P. Terms that appear as subterm underneath some (action) prefix are called *guarded*, because the guarded subterm cannot be executed before the guarding action has been performed. Also conditionals, such as if-then-else-constructs, guard their respective subterms. *Expressions* are constructed from variables, unit, and standard boolean operators. We assume an evaluation function $\mathsf{eval}(\cdot)$ that evaluates expressions to *values*.

We assume that the *semantics* is given as an *operational semantics* consisting of inference rules defined on the operators of the language [35]. For many process calculi, the semantics is provided in two forms, as *reduction semantics* and as *labelled transition semantics*. We assume that at least the reduction semantics $\longmapsto$ is given as part of the definition, because its treatment is easier in the context of encodings. A *(reduction) step* is written as $P \longmapsto P'$. If $P \longmapsto P'$, then P' is called *derivative* of P. Let $P \longmapsto$ (or $P \not\longmapsto$) denote the existence (absence) of a step from P, and let $\Longmapsto$ denote the reflexive and transitive closure of $\longmapsto$. A sequence of reduction steps is called a *reduction*. We write $P \longmapsto^{\omega}$ if P has an infinite sequence of steps. We also use *execution* to refer to a reduction starting from a particular term. A process that cannot reduce is called *stuck*.

The application $P\sigma$ of a substitution $\sigma = \{{}^{y_1}/_{x_1}, \ldots, {}^{y_n}/_{x_n}\}$ on a term is defined as the result of simultaneously replacing all free occurrences of x_i by y_i for $i \in \{1, \ldots, n\}$, possibly applying α-conversion to avoid capture or name clashes. For all names in $\mathscr{N} \setminus \{x_1, \ldots, x_n\}$ the substitution behaves as the identity mapping.

2.1 Process and Session Calculi

The π-calculus was introduced by Milner, Parrow, and Walker in [19] and is one of the most well-known process calculi. We consider a variant of the π-calculus with mixed guarded choice and replication but without matching (as in [21]). This variant is often called the synchronous or full π-calculus. *Mixed sessions* are a variant of binary session types introduced by Casal, Mordido, and Vasconcelos in [6] with a choice construct that combines prefixes for sending and receiving. We denote this language as CMV^+. CMV is the session type variant on which CMV^+ is based. A central idea of CMV^+ (and CMV) is that channels are separated in two *channel endpoints* and that interaction is by two processes acting on the respective different ends of such a channel.

The syntax of the π-calculus, CMV^+, and CMV is given as:

$$\mathscr{P}_{\pi}\colon P ::= \sum_{i\in \mathrm{I}} \alpha_i.P_i \mid (\nu x)P \mid P \mid P \mid\ !P \qquad \alpha ::= y(x) \mid \overline{y}z \mid \tau$$

$$\mathscr{P}_{\mathsf{CMV}^+}\colon P ::= y \sum_{i\in \mathrm{I}} M_i \mid P \mid P \mid (\nu yz)P \mid \mathsf{if}\, v\, \mathsf{then}\, P\, \mathsf{else}\, P \mid \mathbf{0} \qquad M ::= \mathrm{l}{*}v.P \qquad * ::= ! \mid ?$$

$$\mathscr{P}_{\mathsf{CMV}}\colon P ::= y!v.P \mid y?xP \mid x \triangleleft \mathrm{l}.P \mid x \triangleright \{\mathrm{l}_i : P_i\}_{i\in \mathrm{I}} \mid P \mid P \mid (\nu yz)P \mid \mathsf{if}\, v\, \mathsf{then}\, P\, \mathsf{else}\, P \mid \mathbf{0}$$

A choice $\sum_{i\in \mathrm{I}} \alpha_i.P_i$ in $\mathscr{P}_{\pi}$ offers for each i in the index set I a subterm guarded by some action prefix α_i, where α_i is an input action $y(x)$, an output action $\overline{y}z$, or the internal action τ. In contrast choice $y \sum_{i\in \mathrm{I}} M_i$ in CMV^+ operates on a single channel endpoint. In [6] a choice is declared as either linear

(lin) or unrestricted (un), where the latter introduces recursion in the calculus. We omit these qualifiers to simplify the presentation. However, the proofs are carried out on the languages CMV^+ and CMV as given by [6] (see [34]). A branch $\mathrm{l}*v.P$ specifies a label l, a polarity $*$ (! for sending or ? for receiving), a value in output actions or a variable for input actions, and a continuation P. We abbreviate the empty sum by $\mathbf{0}$ and we often separate summands by $+$. The remaining are: restriction $(\nu x)P$ and $(\nu yz)P$, parallel composition $P \mid P$, replication $!P$, conditional $\mathsf{if}\, v\, \mathsf{then}\, P\, \mathsf{else}\, P$, output $y!v.P$, input $y?xP$, selection $x \triangleleft \mathrm{l}.P$, and branching $x \triangleright \{\mathrm{l}_i : P_i\}_{i\in I}$.

The semantics of the languages is given by the axioms:

$$\begin{array}{lc}
\pi\text{:} & \overline{y}z.P + R \mid y(x).Q + N \longmapsto P \mid Q\{^z/_x\} \qquad \tau.P + R \longmapsto P \\
\mathsf{CMV}^+/\mathsf{CMV}\text{:} & \mathsf{if}\, \mathsf{true}\, \mathsf{then}\, P\, \mathsf{else}\, Q \longmapsto P \qquad \mathsf{if}\, \mathsf{false}\, \mathsf{then}\, P\, \mathsf{else}\, Q \longmapsto Q \\
\mathsf{CMV}^+\text{:} & (\nu yz)(y(\mathrm{l}!v.P + M) \mid z(\mathrm{l}?x.Q + N) \mid R) \longmapsto (\nu yz)(P \mid Q\{^v/_x\} \mid R) \\
\mathsf{CMV}\text{:} & (\nu yz)(y!v.P \mid z?xQ \mid R) \longmapsto (\nu yz)(P \mid Q\{^v/_x\} \mid R) \\
 & (\nu yz)\big(y \triangleleft \mathrm{l}_j.P \mid z \triangleright \{\mathrm{l}_i : Q_i\}_{i\in I} \mid R\big) \longmapsto (\nu yz)(P \mid Q_j \mid R)
\end{array}$$

and all three calculi share the following rules:

$$\frac{P \longmapsto P'}{P \mid Q \longmapsto P' \mid Q} \qquad \frac{P \longmapsto P'}{(\nu \hat{x})P \longmapsto (\nu \hat{x})P'} \qquad \frac{P \equiv Q \quad Q \longmapsto Q' \quad Q' \equiv P'}{P \longmapsto P'}$$

where $\hat{x}$ consists of either one or two names; and $\equiv$ denotes the standard *structural congruence* from [6] plus a rule for replication in the π-calculus. More precisely, structural congruence is defined as the least congruence that contains α-conversion and satisfies the rules:

$$(\nu \hat{x})\mathbf{0} \equiv \mathbf{0} \qquad (\nu \hat{x})(\nu \hat{y})P \equiv (\nu \hat{y})(\nu \hat{x})P \qquad P \mid (\nu \hat{x})Q \equiv (\nu \hat{x})(P \mid Q) \quad \text{if } \{\hat{x}\} \cap \mathsf{fn}(P) = \emptyset$$
$$(\nu yz)P \equiv (\nu zy)P \quad P \mid \mathbf{0} \equiv P \quad P \mid Q \equiv Q \mid P \quad P \mid (Q \mid R) \equiv (P \mid Q) \mid R \quad !P \equiv P \mid !P$$

The name x is bound in P by $y(x).P$, $\mathrm{l}?x.P$, and $(\nu x)P$, $(\nu xy)P$, or $(\nu yx)P$. All other names are free. We often omit $\mathbf{0}$ and the argument of action prefixes if it is irrelevant, i.e., we write $y.P$ instead of $y(x).P$ if $x \notin \mathsf{fn}(P)$; and $\overline{y}.P$ instead of $\overline{y}z.P$ if for all matching receivers $y(x).Q$ we have $x \notin \mathsf{fn}(Q)$.

A process P emits a barb $\overline{y}$, denoted as $P{\downarrow_{\overline{y}}}$, if P (in the π-calculus) has an unguarded output $\overline{y}z.P'$ on a free channel $y \in \mathsf{fn}(P)$. Similarly, P has a barb y, denoted as $P{\downarrow_y}$, if P (in the π-calculus) has an unguarded input $y(x).P'$ or if P (in $\mathsf{CMV}^+/\mathsf{CMV}$) has an unguarded choice $y \sum_{i\in I} M_i$, output $y!v.P$, input $y?x.P$, selection $y \triangleleft \mathrm{l}.P$, or branching $y \triangleright \{\mathrm{l}_i : P_i\}_{i\in I}$ on a free $y \in \mathsf{fn}(P)$. In CMV^+ and CMV we do not distinguish between input barbs $\downarrow_y$ and output barbs $\downarrow_{\overline{y}}$ but instead have barbs on different channel end points. The term P reaches a barb β (with $\beta = y$ or $\beta = \overline{y}$), denoted as $P{\Downarrow_\beta}$, if there is some P' such that $P \Longmapsto P'$ and $P'{\downarrow_\beta}$.

The type systems of CMV^+ and CMV in [6] ensure that the two endpoints of a channel are dual, e.g. if one endpoint sends the other has to receive. For the expressiveness results on the choice construct proved in this paper, the type system is not crucial. Indeed, our separation result (in both ways to prove it) is carried out on the untyped version of CMV^+. See [6] or [34] for the full typing systems.

Two terms of a language are usually compared using some kind of a behavioural simulation relation. The most commonly known behavioural simulation relation is bisimulation. A relation $\mathscr{R}$ is a bisimulation if any two related processes mutually simulate their respective sequences of steps, such that the derivatives are again related.

Definition 2.1 (Bisimulation)**.** $\mathscr{R}$ is a (weak reduction, barbed) bisimulation if for each $(P,Q) \in \mathscr{R}$:

- $P \Longmapsto P'$ implies $\exists Q'.\ Q \Longmapsto Q' \wedge (P',Q') \in \mathscr{R}$
- $Q \Longmapsto Q'$ implies $\exists P'.\ P \Longmapsto P' \wedge (P',Q') \in \mathscr{R}$
- $P\Downarrow_\beta$ iff $Q\Downarrow_\beta$ for all barbs β

Two terms are bisimilar if there exists a bisimulation that relates them. For a language $\mathscr{L}$, let $\approx_{\mathscr{L}}$ denote bisimilarity on $\mathscr{L}$.

Another interesting behavioural simulation relation is coupled similarity. It was introduced in [25] and discussed e.g. in [2]. It is strictly weaker than bisimilarity. As pointed out in [25], in contrast to bisimilarity it essentially allows for intermediate states (see § 5). Each symmetric coupled simulation is a bisimulation.

Definition 2.2 (Coupled Simulation)**.** A relation $\mathscr{R}$ is a (weak reduction, barbed) coupled simulation if for each $(P,Q) \in \mathscr{R}$:

- $P \Longmapsto P'$ implies $\exists Q'.\ Q \Longmapsto Q' \wedge (P',Q') \in \mathscr{R}$
- $P \Longmapsto P'$ also implies $\exists Q'.\ Q \Longmapsto Q' \wedge (Q',P') \in \mathscr{R}$
- $P\Downarrow_\beta$ implies $Q\Downarrow_\beta$ for all barbs β

Two terms are coupled similar if they are related by a coupled simulation in both directions.

For all languages considered here, a process P is distributable into $P_1,\ldots,P_n$ if and only if we have $P \equiv (\nu\tilde{y})\,(P_1 \mid \ldots \mid P_n)$ (compare to the notion of a standard form of the π-calculus in [18] and the discussion on distributability in [31]). Moreover, two steps $a : P \longmapsto P_a$ and $b : P \longmapsto P_b$ of P are in conflict if one disables the other, i.e., if a and b compete for some action prefix. More precisely, two steps in the π-calculus are in conflict if they reduce the same choice, two steps in CMV^+ are in conflict if they reduce the same choice or the same conditional, and two steps in CMV are are in conflict if the reduce the same output, input, selection prefix, branching prefix, or conditional. Note that reducing the same choice not necessarily means to reduce the same summand in this choice. The steps a and b are distributable, if P is distributable into at least two parts such that one part performs the step a and the other part performs b.

2.2 Encodability Criteria

An encoding $[\![\cdot]\!]$ is a function from the processes of the source language into the processes of the target language, where we need encodability criteria to rule out trivial or meaningless encodings. In order to provide a general framework, Gorla in [10] suggests five criteria well suited for language comparison. Other frameworks were introduced e.g. in [8, 40]. We replace success sensitiveness of [10] by the stricter barb-sensitiveness, because it is more intuitive. As we claim, all separation results in this paper remain valid w.r.t. success sensitiveness instead of barb-sensitiveness.

The papers [6] and [21] require as additional criterion that the parallel operator is translated homomorphically. To strengthen the separation results, we use the slightly weaker criterion ‘preservation of distributability’ (see [28, 26]). The encoding of [6] that we discuss in § 5 translates the parallel operator homomorphically.

Definition 2.3 (Good Encoding)**.** We consider an encoding $[\![\cdot]\!]$ to be *good* if it is

compositional: The translation of an operator is captured by a context that takes as arguments the translations of the subterms of the operator.

name invariant: For every S and every substitution σ, it holds that $[\![S\sigma]\!] \asymp [\![S]\!]\,\sigma$.

operationally complete: For all $S \Longmapsto_{\mathrm{S}} S'$, it holds $[\![S]\!] \Longmapsto_{\mathrm{T}} \asymp [\![S']\!]$.

operationally sound: For all $[\![S]\!] \Longmapsto_{\mathrm{T}} T$, there is an S' s.t. $S \Longmapsto_{\mathrm{S}} S'$ and $T \Longmapsto_{\mathrm{T}} \asymp [\![S']\!]$.

divergence reflecting: For every S, $[\![S]\!] \longmapsto^{\omega}_{\mathrm{T}}$ implies $S \longmapsto^{\omega}_{\mathrm{S}}$.

barb-sensitive: For every S and every barb y, $S{\Downarrow_y}$ iff $[\![S]\!]\Downarrow_y$.

distributability preserving: For every $S \in \mathscr{P}_{\mathrm{S}}$ and for all terms $S_1,\ldots,S_n \in \mathscr{P}_{\mathrm{S}}$ that are distributable within S there are some $T_1,\ldots,T_n \in \mathscr{P}_{\mathrm{T}}$ that are distributable within $[\![S]\!]$ such that $T_i \asymp [\![S_i]\!]$ for all $1 \le i \le n$.

Moreover the equivalence $\asymp$ is a barb respecting (weak) reduction bisimulation.

Operational correspondence is the combination of operational *completeness* and *soundness*. Since we are focusing on separation results on untyped languages, we do not require an explicit criterion for types. In [6] the encoding from CMV^+ into CMV is proven to be type sound.

3 Separating Mixed Sessions and the Pi-Calculus via Leader Election

The first expressiveness result on the π-calculus that focuses on mixed choice is the separation result by Palamidessi in [21, 22]. This result uses the problem of leader election in symmetric networks as distinguishing feature.

Following [21] we assume that the set of names $\mathscr{N}$ contains names that identify the processes of the network and that are never used as bound names within electoral systems. For simplicity, we use natural numbers for this kind of names. A leader is announced by unguarding an output on its id. Then a network $P = (\nu\tilde{x})(P_1 \mid \ldots \mid P_k)$ in $\mathscr{P}_\pi$ or $P = (\nu\tilde{x}\tilde{y})(P_1 \mid \ldots \mid P_k)$ in $\mathscr{P}_{\mathsf{CMV}^+}$ is an *electoral system* if in every maximal execution exactly one leader is announced. We adapt the definition of electoral systems of [21] to obtain electoral systems in the π-calculus and in CMV^+.

Definition 3.1 (Electoral System)**.** A network $P = (\nu\tilde{x})(P_1 \mid \ldots \mid P_k)$ in $\mathscr{P}_\pi$ or $P = (\nu\tilde{x}\tilde{y})(P_1 \mid \ldots \mid P_k)$ in $\mathscr{P}_{\mathsf{CMV}^+}$ is an *electoral system* if for every execution $E : P \Longmapsto P'$ there exists an extension $E' : P \Longmapsto P' \Longmapsto P''$ and some $n \in \{1,\ldots,k\}$ (the leader) such that $P'''{\downarrow_n}$ for all P''' with $P'' \Longmapsto P'''$, but $P''{\not\Downarrow_m}$ for any $m \in \{1,\ldots,k\}$ with $m \neq n$.

Accordingly, an electoral system in the π-calculus announces a leader by unguarding some output on n that cannot be reduced or removed, where n is the id of the leader. In CMV^+ a leader is announced by unguarding a choice on the channel n. Since n is free this choice cannot be removed. A network is an electoral system if in every maximal execution exactly one leader n is announced.

We adapt the definition of hypergraphs that are associated to a network of processes in the π-calculus defined in [21] to networks in CMV^+. The hypergraph connects the nodes $1,\ldots,k$ of the network by edges representing the free channels that they share, where we ignore the outer restrictions of the network.

Definition 3.2 (Hypergraph)**.** Given a network $P = (\nu\tilde{x})(P_1 \mid \ldots \mid P_k)$ in $\mathscr{P}_\pi$ or $P = (\nu\tilde{x}\tilde{y})(P_1 \mid \ldots \mid P_k)$ in $\mathscr{P}_{\mathsf{CMV}^+}$, the *hypergraph* associated to P is $\mathsf{H}(P) = \langle N, X, t\rangle$ with $N = \{1,\ldots,k\}$, $X = \mathsf{fn}(P_1 \mid \ldots \mid P_n) \setminus N$, and $t(x) = \{n \mid x \in \mathsf{fn}(P_n)\}$ for each $x \in X$.

Because we ignore the outer restrictions of the network in the above definition, the hypergraphs of two structural congruent networks may be different. However, this is not crucial for our results.

Given a hypergraph $H = \langle N, X, t \rangle$, an automorphism on H is a pair $\sigma = \langle \sigma_N, \sigma_X \rangle$ such that $\sigma_N : N \to N$ and $\sigma_X : X \to X$ are permutations which preserve the type of arcs. For simplicity, we usually do not distinguish between σ_N and σ_X and simply write σ. Moreover, since σ is a substitution, we allow to apply σ on terms P, denoted as $P\sigma$. The orbit $O_\sigma(n)$ of $n \in N$ generated by σ is defined as the set of nodes in which the various iterations of σ map n, i.e., $O_\sigma(n) = \{n, \sigma(n), \ldots, \sigma^{h-1}(n)\}$, where σ^i represents the composition of σ with itself i times and $\sigma^h = \mathsf{id}$. We also adapt the notion of a symmetric system of [21] to obtain symmetric systems in the π-calculus as well as in CMV^+.

Definition 3.3 (Symmetric System)**.** Consider a network $P = (\nu \tilde{x})(P_1 \mid \ldots \mid P_k)$ in $\mathscr{P}_\pi$ or a network $P = (\nu \tilde{x}\tilde{y})(P_1 \mid \ldots \mid P_k)$ in $\mathscr{P}_{\mathsf{CMV}^+}$, and let σ be an isomorphism on its associated hypergraph $\mathsf{H}(P) = \langle N, X, t \rangle$. P is *symmetric w.r.t.* σ iff $P_{\sigma(i)} \approx_\pi P_i\sigma$ or $P_{\sigma(i)} \approx_{\mathsf{CMV}^+} P_i\sigma$ for each node $i \in N$. P is *symmetric* if it is symmetric w.r.t. all the automorphisms of $\mathsf{H}(P)$.

In contrast to [21] we use bisimilarity—$\approx_\pi$ and $\approx_{\mathsf{CMV}^+}$—instead of alpha conversion in the definition of symmetry. With this weaker notion of symmetry, we compensate for the weaker criterion on distributability that we use instead of the homomorphic translation of the parallel operator. Accordingly, we also consider networks as symmetric if they behave in a symmetric way; they do not necessarily need to be structurally symmetric.

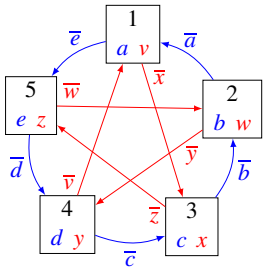
In the π-calculus we find symmetric electoral systems for many kinds of hypergraphs. We use such a solution of leader election in a network with five nodes as counterexample to separate CMV^+ from the π-calculus.

Example 3.4 (Leader Election in the π-Calculus)**.** Consider the network

$$\mathsf{S}_\pi^{\mathsf{LE}} = (\nu a,b,c,d,e,v,w,x,y,z)\,(S_1 \mid S_2 \mid S_3 \mid S_4 \mid S_5)$$

where $S_1 = \overline{e} + a.\big(\overline{x} + v.\overline{1}\big)$, $S_2 = \overline{a} + b.\big(\overline{y} + w.\overline{2}\big)$, $S_3 = \overline{b} + c.\big(\overline{z} + x.\overline{3}\big)$, $S_4 = \overline{c} + d.\big(\overline{v} + y.\overline{4}\big)$, and $S_5 = \overline{d} + e.\big(\overline{w} + z.\overline{5}\big)$. □

$\mathsf{S}_\pi^{\mathsf{LE}}$ is symmetric. Consider e.g. the permutation σ that permutes the channels as follows: $a \to b \to c \to d \to e \to a$, $v \to w \to x \to y \to z \to v$, and $1 \to 2 \to 3 \to 4 \to 5 \to 1$. Then $S_{\sigma(i)} = S_i\sigma$ for all $i \in \{1, \ldots, 5\}$. The network elects a leader in two stages. The first stage (depicted as blue circle) uses mixed choices on the channels a, b, c, d, e; in the second stage (depicted as a red star) we have mixed choices on the channels v, w, x, y, z. The picture on the right gives $\mathsf{H}\big(\mathsf{S}_\pi^{\mathsf{LE}}\big)$ extended by arrow heads to visualise the direction of interactions and the respective action prefixes. The senders in the two stages are losing the leader election game, i.e., are not becoming the leader. In the first stage two processes can be receivers and continue with the second stage. The process that is neither sender nor receiver in the first stage is stuck and also loses. The receiver of the second stage then becomes the leader by unguarding an output on its id. For instance we obtain the execution



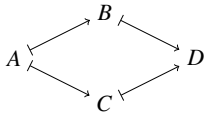
$$\mathsf{S}_\pi^{\mathsf{LE}} \longmapsto (\nu \tilde{n})\big(\overline{x} + v.\overline{1} \mid S_3 \mid S_4 \mid S_5\big) \longmapsto (\nu \tilde{n})\big(\overline{x} + v.\overline{1} \mid \overline{z} + x.\overline{3} \mid S_5\big) \longmapsto \overline{3} \mid (\nu \tilde{n})S_5 \not\longmapsto$$

with $\tilde{n} = a,b,c,d,e,v,w,x,y,z$ by reducing on the channels a and c in the first stage. The network $\mathsf{S}_\pi^{\mathsf{LE}}$ has 10 maximal executions (modulo structural congruence) that are obtained from the above execution by symmetry on the first two steps. In each maximal execution exactly one leader is elected.

We show that there exists no symmetric electoral system for networks of size five in CMV^+; or more generally no symmetric electoral system for networks of odd size in CMV^+. A key ingredient to separate the π-calculus with mixed choice from the asynchronous π-calculus in [21] is a confluence lemma. It states that in the asynchronous π-calculus a step reducing an output and an alternative step reducing an input cannot be conflict to each other and thus can be executed in any order. In the full π-calculus this confluence lemma is not valid, because inputs and outputs can be combined within a single choice construct and can thus be in conflict. For CMV^+ we observe that steps that reduce different endpoints can also not be in conflict to each other, because different channel endpoints cannot be combined in a single choice.

Lemma 3.5 (Confluence)**.** *Let $P,Q \in \mathscr{P}_{\mathsf{CMV}^+}$. Assume that $A = (\nu\tilde{x}\tilde{y})(P \mid Q)$ can make two steps $A \longmapsto (\nu\widetilde{x_1}\widetilde{y_1})(P_1 \mid Q_1) = B$ and $A \longmapsto (\nu\widetilde{x_2}\widetilde{y_2})(P_2 \mid Q_2) = C$ such that P_1 is obtained modulo $\equiv$ from P by reducing a choice on channel endpoint a and P_2 is obtained modulo $\equiv$ from P by reducing a choice on channel endpoint b with $a \neq b$. Then there exist $P_3, Q_3 \in \mathscr{P}_{\mathsf{CMV}^+}$ and $D = (\nu\widetilde{x_3}\widetilde{y_3})(P_3 \mid Q_3)$ such that $B \longmapsto D$ and $C \longmapsto D$, where $\widetilde{x_3} = \widetilde{x_1} \cup \widetilde{x_2}$ and $\widetilde{y_3} = \widetilde{y_1} \cup \widetilde{y_2}$.*

The proof of this confluence lemma relies on the observation that the two steps of A to B and C have to reduce distributable parts of A. Then these two steps are distributable, which in turn allows us to perform them in any order. Thus the expressive power of choice in CMV^+ is limited by the fact that syntactically the choice construct is fixed on a single channel endpoint. With this alternative confluence lemma, we can show that there is no electoral system of odd degree in CMV^+.



Lemma 3.6 (No Electoral System)**.** *Consider a network $P = (\nu\tilde{x}\tilde{y})(P_1 \mid \ldots \mid P_k)$ in CMV^+ with $k > 1$ being an odd number. Assume that the associated hypergraph $\mathsf{H}(P)$ admits an automorphism $\sigma \neq \mathsf{id}$ with only one orbit, and that P is symmetric w.r.t. σ. Then P cannot be an electoral system.*

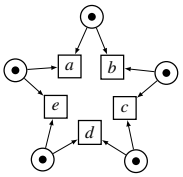
In the proof we construct a potentially infinite sequence of steps such that the system constantly restores symmetry, i.e., whenever a step destroys symmetry we can perform a sequence of steps that restores the symmetry. Therefore we rely on the assumption of σ generating only one orbit. This implies that $O_\sigma(i) = \{i, \sigma(i), \ldots, \sigma^{k-1}(i)\} = \{1, \ldots, k\}$, for each $i \in \{1, \ldots, k\}$. Because of that, whenever part i performs a step that destroys symmetry or parts i and j together perform a step that destroys symmetry, the respective other parts of the originally symmetric network can perform symmetric steps to restore the symmetry of the network. Because of the symmetry, the constructed sequence of steps does not elect a unique leader. Accordingly, the existence of this sequence ensures that P is not an electoral system. The odd degree of the network is necessary to ensure that we can apply the confluence lemma, which in turn ensures that we can always perform the sequence of steps to restore symmetry after the step that destroys the symmetry.

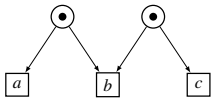
By the preservation of distributability, encodings preserve the structure of networks; and by name invariance, they also preserve the symmetry of networks. With operational correspondence and barb-sensitiveness, any good encoding of $\mathsf{S}_\pi^{\mathsf{LE}}$ is again a symmetric electoral system of size five. Since by Lemma 3.6 this is not possible, we can separate CMV^+ from the π-calculus.

Theorem 3.7 (Separate CMV^+ from the π-Calculus via Leader Election)**.**
There is no good encoding from the π-calculus into CMV^+.

4 Separating Mixed Sessions and the Pi-Calculus via Synchronisation

In [31] the technique used in [21] and its relation to synchronisation are analysed. Two synchronisation patterns, the pattern M and the pattern $\star$, are identified that describe two different levels of synchronisation and allow to more clearly separate languages along their ability to express synchronisation. These patterns are called M and $\star$, because their respective representations as a Petri net (see left and right picture) have these shapes. The pattern $\star$ captures the power of synchronisation of the π-calculus. In particular it captures what is necessary to solve the leader election problem.





The pattern M captures a very weak form of synchronisation, not enough to solve leader election but enough to make a fully distributed implementation of languages with this pattern difficult (see also [30]). This pattern was originally identified in [38] when studying the relevance of synchrony and distribution on Petri nets. As shown in [26, 31], the ability to express these different amounts of synchronisation in the π-calculus lies in its different forms of choices: to express the pattern $\star$ the π-calculus needs mixed choice, whereas separate choice allows to express the pattern M. Indeed we find the pattern M in CMV^+, but there are no $\star$ in CMV^+.

Example 4.1 (A M in CMV^+)**.** The process $\mathsf{P}_{\mathsf{M}}^{\mathsf{CMV}^+}$ is a M in CMV^+:

$$\mathsf{P}_{\mathsf{M}}^{\mathsf{CMV}^+} = (\nu xy)\Big(\underbrace{\begin{array}{l} x(1!\mathsf{true}.P_1 + 1?z.P_2) \\ y(1?z.P_5 + 1!\mathsf{true}.P_6) \end{array}}_{\text{location 1}} \mid \underbrace{\begin{array}{l} x(1!\mathsf{false}.P_3 + 1?z.P_4) \\ y(1?z.P_7 + 1!\mathsf{false}.P_8) \end{array}}_{\text{location 2}} \mid \Big)$$

A process is a M if it can perform three steps a,b,c, where a,b,c are names and not labels, such that a and b as well as b and c are in conflict whereas a and c are distributable steps. For instance we can pick the steps a, b, and c as:

Step a**:** $\mathsf{P}_{\mathsf{M}}^{\mathsf{CMV}^+} \longmapsto (\nu xy)\,(P_1 \mid x(1!\mathsf{false}.P_3 + 1?z.P_4) \mid P_5\{^{\mathsf{true}}/_z\} \mid y(1?z.P_7 + 1!\mathsf{false}.P_8))$

Step b**:** $\mathsf{P}_{\mathsf{M}}^{\mathsf{CMV}^+} \longmapsto (\nu xy)\,(P_1 \mid x(1!\mathsf{false}.P_3 + 1?z.P_4) \mid y(1?z.P_5 + 1!\mathsf{true}.P_6) \mid P_7\{^{\mathsf{true}}/_z\})$

Step c**:** $\mathsf{P}_{\mathsf{M}}^{\mathsf{CMV}^+} \longmapsto (\nu xy)\,(x(1!\mathsf{true}.P_1 + 1?z.P_2) \mid P_3 \mid y(1?z.P_5 + 1!\mathsf{true}.P_6) \mid P_7\{^{\mathsf{false}}/_z\})$

The process $\mathsf{P}_{\mathsf{M}}^{\mathsf{CMV}^+}$ is well-typed (see [34]). □

We use synchronisation patterns and the proof technique presented in [31] to present an alternative way to prove Theorem 3.7. By that we underpin our claim that the choice construct of CMV^+ is separate and not mixed, and we provide further intuition on why this choice construct is less expressive.

We inherit the definition of the synchronisation pattern $\star$ from [31], where we do not distinguish between local and non-local $\star$ since in the π-calculus there is no difference between parallel and distributable steps.

Definition 4.2 (Synchronisation Pattern $\star$)**.** Let $\langle \mathscr{P}, \longmapsto \rangle$ be a process calculus and $\mathsf{P}^\star \in \mathscr{P}$ such that:

- $\mathsf{P}^\star$ can perform at least five alternative reduction steps $i : \mathsf{P}^\star \longmapsto P_i$ for $i \in \{a,b,c,d,e\}$ such that the P_i are pairwise different;
- the steps a, b, c, d, and e form a circle such that a is in conflict with b, b is in conflict with c, c is in conflict with d, d is in conflict with e, and e is in conflict with a; and

- every pair of steps in $\{a,b,c,d,e\}$ that is not in conflict due to the previous condition is distributable in $\mathsf{P}^\star$.

In this case, we denote the process $\mathsf{P}^\star$ as $\star$.

In contrast to CMV^+ we do find $\star$ in the π-calculus.

Example 4.3 (The $\star$ in the π-Calculus)**.** Consider the following $\star$ in the π-calculus:

$$\mathsf{S}^\star_\pi = \overline{a} + b.\overline{o_b} \mid \overline{b} + c.\overline{o_c} \mid \overline{c} + d.\overline{o_d} \mid \overline{d} + e.\overline{o_e} \mid \overline{e} + a.\overline{o_a}$$

The steps $a,\ldots,e$ of Definition 4.2 are the steps on the respective channels.

Step a**:** $\mathsf{S}^\star_\pi \longmapsto S_a$ with $S_a = \overline{b} + c().\overline{o_c} \mid \overline{c} + d().\overline{o_d} \mid \overline{d} + e().\overline{o_e} \mid \overline{o_a}$,

Step b**:** $\mathsf{S}^\star_\pi \longmapsto S_b$ with $S_b = \overline{o_b} \mid \overline{c} + d().\overline{o_d} \mid \overline{d} + e().\overline{o_e} \mid \overline{e} + a().\overline{o_a}$,

Step c**:** $\mathsf{S}^\star_\pi \longmapsto S_c$ with $S_c = \overline{a} + b().\overline{o_b} \mid \overline{o_c} \mid \overline{d} + e().\overline{o_e} \mid \overline{e} + a().\overline{o_a}$,

Step d**:** $\mathsf{S}^\star_\pi \longmapsto S_d$ with $S_d = \overline{a} + b().\overline{o_b} \mid \overline{b} + c().\overline{o_c} \mid \overline{o_d} \mid \overline{e} + a().\overline{o_a}$

Step e**:** $\mathsf{S}^\star_\pi \longmapsto S_e$ with $S_e = \overline{a} + b().\overline{o_b} \mid \overline{b} + c().\overline{o_c} \mid \overline{c} + d().\overline{o_d} \mid \overline{o_e}$

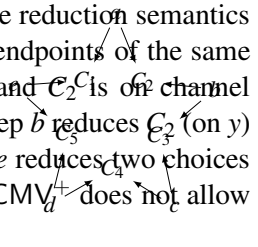
The different outputs $\overline{o_x}$ allow to distinguish between the different steps by their observables. □

We use the $\star$ $\mathsf{S}^\star_\pi$ as counterexample to show that there is no good encoding from the π-calculus into CMV^+. From Lemma 3.6 we learned that CMV^+ cannot express certain electoral systems. Accordingly, we are not surprised that CMV^+ cannot express the pattern $\star$.

Lemma 4.4. *There are no* $\star$ *in* CMV^+.

Proof. Assume the contrary, i.e., assume that there is a term $\mathsf{P}^\star_{\mathsf{CMV}^+}$ in CMV^+ that is a $\star$. Then $\mathsf{P}^\star_{\mathsf{CMV}^+}$ can perform at least five alternative reduction steps a,b,c,d,e such that neighbouring steps in the sequence a,b,c,d,e,a are pairwise in conflict and non-neighbouring steps are distributable. Since steps reducing a conditional cannot be in conflict with any other step, none of the steps in $\{a,b,c,d,e\}$ reduces a conditional. Then all steps in $\{a,b,c,d,e\}$ are communication steps that reduce an output and an input that both are part of choices (with at least one summand). Because of the conflict between a and b, these two steps reduce the same choice but this choice is not reduced in c, because a and c are distributable.

By repeating this argument, we conclude that in the steps a,b,c,d,e five choices $C_1,\ldots,C_5$ are reduced as depicted on the right, where e.g. the step a reduces the choices C_1 and C_2. By the reduction semantics of CMV^+, the two choices C_1 and C_2 that are reduced in step a need to use dual endpoints of the same channel. Without loss of generality, assume that C_1 is on channel endpoint x and C_2 is on channel endpoint y. Then the choice C_3 needs to be on channel endpoint x again, because step b reduces C_2 (on y) and C_3. By repeating this argument, then C_4 is on y and C_5 is on x. But then step e reduces two choices C_1 and C_5 that are both on channel endpoint x. Since the reduction semantics of CMV^+ does not allow such a step, this is a contradiction.



We conclude that there are no $\star$ in CMV^+. □

The proof of the above lemma tells us more about why choice in CMV^+ is limited. From the confluence property in CMV^+ we get the hint that the problem is the restriction of choice to a single channel endpoint. A $\star$ is a circle of steps of odd degree, where neighbouring steps are in conflict. More precisely, the star with five points in $\star$ is the smallest cycle of steps where neighbouring steps are in conflict and that contains non-neighbouring distributable steps. The proof shows that the limitation of choice to a single channel endpoint and the requirement of the semantics that a channel endpoint can interact with exactly

one other channel endpoint causes the problem. This also explains why Lemma 3.6 considers electoral systems of odd degree, because the odd degree does not allow to close the cycle as explained in the proof above. Indeed, if we change the syntax to allow mixed choice with summands on more than one channel, we obtain the mixed-choice-construct of the π-calculus. Similarly, we invalidate our separation result in the Theorems 3.7 and 4.5, if we change the semantics to allow two choices to communicate even if they are on the same channel. The latter may be more surprising, but indeed we do not need more than a single channel to solve leader election and build $\star$, e.g. $\mathsf{S}^{\star}_{\pi}$ remains a star if we choose $a = b = c = d = e$ (though we might want to pick different names $o_a, \ldots, o_e$ to be able to distinguish the steps).

We use $\mathsf{S}^{\star}_{\pi}$ in Example 4.3 as counterexample.

Theorem 4.5 (Separate CMV^{+} and the π-Calculus via $\star$)**.**
There is no good and distributability preserving encoding from the π-calculus into CMV^{+}.

To prove the above theorem, we show that the conflicts in the counterexample $\mathsf{S}^{\star}_{\pi}$ have to be translated into conflicts in its literal translation. Since the target language CMV^{+} cannot express a $\star$, to emulate $\mathsf{S}^{\star}_{\pi}$ it has to break the cycle and split at least one of the conflicts with the respective two neighbouring steps into two distributable conflicts: one for each neighbour. This causes a contradiction, because the distribution of the conflict induces new behaviour that is observable modulo the criteria we picked for good encodings.

5 Encoding Mixed Sessions into Separate Choice

In [6, § 7] an encoding of mixed sessions (CMV^{+}) into the variant of this session type system CMV with only separate choice (branching and selection) is presented. The proof of soundness of this encoding is missing in [6]. They suggest to prove soundness modulo "a weak form of bisimulation". As discussed below, the soundness criterion used in [6] needs to be corrected first.

The main idea of $[\![\cdot]\!]^{\mathsf{CMV}^{+}}_{\mathsf{CMV}}$ is to encode the information about whether a summand is an output or an input into the label used in branching, where a label l_i used with polarity ! in a choice typed as internal becomes $\mathrm{l}_{i,!}$ and in a choice typed as external it becomes $\mathrm{l}_{i,?}$. The dual treatment of polarities w.r.t. the type ensures that the labels of matching communication partners are translated to the same label.

Example 5.1 (Translation)**.** Consider for example the term $S \in \mathscr{P}_{\mathsf{CMV}^{+}}$:

$$S = (\nu xy)\,(y\,(\mathrm{l}!\mathsf{false}.S_1 + \mathrm{l}?z.S_2) \mid x\,(\mathrm{l}!\mathsf{true}.\mathbf{0} + \mathrm{l}?z.\mathbf{0}) \mid y\,(\mathrm{l}!\mathsf{false}.S_3 + \mathrm{l}?z.S_4))$$

S is well-typed but the type system forces us to assign dual types to x and y. Because of that, the choices on one channel need to be internal and on the other external. Let us assume that we have external choices on y and that the choice on x is internal. Moreover, we assume that both channels are marked as linear but typed as unrestricted. Then the translation[1] yields $[\![S]\!]^{\mathsf{CMV}^{+}}_{\mathsf{CMV}} \Longmapsto T_1$ with

$$\begin{aligned} T_1 = (\nu xy)\big(\, & y?c.c \triangleright \Big\{ \mathrm{l}_? : \Big(c!\mathsf{false}.[\![S_1]\!]^{\mathsf{CMV}^{+}}_{\mathsf{CMV}} \mid J_1\Big),\quad \mathrm{l}_! : \Big(c?z.[\![S_2]\!]^{\mathsf{CMV}^{+}}_{\mathsf{CMV}} \mid J_2\Big) \Big\} \\ & \mid (\nu st)\big(\, s \triangleright \{\mathrm{l}_1 : (\nu cd)\,(x!c.d \triangleleft \mathrm{l}_!.(d!\mathsf{true}.\mathbf{0} \mid J_3)),\quad \mathrm{l}_2 : (\nu cd)\,(x!c.d \triangleleft \mathrm{l}_?.(d?z.\mathbf{0} \mid J_4))\} \\ & \qquad \mid t \triangleleft \mathrm{l}_1.\mathbf{0} \mid t \triangleleft \mathrm{l}_2.\mathbf{0}\big) \\ & \mid y?c.c \triangleright \Big\{ \mathrm{l}_? : \Big(c!\mathsf{false}.[\![S_3]\!]^{\mathsf{CMV}^{+}}_{\mathsf{CMV}} \mid J_5\Big),\quad \mathrm{l}_! : \Big(c?z.[\![S_4]\!]^{\mathsf{CMV}^{+}}_{\mathsf{CMV}} \mid J_6\Big) \Big\}\big) \end{aligned}$$

[1] Note that [6] introduces a typed encoding, thus $[\![P]\!]^{\mathsf{CMV}^{+}}_{\mathsf{CMV}}$ actually means $[\![\Gamma \vdash P]\!]^{\mathsf{CMV}^{+}}_{\mathsf{CMV}}$, where $\Gamma \vdash P$ is the type statement ensuring that P is well-typed.

where we already performed a few steps to hide some technical details of the encoding function $[\![\cdot]\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}}$ that are not relevant for this explanation and where the $J_1,\ldots,J_6$ remain as junk from performing these steps. We call terms junk if they are stuck and do not emit barbs, i.e., we can ignore the junk. In particular, junk is invisible modulo $\approx_{\mathsf{CMV}}$. We observe, that in the translation of the first $y\,(\mathsf{l}!\mathsf{false}.S_1 + \mathsf{l}?z.S_2)$ in the first line of T_1 the output with label l is translated to the label $\mathsf{l}_?$ and the input with label l is translated to the label $\mathsf{l}_!$, whereas in the translation of its dual $x\,(\mathsf{l}!\mathsf{true}.\mathbf{0} + \mathsf{l}?z.\mathbf{0})$ in the second line of T_1 we obtain $\mathsf{l}_!$ for the output and $\mathsf{l}_?$ for the input. To emulate the step $S \longmapsto S_2' = (\nu xy)\,(S_2\{^{\mathsf{true}}/_z\} \mid y\,(\mathsf{l}!\mathsf{false}.S_3 + \mathsf{l}?z.S_4))$ of S in that true is transmitted to S_2, we start by picking the corresponding alternative, namely l_1 for sending, in the second and third line of T_1

$$\begin{aligned} T_1 \longmapsto T_2 = (\nu xy)\big(\, y?c.c \triangleright \Big\{ \mathsf{l}_? : \Big(c!\mathsf{false}.[\![S_1]\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}} \mid J_1\Big),\quad \mathsf{l}_! : \Big(c?z.[\![S_2]\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}} \mid J_2\Big)\Big\} \\ \mid (\nu cd)\,(x!c.d \triangleleft \mathsf{l}_!.\,(d!\mathsf{true}.\mathbf{0} \mid J_3)) \mid J_7 \\ \mid y?c.c \triangleright \Big\{ \mathsf{l}_? : \Big(c!\mathsf{false}.[\![S_3]\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}} \mid J_5\Big),\quad \mathsf{l}_! : \Big(c?z.[\![S_4]\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}} \mid J_6\Big)\Big\}\big) \end{aligned}$$

where J_7 again remains as junk. Then we perform a communication on xy, where we chose the input on y in the first line:

$$\begin{aligned} T_2 \longmapsto T_3 = (\nu xy)\big(\, (\nu cd)\big(\, c \triangleright \Big\{ \mathsf{l}_? : \Big(c!\mathsf{false}.[\![S_1]\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}} \mid J_1\Big),\quad \mathsf{l}_! : \Big(c?z.[\![S_2]\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}} \mid J_2\Big)\Big\} \\ \mid d \triangleleft \mathsf{l}_!.\,(d!\mathsf{true}.\mathbf{0} \mid J_3)\big) \mid J_7 \\ \mid y?c.c \triangleright \Big\{ \mathsf{l}_? : \Big(c!\mathsf{false}.[\![S_3]\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}} \mid J_5\Big),\quad \mathsf{l}_! : \Big(c?z.[\![S_4]\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}} \mid J_6\Big)\Big\}\big) \end{aligned}$$

Finally, two more steps on cd resolve the branching and transmit true:

$$\begin{aligned} T_3 \longmapsto\longmapsto T_4 = (\nu xy)\big(\, [\![S_2]\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}}\{^{\mathsf{true}}/_z\} \mid J_2 \mid J_3 \mid J_7 \mid J_8 \\ \mid y?c.c \triangleright \Big\{ \mathsf{l}_? : \Big(c!\mathsf{false}.[\![S_3]\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}} \mid J_5\Big),\quad \mathsf{l}_! : \Big(c?z.[\![S_4]\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}} \mid J_6\Big)\Big\}\big) \end{aligned}$$

This completes the emulation of $S \longmapsto S_2'$, i.e., the emulation of the single source term step $S \longmapsto S_2'$ required a sequence of target term steps $[\![S]\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}} \Longrightarrow T_1 \longmapsto T_2 \longmapsto T_3 \longmapsto\longmapsto T_4$. □

The operational soundness is defined in [6] as (adapting the notation):

$$\text{If } [\![S]\!] \longmapsto_{\mathrm{T}} T \text{ then } S \longmapsto_{\mathrm{S}} S' \text{ and } T \Longmapsto_{\mathrm{T}} \asymp [\![S']\!]. \tag{1}$$

As visualised above, the encoding translates a single source term step into a sequence of target term steps. Unfortunately, for such encodings the statement in (1) is not strong enough: with (1), we check only that the first step on a literal translation does not introduce new behaviour. The requirement $T \Longmapsto_{\mathrm{T}} \asymp [\![S']\!]$ additionally checks that the emulation started with $[\![S]\!] \longmapsto_{\mathrm{T}} T$ can be completed, but not that there are no alternative steps introducing new behaviour. Hence we prove a correct version of soundness as defined in [10] (see Definition 2.3).

Lemma 5.2 (Soundness, $[\![\cdot]\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}}$)**.** *The encoding* $[\![\cdot]\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}}$ *is operationally sound modulo* $\approx_{\mathsf{CMV}}$*, i.e.,* $[\![S]\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}} \Longmapsto T$ *implies* $S \Longmapsto S'$ *and* $T \Longmapsto \approx_{\mathsf{CMV}} [\![S']\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}}$.

As suggested we use $\approx_{\mathsf{CMV}}$, i.e., a form of weak reduction barbed bisimilarity that we simply call bisimilarity in the following. For soundness we have to show that all steps of encoded terms belong modulo bisimilarity to the emulation of a source term step. To prove Lemma 5.2, we analyse the sequence of steps $[\![S]\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}} \Longmapsto T$ and identify all source term steps $S \Longmapsto S'$ whose emulation is started within

$[\![S]\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}} \Longmapsto T$ and the target term steps $T \Longmapsto\approx_{\mathsf{CMV}} [\![S']\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}}$ that are necessary to complete all started emulations modulo bisimulation. Therefore, we use an induction on the number of steps in the sequence $[\![S]\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}} \Longmapsto T$ and analyse the encoding function in order to distinguish between different kinds of target term steps and the emulations of source term steps to that they belong. Note that, as it is typical for many encodability results, the proof of operational soundness is more elaborate than the proof of operational completeness presented in [6].

In Example 5.1 we have $T_4 \approx_{\mathsf{CMV}} [\![S_2']\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}}$, because all differences between T_4 and $[\![S_2']\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}}$ are due to junk that cannot be observed modulo $\approx_{\mathsf{CMV}}$. In fact, we have already $T_3 \approx_{\mathsf{CMV}} [\![S_2']\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}}$, since we consider a weak form of bisimulation here.

In the above variant of soundness T can catch up with the source term S' by the steps $T \Longmapsto\approx_{\mathsf{CMV}} [\![S']\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}}$. This allows for so-called *intermediate states*: target terms that are strictly in between the translation of two source terms, i.e., T such that $S \longmapsto S'$, $[\![S]\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}} \Longmapsto T \Longmapsto\approx_{\mathsf{CMV}} [\![S']\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}}$, but neither $[\![S]\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}} \approx_{\mathsf{CMV}} T$ nor $[\![S']\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}} \approx_{\mathsf{CMV}} T$ (see [25, 31]). In $[\![\cdot]\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}}$ such intermediate states are caused by mapping the task of finding matching communication partners of a single source term step onto several steps in the target. Consider the term T_2 in the above emulation of $S \longmapsto S_2'$. By picking the branch with label l_1, we discarded the branch with label l_2. Because of that, the emulation starting with $[\![S]\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}} \Longmapsto T_2$ can no longer emulate source term steps of S that use channel x for receiving, i.e., $T_2 \not\approx_{\mathsf{CMV}} [\![S]\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}}$. But, since we have not yet decided whether we emulate a communication with the first or second choice on y, we also have $T_2 \not\approx_{\mathsf{CMV}} [\![S_2']\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}}$ whenever $S_2 \not\approx_{\mathsf{CMV}^+} S_4$. Indeed, if we assume that S_1, S_2, S_3, S_4 are pairwise not bisimilar, then $T_2 \not\approx_{\mathsf{CMV}} [\![S']\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}}$ for all $S \longmapsto S'$, i.e., T_2 is an intermediate state.

The existence of intermediate states prevents us from using stronger versions of soundness, i.e., with $T \asymp [\![S']\!]$ instead of the requirement $T \Longmapsto_{\mathrm{T}} \asymp [\![S']\!]$ in soundness. The encoding $[\![\cdot]\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}}$ needs the steps in $T \Longmapsto\approx_{\mathsf{CMV}} [\![S']\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}}$ to complete the emulation of source term steps started in $[\![S]\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}} \Longmapsto T$. With the soundness result we can complete the proof of [6] that $[\![\cdot]\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}}$ presented in [6, § 7] is good.

Theorem 5.3 (Encoding from CMV^+ into CMV)**.** *The encoding* $[\![\cdot]\!]^{\mathsf{CMV}^+}_{\mathsf{CMV}}$ *from* CMV^+ *into* CMV *presented in [6] is good. By this encoding source terms in* CMV^+ *and their literal translations in* CMV *are related by coupled similarity.*

For the proof we take the completeness result from [6] and our soundness result in Lemma 5.2. The proof of the remaining properties is simple. That the combination of operational correspondence and barb sensitiveness induces a (weak reduction, barbed) coupled similarity that relates all source terms and their literal translations was proved in [33]. To obtain a tighter connection such as the bisimilarity, we would need the stronger version of soundness with $T \asymp [\![S']\!]$ instead of $T \Longmapsto_{\mathrm{T}} \asymp [\![S']\!]$ (see [33]).

As mentioned, a key feature of the encoding is to translate the nature of its summands, i.e., whether they are send or receive actions, into the label used by the target term. That this is possible, i.e., that the prefixes for send and receive in a choice of CMV^+ can be translated to labels in a separate choice of CMV such that the difference is not observable modulo the criteria in Definition 2.3, gives us the last piece of evidence that we need. CMV^+ does not allow to solve problems such as leader election (Theorem 3.7) that are standard problems for mixed choice; CMV^+ cannot express the synchronisation pattern $\star$ either that we associate with mixed choice (Theorem 4.5). Yet, CMV^+ can express the pattern M which is associated with *separate choice*, and is encoded by a language with only separate choice (Theorem 5.3). We conclude that choice in CMV^+ is semantically rather a separate choice.

Corollary 5.4.
The extension of CMV *given by* CMV^+ *introduces a form of separate choice rather than mixed choice.*

6 Related Work and Outlook

We conclude by discussing related work, summing up our results, and briefly discussing our next steps.

6.1 Related Work

Encodings or the proof of their absence are the main way to compare process calculi [3, 23, 10, 9, 26, 39, 24, 8, 40]. See [27] for an overview and discussion on encodings. We used this methodology to compare different variants of choice in session types.

The relevance of mixed choice for the expressive power of the π-calculus was extensively studied. An important encodability result on choices is the existence of a good encoding from the choice-free synchronous π-calculus into its asynchronous variant [4, 12], since it proves the relevance of choice. As for the separation result, [22, 10, 29] have shown that there is no good encoding from the full π-calculus, i.e., the synchronous π-calculus including mixed choice, into its asynchronous variant if an encoding should preserve the distribution of systems. Palamidessi in [21] was the first to point out that mixed choice strictly raises the expressive power of the π-calculus. Later work studies the criteria under that this separation result holds and alternative ways to prove this result: [20] studies the relevance of divergence reflection for this result and considers separate choice. [10, 23] discuss how to reprove this result if the rather strict criterion on the homomorphic translation of the parallel operator is replaced by compositionality. [26, 28] show that compositionality itself is not strong enough to replace the homomorphic translation of the parallel operator by presenting an encoding and then propose the preservation of distributability as criterion to regain the result of Palamidessi. [29] uses the more fundamental problem of breaking symmetries instead of leader election. [31] further simplifies this separation result by introducing synchronisation patterns to distinguish the languages. [32] shows that instead of the preservation of distributability or the homomorphic translation of the parallel operator also the preservation of causality can be used as criterion.

While there are a vast amount of theories [15], programming languages [1], and tools [36] of session types, as far as we know, the CMV^+-calculus is the only session π-calculus which extends external and internal choices to their mixtures with full constructs, i.e. delegation, shared (or unlimited) name passing, value passing, and recursion in its process syntax, proposes its typing system and proves type-safety. In the context of *multiparty session types* [14], there are several works that extend the original form of global types where choice is fixed (from one sender to one receiver) with more flexible forms of choices: Recent work in [17] e.g. allows the global type to specify a choice of one sender to transmit to one of several receivers. In [16] flexible choices are discussed but their well-formedness (which ensures deadlock-freedom of local types) needs to be checked by bisimuluation. These works focus on gaining expressiveness of behaviours of a set of local types (or a simple form of CCS-like processes which are equivalent to local types [17]) which correspond to *a single multiparty session*, without delegations, interleaved sessions, restrictions nor name passing.

More recently, [41] compares the expressive power of a variant of the π-calculus (with implicit matching) and the variant of CCS where the result of a synchronisation of two actions is itself an action subject to relabelling or restriction. Because of the connection between CCS-like languages and local types, it may be interesting to compare the expressiveness results in [41] with (variants of) multiparty

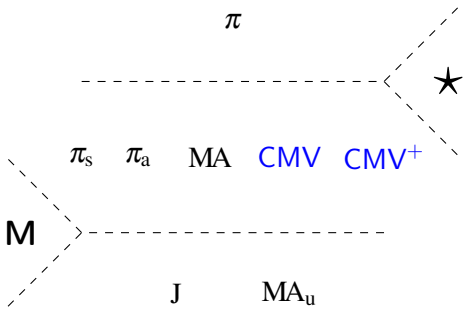


Figure 1: Hierarchy of Pi-like Calculi.

session types.

6.2 Summary and Outlook

We proved that CMV$^+$ is strictly less expressive than the π-calculus in two different ways: by showing that CMV$^+$ cannot solve leader election in symmetric networks of odd degree and that CMV$^+$ cannot express the synchronisation pattern $\star$. Then we provide the missing soundness proof for the encoding presented in [6]. From these results and the insights on the reasons of these results, we conclude that the choice primitive added to CMV in [6] is rather a separate choice and not a mixed choice at least with respect to its expressive power.

With these results we can extend the hierarchy of pi-like calculi obtained in [31, 30] by two more languages as depicted in Figure 1. This hierarchy orders languages according to their ability to express certain synchronisation patterns. At the top we have the π-calculus (π), because it can express the synchronisation pattern $\star$. In the middle are languages that can express M but not $\star$: the π-calculus with separate choice (π_s) [20], the asynchronous π-calculus without choice (π_a) [12, 4], Mobile Ambients (MA) [5], CMV, and CMV$^+$. In the bottom we have the join-calculus (J) [7] and Mobile Ambients with unique Ambient names (MA$_u$) [30], i.e., the languages that cannot express $\star$ or M. That $\pi, \pi_s, \pi_a, \text{MA}, \text{J}$, and MA$_u$ can or cannot express the respective pattern was shown in [31, 30].

Linearity as enforced by the type system of CMV/CMV$^+$ restricts the possible structures of communication protocols. In particular, the type system ensures that it is impossible to unguard two competing inputs or outputs on the same linear channel at the same time. Accordingly, it is not surprising that adding choice, even mixed choice, towards communication primitives under a type discipline that enforces linearity does not significantly increase the expressive power of the respective language (though it still might increase flexibility). However, that adding mixed choice between unrestricted communication primitives does not significantly increase the expressive power of the language, did surprise us. Unrestricted channels allow to have several in- or outputs on these channels in parallel, because the type system only ensures the absence of certain communication mismatches as e.g. that the sort of a transmitted value is as expected by the receiver; but not linearity (compare also to shared channels as e.g. in [13]). So, there is no obvious reason why the type system should limit the expressive power of unrestricted channels within a mixed choice. Indeed, it turns out that the problem lies not in the type system. In both ways to prove the separation result in § 3 and § 4 we completely ignore the type system and carry out the proof on the untyped version of the language, i.e., it is already the untyped version of CMV$^+$ that cannot express mixed choice despite a mixed-choice-like primitive. This limitation of the language definition, i.e., in its syntax and semantics, is not obvious and indeed it was very hard to spot

the problem.

We expect that adding mixed choice to the non-linear parts of other session type systems will instead significantly increase the expressive power. Accordingly, as the next step, we want to add a primitive for mixed choice between shared channels in session types such as described e.g. in [13, 42] and analyse the expressiveness of the resulting language.

Acknowledgements. The work is partially supported by EPSRC (EP/T006544/1, EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/1, EP/N028201/1, EP/T006544/1, EP/T014709/1, EP/V000462/1 and EP/X015955/1) and NCSS/EPSRC VeTSS.

References

[1] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral Types in Programming Languages. *Foundations and Trends in Programming Languages*, 3(2-3):95–230, 2016. doi:10.1561/2500000031.

[2] Benjamin Bisping, Uwe Nestmann, and Kirstin Peters. Coupled Similarity: the first 32 years. *Acta Informatica*, 57:439–463, 2019. doi:10.1007/s00236-019-00356-4.

[3] Frank S. Boer and Catuscia Palamidessi. Embedding as a tool for Language Comparison: On the CSP hierarchy. In *Proc. of CONCUR*, volume 527 of *LNCS*, pages 127–141. Springer, 1991. doi:10.1007/3-540-54430-5_85.

[4] Gérard Boudol. Asynchrony and the π-calculus (Note). Rapport de Recherche 1702, 1992. URL: https://hal.inria.fr/inria-00076939/document.

[5] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000. doi:10.1016/S0304-3975(99)00231-5.

[6] Filipe Casal, Andreia Mordido, and Vasco T. Vasconcelos. Mixed sessions. *Theoretical Computer Science*, 897:23–48, 2022. doi:10.1016/j.tcs.2021.08.005.

[7] Cédric Fournet and Georges Gonthier. The Reflexive Chemical Abstract Machine and the Join-Calculus. In Jr.G̃uy Steele, editor, *Proc. of POPL*, pages 372–385. ACM, 1996. doi:10.1145/237721.237805.

[8] Yuxi Fu. Theory of Interaction. *Theoretical Computer Science*, 611:1–49, 2016. doi:10.1016/j.tcs.2015.07.043.

[9] Yuxi Fu and Hao Lu. On the expressiveness of interaction. *Theoretical Computer Science*, 411(11-13):1387–1451, 2010. doi:10.1016/j.tcs.2009.11.011.

[10] Daniele Gorla. Towards a Unified Approach to Encodability and Separation Results for Process Calculi. *Information and Computation*, 208(9):1031–1053, 2010. doi:10.1016/j.ic.2010.05.002.

[11] Kohei Honda. Types for Dyadic Interaction. In Eike Best, editor, *Proc. of CONCUR*, volume 715 of *LNCS*, pages 509–523. Springer, 1993. doi:10.1007/3-540-57208-2_35.

[12] Kohei Honda and Mario Tokoro. An Object Calculus for Asynchronous Communication. In Mario Tokoro, Oscar Nierstrasz, and Peter Wegner, editors, *Proc. of ECOOP*, volume 612 of *LNCS*, pages 133–147. Springer, 1992. doi:10.1007/BFb0057019.

[13] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *Proc. of ESOP*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998. doi:10.1007/BFb0053567.

[14] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. *JACM*, 63:1–67, 2016. doi:10.1145/1328438.1328472.

[15] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of Session Types and Behavioural Contracts. *ACM Computing Surveys*, 49(1):3:1–3:36, 2016. doi:10.1145/2873052.

[16] Sung-Shik Jongmans and Nobuko Yoshida. Exploring Type-Level Bisimilarity towards More Expressive Multiparty Session Types. In *Proc. of ESOP*, volume 12075 of *LNCS*, pages 251–279. Springer, 2020. doi:10.1007/978-3-030-44914-8_10.

[17] Rupak Majumdar, Madhavan Mukund, Felix Stutz, and Damien Zufferey. Generalising Projection in Asynchronous Multiparty Session Types. In Serge Haddad and Daniele Varacca, editors, *Proc. of CONCUR*, volume 203 of *LIPIcs*, pages 35:1–35:24, 2021. doi:10.4230/LIPIcs.CONCUR.2021.35.

[18] Robin Milner. *Communicating and Mobile Systems: The π-Calculus*. Cambridge University Press, 1999.

[19] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, Part I and II. *Information and Computation*, 100(1):1–77, 1992. doi:10.1016/0890-5401(92)90008-4.

[20] Uwe Nestmann. What is a "Good" Encoding of Guarded Choice? *Information and Computation*, 156(1-2):287–319, 2000. doi:10.1006/inco.1999.2822.

[21] Catuscia Palamidessi. Comparing the Expressive Power of the Synchronous and the Asynchronous π-calculus. In *Proc. of POPL*, pages 256–265, 1997. doi:10.1145/263699.263731.

[22] Catuscia Palamidessi. Comparing the Expressive Power of the Synchronous and the Asynchronous π-calculus. *Mathematical Structures in Computer Science*, 13(5):685–719, 2003. doi:10.1017/S0960129503004043.

[23] Joachim Parrow. Expressiveness of Process Algebras. *Electronic Notes in Theoretical Computer Science*, 209:173–186, 2008. doi:10.1016/j.entcs.2008.04.011.

[24] Joachim Parrow. General conditions for full abstraction. *Mathematical Structures in Computer Science*, 26(4):655–657, 2014. doi:10.1017/S0960129514000280.

[25] Joachim Parrow and Peter Sjödin. Multiway synchronization verified with coupled simulation. In W.R. Cleaveland, editor, *Proc. of CONCUR*, pages 518–533. Springer Berlin Heidelberg, 1992. doi:10.1007/BFb0084813.

[26] Kirstin Peters. *Translational Expressiveness*. PhD thesis, TU Berlin, 2012. URL: http://opus.kobv.de/tuberlin/volltexte/2012/3749/.

[27] Kirstin Peters. Comparing Process Calculi Using Encodings. In *Proc. of EXPRESS/SOS*, EPTCS, pages 19–38, 2019. doi:10.48550/arXiv.1908.08633.

[28] Kirstin Peters and Uwe Nestmann. Is it a "Good" Encoding of Mixed Choice? In *Proc. of FoSSaCS*, volume 7213 of *LNCS*, pages 210–224, 2012. doi:10.1007/978-3-642-28729-9_14.

[29] Kirstin Peters and Uwe Nestmann. Breaking Symmetries. *Mathematical Structures in Computer Science*, 26(6):1054–1106, 2016. doi:10.1017/S0960129514000346.

[30] Kirstin Peters and Uwe Nestmann. Distributability of Mobile Ambients. *Information and Computation*, 275:104608, 2020. doi:10.1016/j.ic.2020.104608.

[31] Kirstin Peters, Uwe Nestmann, and Ursula Goltz. On Distributability in Process Calculi. In *Proc. of ESOP*, volume 7792 of *LNCS*, pages 310–329, 2013. doi:10.1007/978-3-642-37036-6_18.

[32] Kirstin Peters, Jens-Wolfhard Schicke-Uffmann, Ursula Goltz, and Uwe Nestmann. Synchrony versus Causality in Distributed Systems. *Mathematical Structures of Computer Science*, 26:1459–1498, 2016. doi:10.1017/S0960129514000644.

[33] Kirstin Peters and Rob van Glabbeek. Analysing and Comparing Encodability Criteria. In Silvia Crafa and Daniel Gebler, editors, *Proc. of EXPRESS/SOS*, volume 190 of *EPTCS*, pages 46–60, 2015. doi:10.4204/EPTCS.190.4.

[34] Kirstin Peters and Nobuko Yoshida. On the Expressiveness of Mixed Choice Sessions (Technical Report). Technical report, 2022. doi:10.48550/arXiv.2208.07041.

[35] Gordon D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60:17–140, 2004. [An earlier version of this paper was published as technical report at Aarhus University in 1981.]. doi:10.1016/j.jlap.2004.03.009.

[36] António Ravara Simon Gay, editor. *Behavioural Types: from Theory to Tools*. River Publisher, 2017. URL: https://www.riverpublishers.com/research_details.php?book_id=439.

[37] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An Interaction-based Language and its Typing System. In *Proc. of PARLE*, volume 817 of *LNCS*, pages 398–413, 1994. doi:10.1007/3-540-58184-7_118.

[38] Rob van Glabbeek, Ursula Goltz, and Jens-Wolfhard Schicke. On Synchronous and Asynchronous Interaction in Distributed Systems. In *Proc. of MFCS*, volume 5162 of *LNCS*, pages 16–35, 2008. doi:10.1007/978-3-540-85238-4.

[39] Rob J. van Glabbeek. Musings on Encodings and Expressiveness. In *Proc. of EXPRESS/SOS*, volume 89 of *EPTCS*, pages 81–98, 2012. doi:10.4204/EPTCS.89.7.

[40] Rob J. van Glabbeek. A Theory of Encodings and Expressiveness (Extended Abstract). In *Proc. of FoSSaCS*, volume 10803 of *LNCS*, pages 183–202, 2018. doi:10.1007/978-3-319-89366-2_10.

[41] Rob J. van Glabbeek. Comparing the expressiveness of the π-calculus and CCS. In Ilya Sergey, editor, *Proc. of ETAPS*, volume 13240 of *LNCS*, pages 548–574. Springer, 2022. URL: https://doi.org/10.1007/978-3-030-99336-8_20, doi:10.1007/978-3-030-99336-8_20.

[42] Nobuko Yoshida and Vascos T. Vasconcelos. Language Primitives and Type Discipline for Structured Communication-Based Programming Revisited: *Two Systems for Higher-Order Session Communication*. In *Proc. of SecReT*, volume 171, pages 73–93, 2006. doi:10.1016/j.entcs.2007.02.056.

Token Multiplicity in Reversing Petri Nets Under the Individual Token Interpretation

Anna Philippou Kyriaki Psara
Department of Computer Science, University of Cyprus
Nicosia, Cyprus
annap@ucy.ac.cy kpsara01@ucy.ac.cy

Reversing Petri nets (RPNs) have recently been proposed as a net-based approach to model causal and out-of-causal order reversibility. They are based on the notion of individual tokens that can be connected together via bonds. In this paper we extend RPNs by allowing multiple tokens of the same type to exist within a net based on the individual token interpretation of Petri nets. According to this interpretation, tokens of the same type are distinguished via their causal path. We develop a causal semantics of the model and we prove that the expressive power of RPNs with multiple tokens is equivalent to that of RPNs with single tokens by establishing an isomporphism between the Labelled Transition Systems (LTSs) capturing the reachable parts of the respective RPN models.

1 Introduction

Reversible computation is a form of computing where transitions can be executed in both the forward and the reverse direction, allowing systems to return to past states. It has been attracting increasing attention due to its application in a variety of fields such as low-power computing, biological modelling, quantum computation, robotics, and distributed systems.

In the sequential setting reversibility is generally understood as the ability to execute past actions in the exact inverse order in which they occurred, a process referred to as *backtracking*. However, in the concurrent setting matters are less clear. Indeed, various approaches have been investigated within a variety of formalisms [8, 26, 16, 33, 24, 20]. One of the most well-studied approaches considered suitable for a wide variety of concurrent systems is that of *causal-consistent reversibility* advocating that a transition can be undone only if all its effects, if any, have been undone beforehand [7]. The study of reversibility also extends to *out-of-causal-order reversibility*, a form of reversing where executed actions can be reversed in an out-of-causal order [28, 27, 15] most notably featured in biochemical systems.

In this work, we focus on Reversing Petri Nets [24] (RPNs), a reversible model inspired by Petri nets that allows the modelling of reversibility as realised by backtracking, causal-order, and out-of-causal-order reversing. A key challenge when reversing computations in Petri nets is handling *backward conflicts*. These conflicts arise when tokens occur in a certain place due to different causes making unclear which transitions ought to be reversed. To handle this ambiguity, RPNs introduce the notion of a *history* of transitions, which records causal information of executions. Furthermore, inspired by biochemical systems as well as other resource-aware applications, the model employs named tokens that can be connected together to form bonds, and are preserved during execution.

A restriction in RPNs is that each token is unique and in order to model a system with multiple items of the same type, it is necessary to employ a distinct token for each item, at the expense of the net size. In the current paper we consider an extension of RPNs, which allows multiple tokens of the same type. The introduction of multiple identical tokens creates further challenges involving backward conflicts and requires to extend the RPN machinery for extracting the causal dependencies between transitions.

V. Castiglioni and C. A. Mezzina (Eds): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics 2022 (EXPRESS/SOS 2022).
EPTCS 368, 2022, pp. 131–150, doi:10.4204/EPTCS.368.8

We note that formalizing causal dependencies is a well-studied problem in the context of Petri nets, where various approaches have been proposed to reason about causality [11, 13, 32]. In this work we draw inspiration from the so-called individual token and collective interpretations of Petri nets [12, 10]. The collective token philosophy considers all tokens of a certain type to be identical, which results in ambiguities when it comes to causal dependencies. In contrast, in an individual token interpretation, tokens are distinguished based on their causal path. This approach leads to more complicated semantics since to achieve token individuality requires precise correspondence between the token instances and their past. However, it enables backward determinism, which is a crucial property of reversible systems.

Contribution. In this paper we extend RPNs to support multiple tokens of the same type following the individual token interpretation. As such, tokens are associated with their causal history and, while tokens of the same type are equally eligible to fire a transition when going forward, when going backwards they are able to reverse only the transitions they have previously fired. In this context, we define a causal semantics for the model, based on the intuition that a causal link exists between two transitions if a token produced by one was used to fire the other. This leads to the observation that a transition may reverse in causal order only if it was the last transition executed by all the tokens it has involved. We note that this approach allows a causal-order reversible semantics that, unlike the original RPN model, does not require any global history information. In fact, all information necessary for reversal is available locally within the history of tokens. Subsequently, we turn to study the expressiveness of the presented model in comparison to RPNs with single tokens. To do this we employ Labelled Transition Systems (LTSs) capturing the state space of RPN models. We show that for any RPN with multiple tokens there exists an RPN with single tokens with an isomorphic LTS, thereby confirming our conjecture that RPNs with single tokens are as expressive as RPNs with multiple tokens.

Related Work. The first study of reversible computation within Petri nets was proposed in [4, 5], where the authors investigated the effects of adding *reversed* versions of selected transitions by reversing the directions of a transition's arcs. Unfortunately, this approach to reversibility violates causality. Towards examining causal consistent reversibility the work in [21] investigates whether it is possible to add a complete set of effect-reverses for a given transition without changing the set of reachable markings, showing that this problem is in general undecidable. In another line of work [20] propose a causal semantics for P/T nets by identifying the causalities and conflicts of a P/T net through unfolding it into an equivalent occurrence net and subsequently introducing appropriate reverse transitions to create a coloured Petri net (CPN) that captures a causal-consistent reversible semantics. On a similar note, [19] introduces the notion of reversible occurrence nets and associates a reversible occurrence net to a causal reversible prime event structure, and vice versa. Finally, [6] introduces a reversible approach to Petri nets following the individual token interpretation. This work is similar to our approach though it refers to a basic PN model, which does not contain named tokens nor bonds, and it does not support backtracking and out-of-causal reversibility.

The modelling of bonding in the context of reversibility was first considered within reversible processes and event structures in [29], where its usefulness was illustrated with examples taken from software engineering and biochemistry. Reversible frameworks that feature bonds as first-class entities, like RPNs, also include the Calculus of Covalent Bonding [15], which supports causal and out-of-causal-order reversibility in the context of chemical reactions, as well as the Bonding Calculus [1], a calculus developed for modeling covalent bonds between molecules in biochemical systems. In fact, the latter two frameworks and RPNs were reviewed and compared for modeling chemical reactions in [14] with

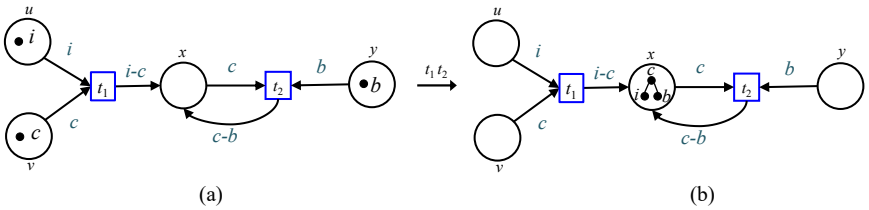


Figure 1: RPN example of a pen assembly/dissassembly

case study the autoprotolysis of water.

This paper extends a line of research on reversing Petri nets, initially introduced for acyclic nets [22] and subsequently for nets with cycles [24]. The usefulness of the framework was illustrated in a number of examples including the modelling of long-running transactions with compensation and a signal-passing mechanism used by the ERK pathway. The RPN framework has been extended to control reversibility in [25] with an application to Massive MIMO. Introducing multiple tokens in RPNs was also examined in [23] by allowing multiple tokens of the same type to exist within a net following the collective interpretation and yielding a locally-controlled, out-of-causal-order reversibility semantics. RPNs have been translated to Answer Set Programming (ASP), a declarative programming framework with competitive solvers [9], and to bounded Coloured Petri Nets [2, 3].

2 Reversing Petri Nets with Multiple tokens

In our previous works we introduced Reversing Petri Nets, a net-based formalism, which features individual tokens that can be connected together via bonds [24]. An assumption of RPNs is that tokens are pairwise distinct. To relax this restriction, subsequent work [23] introduced token multiplicity whereby a model may contain multiple tokens of the same type. It was observed that the possibility of firing a transition multiple times using different sets of tokens, may introduce nondeterminism, also known as backward conflict, when going backwards. Furthermore, two approaches were identified to define reversible semantics in the presence of such backward conflicts, inspired by the individual token and the collective token interpretations [10, 12], defined to reason about causality in Petri nets. In the individual token approach, multiple tokens of the same type residing in the same place are distinguished based on their causal path, whereas in the collective token interpretation they are not distinguished. In [23] the model of RPNs with multiple tokens was investigated under the collective token approach, yielding an out-of-causal-order form of reversibility. In this work, we instead apply the individual token interpretation to define a causal semantics, and we establish that in fact the addition of multiple tokens does not add to the expressiveness of the model, in that for any RPN with multiple tokens there exists an equivalent RPN with only a single token of each type.

To appreciate the challenges induced through the introduction of multiple tokens and the difference between the individual and the collective token interpretations, let us consider the example in Fig. 1(a). In this example we may see an RPN model of an assembly/disassembly of a pen. The product consists of the ink, the cup, and the button of the pen, modelled by tokens i, c, and b, respectively. We may observe that transitions, in addition to transferring tokens between places, have the capacity of creating bonds. Thus, the process of manufacturing the pen requires the ink to be fitted inside the cup, modelled by the creation of the bond $i-c$ by transition t_1 and, subsequently, the fitting of the button on the cup to complete

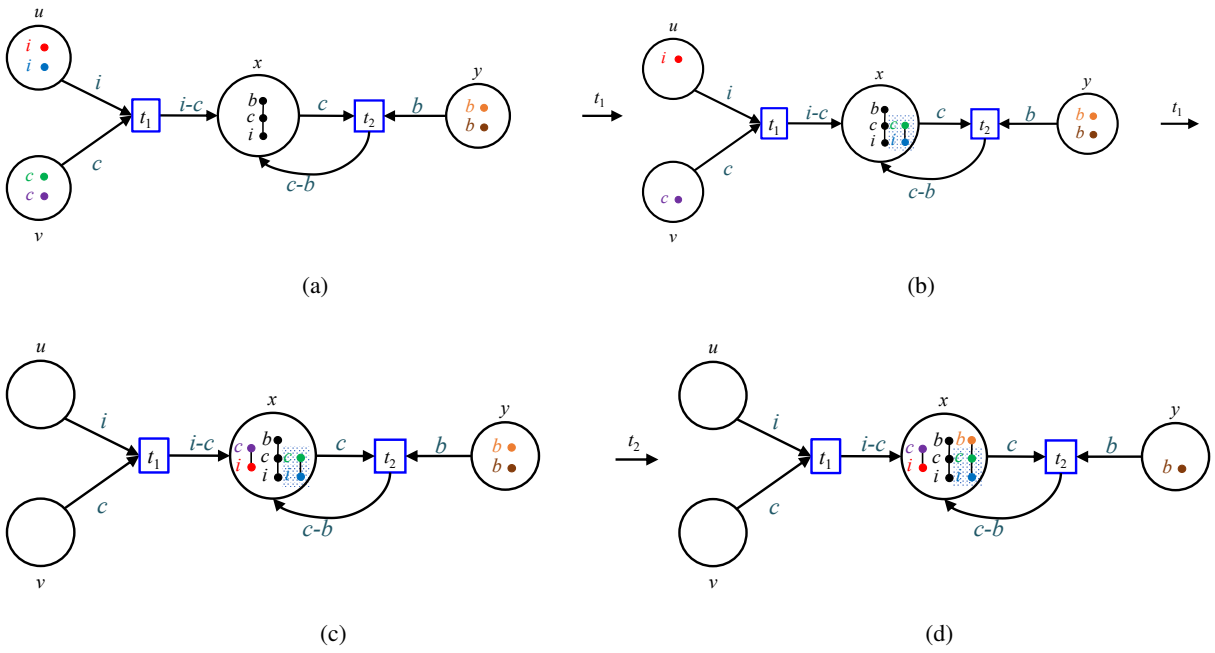


Figure 2: Executing transition t_1 in the net (a) may yield the net in (b). Different selections of tokens could have been made. In net (b) transition t_1 is executed with the (only) available tokens leading to net (c), whereby execution of t_2 with the component produced by the first execution of t_1 yields net (d).

the assembly, modelled as the creation of the bond $c-b$ by transition t_2 (RPN in Fig. 1(b)). The effect of reversing a transition in RPNs is to break the bonds created by the transition (if any) and returning the tokens/bonds from the outgoing places of the transition. In [22, 24] machinery has been developed in order to model backtracking, causal, and out-of-causal-order reversibility for the model. In particular, in the example of Fig. 1(b) reversing transition t_2 will result in the destruction of bond $c-b$ and the return of token b to place y.

Suppose we wish to extend the model of Fig. 1(a) for the assembly of two pens. Given that in RPNs tokens are unique, it would be necessary to introduce three new and distinct tokens and clone the transitions while renaming their arcs to accommodate for the names of the new tokens to be employed, resulting in a considerable expansion of the model for each new pen to be produced. Thus, a natural extension of the formalism involves relaxing this restriction and allowing multiple tokens of the same type to exist within a model. To this effect consider the scenario of Fig. 2(a) presenting a system with an already assembled/sample pen in place x and two items of each of the ink, cup, and button components.

An issue arising in this new setting is that due to the presence of multiple tokens of the same type, the phenomenon of backwards nondeterminism occurs when transitions are reversed. For instance, after execution of transition t_1 twice and t_2, two assembled pens will exist in place x and well as a component $i-c$, as seen in Fig. 2(d). Suppose that in this state transition t_1 is reversed. In the collective token interpretation, all instances of the bond $i-c$ are considered identical. As a result, any of these bonds could be destroyed during the reversal of transition t_1. However, in the individual token interpretation the various ink and cup tokens are distinguished based on their causal path. Therefore, the first execution of transition t_1 yielding the net in Fig. 2(b) and involving the shaded component of tokens in the figure, is considered to have caused the execution of transition t_2. Given this causal relationships between the

transitions, under a causal reversibility semantics, the specific $i-c$ component should not be decomposed until transition t_2 is reversed. Similarly, the pre-existing pen should not be broken down into its parts as it was not the created by any of the transitions. Instead, reversing transition t_1 in the RPN of Fig. 2(d) should break the bond in the component consisting the single bond $i-c$. Note that this is compatible with the understanding that disassembly of the product would not allow the separation of the ink from the inside of the cup before the button is removed, since this is enclosed within the pair of the cup and the button.

As a result we observe that following the individual token interpretation, reversing a computation requires keeping track of past behavior – in the context of the example, distinguishing the tokens involving the pre-existing pen and the tokens used to fire each transition. In the following sections we implement this approach for introducing multiple tokens and we study its properties in the context of causal-order reversibility. Furthermore, we establish a correspondence between this model and RPNs with single tokens.

3 Multi Reversing Petri Nets

We present multi reversing Petri nets, an extension of RPNs with multiple tokens of the same type that allow transitions to be reversed following the individual token interpretation. Formally, they are defined as follows:

Definition 1 A *multi reversing Petri net* (MRPN) is a tuple $(P,T,\mathscr{A},\mathscr{A}_V,\mathscr{B},F)$ where:

1. P is a finite set of *places* and T is a finite set of *transitions*.
2. $\mathscr{A}$ is a finite set of *base* or *token types* ranged over by $A,B,\ldots$
3. $\mathscr{A}_V$ is a finite set of *token variables* ranged over by $a,b,\ldots$ We write $\mathsf{type}(a)$ for the type of variable a and assume that $\mathsf{type}(a)\in\mathscr{A}$ for all $a\in\mathscr{A}_V$.
4. $\mathscr{B}\subseteq\mathscr{A}\times\mathscr{A}$ is a finite set of undirected *bond types* ranged over by $\beta,\gamma,\ldots$ We assume $\mathscr{B}$ to be a symmetric relation and we consider the elements (A,B) and (B,A) to refer to the same bond type, which we also denote by $A-B$. Furthermore, we write $\mathscr{B}_V\subseteq\mathscr{A}_V\times\mathscr{A}_V$, assuming that (a,b) and (b,a) represent the same bond, also denoted as $a-b$.
5. $F:(P\times T\cup T\times P)\to\mathscr{P}(\mathscr{A}_V\cup\mathscr{B}_V)$ defines a set of directed labelled *arcs* each associated with a subset of $\mathscr{A}_V\cup\mathscr{B}_V$, where $(a,b)\in F(x,y)$ implies that $a,b\in F(x,y)$. Moreover, for all $t\in T$, $x,y\in P$, $x\neq y$, $F(x,t)\cap F(y,t)=\emptyset$.

A multi reversing Petri net is built on the basis of a set of *token types*. Multiple occurrences of a token type, referred to as *token instances*, may exist in a net. Tokens of the same type have identical capabilities on firing transitions and can participate only in transitions with variables of the same type.

As standard in net-based frameworks, places and transitions are connected via labelled directed arcs. These labels are derived from $\mathscr{A}_V\cup\mathscr{B}_V$. They express the requirements and the effects of transitions based on the type of tokens consumed. Thus, collections of tokens corresponding to the same types and connections as the variables on the labelled arc are able to participate in the transition. More precisely, if $F(x,t)=X\cup Y$, where $X\subseteq\mathscr{A}_V$, $Y\subseteq\mathscr{B}_V$, the firing of t requires a distinct token instance of type $\mathsf{type}(a)$ for each $a\in X$, such that the overall selection of tokens are connected together satisfying the restrictions posed by Y. Similarly, if $F(t,x)=X\cup Y$, where $X\subseteq\mathscr{A}_V$, $Y\subseteq\mathscr{B}_V$, this implies that during the forward execution of the transition for each $a\in X$ a token instance of type $\mathsf{type}(a)$ will be transmitted to place x by the transition, in addition to the bonds specified by Y, some of which will be created as an effect of

the transition. We make the assumption that if $(a,b) \in Y$ then $a,b \in X$ and the same variable cannot be used on two incoming arcs of a transition.

We introduce the following notations. We write $\circ t = \{x \in P \mid F(x,t) \neq \emptyset\}$ and $t\circ = \{x \in P \mid F(t,x) \neq \emptyset\}$ for the incoming and outgoing places of transition t, respectively. Furthermore, we write $\mathsf{pre}(t) = \bigcup_{x \in P} F(x,t)$ for the union of all labels on the incoming arcs of transition t, and $\mathsf{post}(t) = \bigcup_{x \in P} F(t,x)$ for the union of all labels on the outgoing arcs of transition t.

We restrict our attention to well-formed MRPNs, which satisfy the conservation property [18] in the sense that the number of tokens in a net remains constant during execution. In fact, as we will prove in the sequel, in well-formed nets individual tokens are conserved.

Definition 2 An MRPN $(P,T,\mathscr{A},\mathscr{A}_V,\mathscr{B},F)$ is *well-formed* if for all $t \in T$:

1. $\mathscr{A}_V \cap \mathsf{pre}(t) = \mathscr{A}_V \cap \mathsf{post}(t)$ and
2. $F(t,x) \cap F(t,y) = \emptyset$ for all $x,y \in P$, $x \neq y$.

Thus, a well-formed MRPN satisfies (1) whenever a variable exists in the incoming arcs of a transition then it also exists on its outgoing arcs, and vice versa, which implies that transitions neither create nor erase tokens, and (2) tokens/bonds cannot be cloned into more than one outgoing place.

In the context of token multiplicity, a mechanism is needed in order to distinguish between token instances with respect to their causal path. For instance, consider the MRPN in Fig. 2(d). In this state, three connected components of tokens are positioned in place x, where tokens of the same type, e.g. the three c tokens have distinct connections and causal histories. To capture this, we distinguish between token instances, as follows:

Definition 3 Given an MRPN $(P,T,\mathscr{A},\mathscr{A}_V,\mathscr{B},F)$ a *token instance* has the form (A,i,xs) where xs is a (possibly empty) list of triples $[(k_1,t_1,v_1),\ldots,(k_n,t_n,v_n)]$ with $n \geq 0$, where $i \geq 1$, $A \in \mathscr{A}$, and for all i, $k_i \in \mathbb{N}$, $t_i \in T$, and $v_i \in \{*\} \cup \mathscr{A}_V$. We write $\mathscr{A}_I$ for the set of token instances ranged over by A_1, $A_2,\ldots$, and we define the set of bond instances $\mathscr{B}_I$ by $\mathscr{B}_I = \mathscr{A}_I \times \mathscr{A}_I$. Furthermore, given $A_i = (A,i,[(k_1,t_1,v_1),\ldots,(k_n,t_n,v_n)])$, we write

$$\begin{array}{rcl}
\mathsf{type}(A_i) & = A & \\
A_i\downarrow & = & (A,i) \\
\mathsf{cpath}A_i & = & [(t_1,v_1),\ldots,(t_n,v_n)] \\
\mathsf{last}(A_i) & = & (k_n,t_n,v_n) \\
A_i + (k,t,v) & = & (A,i,[(k_1,t_1,v_1),\ldots,(k_n,t_n,v_n),(k,t,v)]) \\
\mathsf{init}(A_i) & = & (A,i,[(k_1,t_1,v_1),\ldots,(k_{n-1},t_{n-1},v_{n-1})])
\end{array}$$

The set of token instances $\mathscr{A}_I$ corresponds to the basic entities that occur in a system. In the initial state of a net, tokens have the form $(A,i,[])$ where i is a unique identifier for the specific token instance of type A. As computation proceeds the tokens evolve to capture their causal path. If a transition t is executed in the forward direction, with some token instance $(A,i,[(k_1,t_1,v_1),\ldots,(k_n,t_n,v_n)])$ substituted for a variable v, then the token evolves to $(A,i,[(k_1,t_1,v_1),\ldots,(k_n,t_n,v_n),(k,t,v)])$, where k is an integer that characterizes the executed transition, as we will formally define in the sequel.

In a graphical representation, tokens instances are indicated by $\bullet$ associated with their description, places by circles, transitions by boxes, and bonds by lines between tokens. Note that token variables $a \in F(x,t) \cap \mathscr{A}_V$ with $\mathsf{type}(a) = A$ are denoted by $a:A$ over the corresponding arc $F(x,t)$. An example of an MRPN can be seen in Fig. 3. In this example, we have $\mathscr{A} = \{I,C,B\}$, $\mathscr{A}_V = \{i,c,b\}$, and the set of token instances in the specific state are $\{(I,i,[]),(B,i,[]),(C,i,[]) \mid i \in \{1,2,3\}\}$.

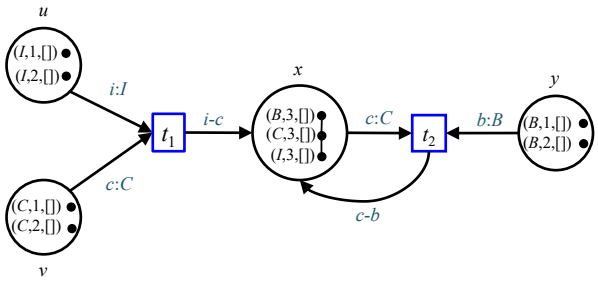


Figure 3: The net of Fig. 2(a) presented as an MRPN.

As with RPNs the association of token/bond instances to places is called a *marking* such that $M : P \to 2^{\mathscr{A}_I \cup \mathscr{B}_I}$, where we assume that if $(A_i, B_i) \in M(x)$ then $A_i, B_i \in M(x)$. In addition, we employ the notion of a *history*, which assigns a memory to each transition $H : T \to 2^{\mathbb{N}}$. Intuitively, a history of $H(t) = \emptyset$ for some $t \in T$ captures that the transition has not taken place, or every execution of it has been reversed, and a history such that $k \in H(t)$, captures that the transition had a firing with identifier k that was not reversed. Note that $|H(t)| > 1$ may arise due to cycles but also due to the consecutive execution of the transition by different token instances. A pair of a marking and a history, $\langle M, H \rangle$, describes a *state* of an MRPN with $\langle M_0, H_0 \rangle$ the initial state, where $H_0(t) = \emptyset$ for all $t \in T$ and if $A_i \in M_0(x)$, $x \in P$, then $A_i = (A, i, [])$, and $A_i \in M_0(y)$ implies that $x = y$.

Finally, we define $\mathsf{con}(A_i, W)$, where $A_i \in \mathscr{A}_I$ and $W \subseteq \mathscr{A}_I \cup \mathscr{B}_I$, to be the tokens connected to A_i as well as the bonds creating these connections according to set W:

$$\begin{array}{rcl} \mathsf{con}(A_i, W) & = & (\{A_i\} \cap W) \\ & \cup & \{x \mid \exists w \text{ s.t. } \mathsf{path}(A_i, w, W), (B_i, C_i) \in w, x \in \{(B_i, C_i), B_i, C_i\}\} \end{array}$$

where $\mathsf{path}(A_i, w, W)$ if $w = \langle \beta_1, \ldots, \beta_n \rangle$, and for all $1 \leq i \leq n$, $\beta_i = (x_{i-1}, x_i) \in W \cap \mathscr{B}_I$, $x_i \in W \cap \mathscr{A}_I$, and $x_0 = A_i$. For example, consider the net in Fig. 3 and let W represent the set of token and bond instances in place x. Then, $\mathsf{con}((I, 3, []), W) = \{(I_3, C_3, B_3, (I_3, C_3), (C_3, B_3)\}$, where $I_3 = (I, 3, [])$, $B_3 = (B, 3, [])$, and $C_3 = (C, 3, [])$.

3.1 Forward Execution

During the forward execution of a transition in an MRPN, a set of token and bond instances, as specified by the incoming arcs of the transition, are selected and moved to the outgoing places of the transition, possibly forming and/or destroying bonds. Precisely, for a transition t we define $\mathsf{eff}^+(t)$ to be the bonds that occur on its outgoing arcs but not the incoming ones and by $\mathsf{eff}^-(t)$ the bonds that occur in the incoming arcs but not the outgoing ones:

$$\mathsf{eff}^+(t) = \mathsf{post}(t) - \mathsf{pre}(t) \qquad \mathsf{eff}^-(t) = \mathsf{pre}(t) - \mathsf{post}(t)$$

Due to the presence of multiple instances of the same token type, it is possible that different token instances are selected during the transition's execution. To enable such a selection of tokens we define the following:

Definition 4 An injective function $\mathscr{W} : V \to \mathscr{A}_I$, where $V \subseteq \mathscr{A}_V$, is called a *type-respecting assignment* if for all $a \in V$, if $\mathscr{W}(a) = A_i$ then $\mathsf{type}(a) = \mathsf{type}(A_i)$.

We extend the above notation and write $\mathscr{W}(a,b)$ for $(\mathscr{W}(a),\mathscr{W}(b))$ and, given a set $L \subseteq \mathscr{A}_V \cup \mathscr{B}_V$, we write $\mathscr{W}(L) = \{\mathscr{W}(x) \mid x \in L\}$.

Based on the above we define the following:

Definition 5 Given an MRPN $(P,T,\mathscr{A},\mathscr{A}_V,\mathscr{B},F)$, a state $\langle M,H\rangle$, and a transition t, we say that t is *forward-enabled* in $\langle M,H\rangle$ if there exists a type-respecting assignment $\mathscr{S} : \mathsf{pre}(t) \cap \mathscr{A}_V \to \mathscr{A}_I$ such that:

1. $\mathscr{S}(F(x,t)) \subseteq M(x)$ for all $x \in {}^\circ t$.
2. If $a,b \in F(x,t)$ for some $x \in {}^\circ t$ and $(a,b) \in \mathsf{eff}^+(t)$, then $\mathscr{S}(a,b) \notin M(x)$.
3. If $a \in F(t,y_1)$ and $b \in F(t,y_2)$, for some $y_1,y_2 \in t^\circ$, $y_1 \neq y_2$, then $\mathsf{con}(\mathscr{S}(a),\mathsf{comp}_\mathsf{f}(t,\mathscr{S},M)) \neq \mathsf{con}(\mathscr{S}(b),\mathsf{comp}_\mathsf{f}(t,\mathscr{S},M))$.

where $\mathsf{comp}_\mathsf{f}(t,\mathscr{S},M) = (\bigcup_{x\in {}^\circ t} M(x) \cup \mathscr{S}(\mathsf{eff}^+(t))) - \mathscr{S}(\mathsf{eff}^-(t))$.

Thus, t is forward-enabled in state $\langle M,H\rangle$ if there exists a type-respecting assignment $\mathscr{S}$ of token instances to the variables on the incoming edges of t, which we will refer to as a *forward-enabling assignment* of t, such that (1) the token instances and bonds required by the transition's incoming edges, according to $\mathscr{S}$, are available from the appropriate input places, (2) if the selected token instances to be transferred by the transition are to be bonded together by the transition then they should not be already bonded in an incoming place of the transition (thus the bonds that occur only on the outgoing arcs of a transition are the bonds being created by the transition), and (3) if two token instances are transferred by a transition to different outgoing places then these tokens should not be connected. This is to ensure that connected components are not cloned. Note that $\mathsf{comp}_\mathsf{f}(t,\mathscr{S},M) = (\bigcup_{x\in {}^\circ t} M(x) \cup \mathscr{S}(\mathsf{eff}^+(t))) - S(\mathsf{eff}^-(t))$ denotes the set of token and bond instances that occur in the incoming places of t ($\bigcup_{x\in {}^\circ t} M(x)$), including the new bond instances created by t ($\mathscr{S}(\mathsf{eff}^+(t))$), and removing the bonds destroyed by it ($\mathscr{S}(\mathsf{eff}^-(t))$). Intuitively, $\mathsf{comp}_\mathsf{f}(t,\mathscr{S},M)$ contains the components that are moved forward by the transition.

To execute a transition t according to an enabling assignment $\mathscr{S}$, the selected token instances along with their connected components are relocated to the outgoing places of the transition as specified by the outgoing arcs, with bonds created and destroyed accordingly. An additional effect is the update of the affected token and bond instances to capture the executed transition in their causal path. To capture this update we define where k is an integer associated with the specific transition instance:

$$A_i \oplus (\mathscr{S},t,k) = \begin{cases} A_i + (k,t,a) & \text{if } \mathscr{S}(a) = A_i \\ A_i + (k,t,*) & \text{if } \mathscr{S}^{-1}(A_i) = \bot \end{cases}$$

Note that A_i may not belong to the range of $\mathscr{S}$, i.e. $\mathscr{S}^{-1}(A_i) = \bot$, if A_i was not specifically selected to instantiate a variable in $\mathsf{pre}(t)$ but, nonetheless, belonged to a connected component transferred by the transition. This is recorded in the causal path of the token instance via the triple $(k,t,*)$. Moreover, we write $(A_i,B_j) \oplus (\mathscr{S},t,k)$ for $(A_i \oplus (\mathscr{S},t,k), B_j \oplus (\mathscr{S},t,k))$ and, given $L \subseteq \mathscr{A}_I \cup \mathscr{B}_I$, we write $L \oplus (\mathscr{S},t,k) = \{x \oplus (\mathscr{S},t,k) \mid x \in L\}$. Finally, the history of the executed transition is updated to include the next unused integer. Given the above we define:

Definition 6 Given an MRPN $(P,T,\mathscr{A},\mathscr{A}_V,\mathscr{B},F)$, a state $\langle M,H\rangle$, a transition t that is enabled in state $\langle M,H\rangle$, and an enabling assignment $\mathscr{S}$, we write $\langle M,H\rangle \xrightarrow{(t,S)} \langle M',H'\rangle$ where for all $x \in P$:

$$\begin{aligned} M'(x) \;=\; & (M(x) - \bigcup_{a\in F(x,t)} \mathsf{con}(\mathscr{S}(a),M(x))) \\ & \cup \bigcup_{a\in F(t,x)} \mathsf{con}(\mathscr{S}(a),\mathsf{comp}_\mathsf{f}(t,\mathscr{S},M)) \oplus (\mathscr{S},t,k) \end{aligned}$$

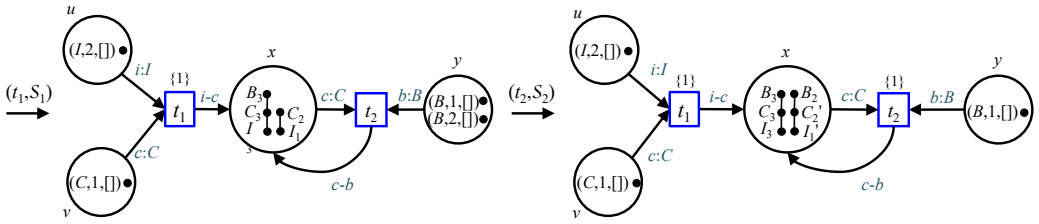


Figure 4: The effect of executing t_1 and t_2 in the net of Fig. 3, where $B_3 = (B,3,[])$, $C_3 = (C,3,[])$, $I_3 = (I,3,[])$, $I_1 = (I,1,[(1,t_1,i)])$, $C_2 = (C,2,[(1,t_1,c)]$, $I_1' = (I,1,[(1,t_1,i),(1,t_2,*)])$, $C_2' = (C,2,[(1,t_1,c),(1,t_2,c)])$, and $B_2 = (B,2,[(1,t_2,b)])$.

where $k = max(\{0\} \cup H(t)) + 1$ and

$$H'(t') = \begin{cases} H(t') \cup \{k\}, & \text{if } t' = t \\ H(t'), & \text{otherwise} \end{cases}$$

Fig. 4 shows the result of consecutively firing transitions t_1 and t_2 from the MRPN in Fig. 3 with enabling assignments $\mathscr{S}_1$, where $\mathscr{S}_1(i) = (I,1,[])$, $\mathscr{S}_1(c) = (C,2,[])$, and $\mathscr{S}_2$, where $\mathscr{S}_2(b) = (B,2,[])$, $\mathscr{S}_2(c) = (C,2,[(1,t_1,c)])$. We note the non-empty histories of the transitions depicted in the graphical representation, as well as the updates in the causal paths of the tokens.

3.2 Causal-order Reversing

We now move on to consider causal-order reversibility for MRPNs. In this form of reversibility, a transition can be reversed only if all its effects (if any), i.e. transitions that it has caused, have already been reversed. As argued in [24], two transition occurrences are causally dependent, if a token produced by the one was subsequently used to fire the other. Since token instances in MPRNs are associated with their causal path, we are able to identify the transitions that each token has participated in by observing its memory. Furthermore, if $\mathsf{last}(A_i) = (k,t,a)$ then the last transition that the token instance A_i has participated in was transition t and specifically its occurrence with history k.

Based on this observation, a transition occurrence t can be reversed in a certain state if the token/bonds instances it has employed have not engaged in any further transitions. Thus, we define causal reverse enabledness as follows.

Definition 7 Consider an MRPN $(P,T,\mathscr{A},\mathscr{A}_V,\mathscr{B},F)$, a state $\langle M,H\rangle$, and a transition t. We say that t is *co-enabled* in $\langle M,H\rangle$ if there exists a type-respecting assignment $\mathscr{R} : \mathsf{post}(t) \cap \mathscr{A}_V \to \mathscr{A}_I$ such that:

1. $\mathscr{R}(F(t,x)) \subseteq M(x)$ for all $x \in t\circ$, and
2. there exists $k \in H(t)$ such that for all $(A,i,xs) \in \bigcup_{x\in P} M(x)$ with $(k,t,b) \in xs$ for some b, $(k,t,b) = \mathsf{last}(A_i)$.

We refer to $\mathscr{R}$ as the *co*-reversal enabling assignment for the k^{th} occurrence of t.

Thus, a transition t is *co*-enabled in $\langle M,H\rangle$ for a specific occurrence k if there exists a type-respecting assignment of token instances on the variables of the outgoing arcs of the transition, which gives rise to a set of token and bond instances that are available in the relevant out-places and, additionally, these token/bond instances were last employed for the firing of the specific occurrence of the transition.

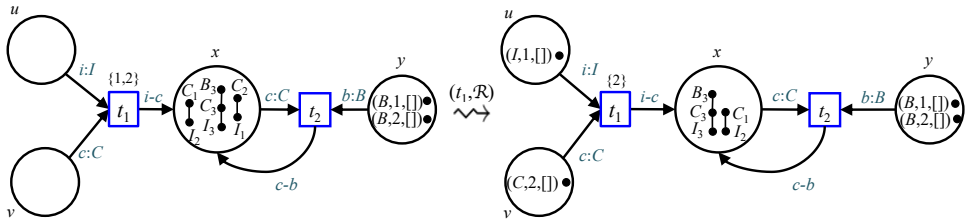


Figure 5: The effect of reversing transition t_1 with enabling assignment $\mathscr{R}(i) = I_1$, $\mathscr{R}(c) = C_2$, in a state following the execution of t_1 twice from the net in Fig. 3, first with enabling assignment $\mathscr{S}_1(i) = (I,1,[])$, $\mathscr{S}_1(c) = (C,2,[])$, and next with enabling assignment $\mathscr{S}_2(i) = (I,2,[])$, $\mathscr{S}_2(c) = (C,1,[])$ where we write $I_1 = (I,1,[(1,t_1,i])$, $I_2 = (I,2,[(2,t_1,i])$, $C_1 = (C,1,[(2,t_1,c])$, and $C_2 = (C,2,[(1,t_1,c])$.

To implement the reversal of a transition t according to a *co*-reversal enabling assignment $\mathscr{R}$, the selected token instances are relocated from the outgoing places of t to its incoming places, with bonds created and destroyed accordingly. The occurrence of the reversed transition is removed from its history.

Definition 8 Given an MRPN $(P,T,\mathscr{A},\mathscr{A}_V,\mathscr{B},F)$, a state $\langle M,H\rangle$, a transition t that is *co*-enabled with *co*-reversal enabling assignment $\mathscr{R}$ for the k^{th} occurrence of t, we write $\langle M,H\rangle \overset{(t,\mathscr{R})}{\rightsquigarrow} \langle M',H'\rangle$ where for all $x \in P$:

$$\begin{aligned} M'(x) &= (M(x) - \bigcup_{a\in F(t,x)} \mathsf{con}(\mathscr{R}(a),M(x))) \\ &\cup \bigcup_{a\in F(x,t)} \mathsf{init}(\mathsf{con}(\mathscr{R}(a),\mathsf{comp}_{\mathsf{r}}(t,\mathscr{R},M))) \end{aligned}$$

and

$$H'(t') = \begin{cases} H(t') - \{k\}, & \text{if } t' = t \\ H(t'), & \text{otherwise} \end{cases}$$

where $\mathsf{comp}_{\mathsf{r}}(t,\mathscr{R},M) = (\bigcup_{x\in t^\circ} M(x) \cup \mathscr{R}(\mathsf{eff}^-(t))) - \mathscr{R}(\mathsf{eff}^+(t))$.

In Fig. 5 we may observe the causal-order reversal of transition t_1. We note that the history information of the affected components is updated by removing the occurrence of the reversed transition and the history information of transition t_1 reflects that occurrence with identifier 1 has been reversed.

Let us now consider executions of both forward and backward moves and write $\longmapsto$ for $\longrightarrow \cup \rightsquigarrow$. We define the reachable states of an MRPN as follows.

Definition 9 Given an MRPN $(P,T,\mathscr{A},\mathscr{A}_V,\mathscr{B},F)$ and an initial state $\langle M_0,H_0\rangle$ we say that state $\langle M,H\rangle$ is *reachable*, if there exist $\langle M_i,H_i\rangle$, $i \le n$ for some $n \ge 0$, such that $\langle M_0,H_0\rangle \overset{(t_1,\mathscr{W}_1)}{\longmapsto} \langle M_1,H_1\rangle \overset{(t_2,\mathscr{W}_2)}{\longmapsto} \ldots \overset{(t_n,\mathscr{W}_n)}{\longmapsto} \langle M_n,H_n\rangle = \langle M,H\rangle$.

Furthermore, given a type A, an integer i, and a marking M, we write $\mathsf{num}(A,i,M)$ for the number of token instances of the form (A,i,xs) in M, defined by $\mathsf{num}(A,i,M) = |\{(x,A_i) \mid \exists x \in P, A_i \in M(x), A_i{\downarrow} = (A,i)\}|$. Similarly, for a bond instance $\beta_i \in \mathscr{B}_I$, we define $\mathsf{num}(\beta_i,M) = |\{x \in P \mid \beta_i \in M(x)\}|$. The following result confirms that in an execution beginning in the initial state of an MRPN, token instances are preserved, at most one bond instance may occur at any time, and a bond instance may be created/destroyed during a forward/reverse execution of a transition that features the bond as its effect.

Proposition 1 Given an MRPN $(P,T,\mathscr{A},\mathscr{A}_V,\mathscr{B},F)$, a reachable state $\langle M,H\rangle$, and a transition firing $\langle M,H\rangle \overset{(t,\mathscr{W})}{\longmapsto} \langle M',H'\rangle$, the following hold:

1. For all A, i, $\mathsf{num}(A,i,M') = \mathsf{num}(A,i,M)$=1.
2. For all $\beta_i \in \mathscr{B}_I$,
   (a) $0 \leq \mathsf{num}(\beta_i,M') \leq 1$,
   (b) if t is executed in the forward direction with forward enabling assignment $\mathscr{S}$ and $\beta_i \in \mathscr{S}(\mathsf{eff}^+(t))$ then $\mathsf{num}(\beta_i,M') = 1$; if instead $\beta_i \in \mathscr{S}(\mathsf{eff}^-(t))$ then $\mathsf{num}(\beta_i,M') = 0$, otherwise $\mathsf{num}(\beta_i,M) = \mathsf{num}(\beta_i,M')$.
   (c) if t is executed in the reverse direction with reverse enabling assignment $\mathscr{R}$ and $\beta_i \in \mathscr{R}(\mathsf{eff}^+(t))$ then $\mathsf{num}(\beta_i,M') = 0$; if instead $\beta_i \in \mathscr{R}(\mathsf{eff}^-(t))$ then $\mathsf{num}(\beta_i,M') = 1$, otherwise $\mathsf{num}(\beta_i,M) = \mathsf{num}(\beta_i,M')$.

Proof: The proof follows by induction on the length of the execution reaching state $\langle M,H\rangle$. If this is the initial state the result (i.e. clauses 1 and 2(a)) follows by our assumption on the initial state. For the induction step, let us assume that $\langle M,H\rangle$ satisfies the conditions of the proposition.

Let us begin with clause (1) and suppose $\overset{(t,\mathscr{W})}{\longmapsto} = \overset{(t,\mathscr{S})}{\longrightarrow}$, where $\mathscr{S}$ is the forward-enabling assignment for the transition, and let $A_i = (A,i,xs) \in \mathscr{A}_I$. Two cases exist:

1. $A_i \in \mathsf{con}(B_j,M(x))$ for some B_j, $\mathscr{S}(a) = B_j$, $a \in F(x,t)$. Note that x is unique by the assumption that $\mathsf{num}(A,i,M) = 1$. To discern the location of A_i in M' two cases exist.
   - Suppose $A_i \in \mathsf{con}(B_j,\mathsf{comp_f}(t,\mathscr{S},M))$. We observe that, by Definition 2(1), $a \in \mathsf{post}(t)$. Thus, there exists $y \in t\circ$, such that $a \in F(t,y)$. Note that this y is unique by Definition 2(2). As a result, by Definition 6, $\mathsf{con}(B_j,\mathsf{comp_f}(t,\mathscr{S},M)) \subseteq M'(y)$, which implies that $A_i \in M'(y)$.
   - Suppose $A_i \notin \mathsf{con}(B_j,\mathsf{comp_f}(t,\mathscr{S},M'))$ and consider $w = \langle (A_{i_1},A_{i_2}),\ldots,(A_{i_n},B_j)\rangle$, $A_i = A_{i_1}$, $n \geq 1$, such that $\mathsf{path}(A_i,w,M(x))$. Since $A_i \notin \mathsf{con}(B_j,\mathsf{comp_f}(t,\mathscr{S},M'))$ it must be that for some k, $(A_{i_{k-1}},A_{i_k}) \in \mathscr{S}(\mathsf{eff}^-(t))$ and $A_i \in \mathsf{con}(A_{i_k},M(x) - \mathscr{S}(\mathsf{eff}^-(t)))$. Using the same argument as in the previous case for A_{i_k} instead of B_j, we may conclude that $A_i \in M(y)$ such that $\mathscr{S}(b) = A_{i_k}$ and $b \in F(t,y)$.

   Now suppose that $A_i \in \mathsf{con}(C_k,\mathsf{comp_f}(t,\mathscr{S},M))$, $C_k = \mathscr{S}(b)$ for some $b \neq a$, $b \in F(t,y')$. Then it must be that $y = y'$. As a result, we have that $\mathsf{num}(A,i,M') = \mathsf{num}(A,i,M) = 1$ and the result follows.
2. $A_i \notin \mathsf{con}(\mathscr{S}(b),M(x))$ for all $b \in F(x,t)$, $x \in P$. This implies that $\{x \in P \mid A_i \in M'(x)\} = \{x \in P \mid A_i \in M(x)\}$ and the result follows.

Now suppose $\overset{(t,\mathscr{W})}{\longmapsto} = \overset{(t,\mathscr{R})}{\rightsquigarrow}$ where $\mathscr{R}$ is the reverse-enabling assignment of the transition. Consider $A_i = (A,i,xs) \in \mathscr{A}_I$. Two cases exist:

1. $A_i \in \mathsf{con}(B_j,M(x))$ for some B_j, $\mathscr{R}(a) = B_j$, $a \in F(t,x)$. Note that x is unique by the assumption that $\mathsf{num}(A,i,M) = 1$. To discern the location of A_i in M' two cases exist.
   - Suppose $A_i \in \mathsf{con}(B_j,\mathsf{comp_r}(t,\mathscr{R},M'))$. We observe that, by Definition 2(1), $a \in \mathsf{pre}(t)$. Thus, there exists $y \in \circ t$, such that $a \in F(y,t)$. Note that this y is unique by Definition 2(3). As a result, by Definition 8,

$$M'(y) = M(x) - \bigcup_{a \in F(t,x)} \mathsf{con}(\mathscr{R}(a),M(x))) \cup \bigcup_{a \in F(x,t)} \mathsf{init}(\mathsf{con}(\mathscr{R}(a),\mathsf{comp_r}(t,\mathscr{R},M)))$$

   Since $a \in F(y,t) \cap F(t,x)$, $A_i \in \mathsf{con}(\mathscr{R}(a),M(x) \cup F(y,t))$, which implies that $a \in M'(y)$.

- Suppose $A_i \notin \mathsf{con}(B_j, \mathsf{comp}_{\mathsf{r}}(t,\mathscr{R},M'))$ and consider $w = \langle (A_{i_1},A_{i_2}),\ldots,(A_{i_n},B_j)\rangle$, $A_i = A_{i_1}$, $n \geq 1$, such that $\mathsf{path}(A_i,w,M(x))$. Since $A_i \notin \mathsf{con}(B_j,\ \mathsf{comp}_{\mathsf{r}}(t,\mathscr{R},M'))$ it must be that for some k, $(A_{i_{k-1}},A_{i_k}) \in \mathscr{R}(\mathsf{eff}^+(t))$ and $A_i \in \mathsf{con}(A_{i_k}, M(x) - \mathscr{R}(\mathsf{eff}^+(t)))$. Using the same argument as in the previous case for A_{i_k} instead of B_j, we may conclude that $A_i \in M(y)$ such that $\mathscr{S}(b) = A_{i_k}$ and $b \in F(y,t)$.

  Now suppose that $A_i \in \mathsf{con}(C_k, \mathsf{comp}_{\mathsf{r}}(t,\mathscr{R},M))$, $C_k = \mathscr{R}(b)$ for some $a \neq b$, $b \in F(y',t)$. Then it must be that $y = y'$. As a result, we have that $\{z \in P \mid A_i \in M'(z)\} = \{y\}$ and the result follows.

2. $A_i \notin \mathsf{con}(\mathscr{R}(a), M(x))$ for all $a \in F(t,x)$, $x \in P$. This implies that $\{x \in P \mid A_i \in M'(x)\} = \{x \in P \mid A_i \in M(x)\}$ and the result follows.

The proof of clause 2 follows similar arguments. □

We may now proceed to establish the causal consistency of our semantics. We begin with defining when two states of an MRPN are considered to be causally equivalent. Intuitively, states $\langle M,H\rangle$ and $\langle M',H'\rangle$ are causally equivalent whenever the executions that have led to them contain the same causal paths. Note that these causal paths refer to different independent threads of computation, possibly executed through different interleavings in the executions leading to $\langle M,H\rangle$ and $\langle M',H'\rangle$. In our setting, we can enunciate this requirement by observing the causal histories of token instances and requiring that for each token instance of some type A in one of the two states there is a token instance of the same type that has participated in the exact same sequence of transitions in the other state:

Definition 10 Consider MRPN $(P,T,\mathscr{A},\mathscr{A}_V,\mathscr{B},F)$ and reachable states $\langle M,H\rangle$, $\langle M',H'\rangle$. Then the states are *causally equivalent*, denoted by $\langle M,H\rangle \asymp \langle M',H'\rangle$, if for each $x \in P$, $A_i \in M(x)$ there exists $A_j \in M'(x)$ with $\mathsf{cpath}(A_i) = \mathsf{cpath}(A_j)$, and vice versa.

We may now establish the Loop Lemma for our model.

Lemma 1 (Loop) For any forward transition $\langle M,H\rangle \xrightarrow{(t,\mathscr{S})} \langle M',H'\rangle$ there exists a backward transition $\langle M',H'\rangle \overset{(t,\mathscr{R})}{\rightsquigarrow} \langle M,H\rangle$ and for any backward transition $\langle M,H\rangle \overset{(t,\mathscr{R})}{\rightsquigarrow} \langle M',H'\rangle$ there exists a forward transition $\langle M',H'\rangle \xrightarrow{(t,\mathscr{S})} \langle M'',H''\rangle$ where $\langle M,H\rangle \asymp \langle M'',H''\rangle$.

Proof: Suppose $\langle M,H\rangle \xrightarrow{(t,\mathscr{S})} \langle M',H'\rangle$. Then t is clearly reverse-enabled in $\langle M',H'\rangle$ with reverse-enabling assignment $\mathscr{R}$ such that if $\mathscr{S}(a) = (A,i,xs)$, then $\mathscr{R}(a) = (A,i,xs+(t,k,a))$, where k is the maximum element of $H(t)$. Furthermore, $\langle M',H'\rangle \overset{t,\mathscr{R}}{\rightsquigarrow} \langle M'',H''\rangle$ where $H'' = H$. In addition, all token and bond instances involved in transition t (except those in $\mathsf{eff}^+(t)$) will be returned from the outgoing places of transition t back to its incoming places. At the same time, all destroyed bonds (those in $\mathsf{eff}^-(p)$) will be re-formed, according to Proposition 1. Specifically, for all $A_i \in \mathscr{A}_I$, it is easy to see by the definition of $\rightsquigarrow$ that $A_i \in M''(x)$ if and only if $A_i \in M(x)$. Similarly, for all $\beta_i \in \mathscr{B}_I$, $\beta_i \in M''(x)$ if and only if $\beta_i \in M(x)$. The opposite direction can be argued similarly, with the distinction that when a transition is executed immediately following its reversal, it is possible that the transition instance is assigned a different key, thus giving rise to a state $\langle M'',H''\rangle$ distinct but causally equivalent to $\langle M,H\rangle$. □

We now proceed to define some auxiliary notions. Given a transition $\langle M,H\rangle \overset{(t,\mathscr{W})}{\longmapsto} \langle M',H'\rangle$, we say that the *action* of the transition is $(t,\mathscr{W})$ if $\langle M,H\rangle \xrightarrow{(t,\mathscr{W})} \langle M',H'\rangle$ and $(\underline{t},\mathscr{W})$ if $\langle M,H\rangle \overset{(t,\mathscr{W})}{\rightsquigarrow} \langle M',H'\rangle$ and we may write $\langle M,H\rangle \overset{(\underline{t},\mathscr{W})}{\longmapsto} \langle M',H'\rangle$. We write Act_N for the set of all actions in an MRPN N. We use α to range over $\{t,\underline{t} \mid t \in T\}$ and write $\underline{\underline{t}} = t$. Given an execution $\langle M_0,H_0\rangle \overset{(\alpha_1,\mathscr{W}_1)}{\longmapsto} \ldots \overset{(\alpha_n,\mathscr{W}_n)}{\longmapsto} \langle M_n,H_n\rangle$, we say that the *trace* of the execution is $\sigma = \langle (\alpha_1,\mathscr{W}_1),(\alpha_2,\mathscr{W}_2),\ldots,(\alpha_n,\mathscr{W}_n)\rangle$, and write $\langle M,H\rangle \overset{\sigma}{\longmapsto}$

$\langle M_n, H_n\rangle$. Given $\sigma_1 = \langle(\alpha_1, \mathscr{W}_1), \ldots, (\alpha_k, \mathscr{W}_k)\rangle$, $\sigma_2 = \langle(\alpha_{k+1}, \mathscr{W}_{k+1}), \ldots, (\alpha_n, \mathscr{W}_n)\rangle$, we write $\sigma_1;\sigma_2$ for $\langle(\alpha_1, \mathscr{W}_1), \ldots, (\alpha_n, \mathscr{W}_n)\rangle$. We may also use the notation $\sigma_1;\sigma_2$ when σ_1 or σ_2 is a single transition. A central concept in what follows is causal equivalence on traces, a notion that employs the concept of concurrent transitions:

Definition 11 Consider an MRPN $(P, T, \mathscr{A}, \mathscr{A}_V, \mathscr{B}, F)$, a reachable state $\langle M, H\rangle$ and actions $(\alpha_1, \mathscr{W}_1)$ and $(\alpha_2, \mathscr{W}_2)$. Then $(\alpha_1, \mathscr{W}_1)$ and $(\alpha_2, \mathscr{W}_2)$ are said to be *concurrent* in state $\langle M, H\rangle$, if for all $u, v \in \mathscr{A}_V$, if $\mathscr{W}_1(u) = A_i$ and $\mathscr{W}_2(v) = B_j$, $A_i, B_j \in M(x)$ then $\mathsf{con}(A_i, M(x)) \neq \mathsf{con}(B_j, M(x))$.

Thus, two actions are concurrent when they employ different token instances. This notion captures when two actions are independent, i.e. the execution of the one does not preclude the other. Indeed, we may prove the following results.

Proposition 2 (Square Property) Consider an MRPN $(P, T, \mathscr{A}, \mathscr{A}_V, \mathscr{B}, F)$, a reachable state $\langle M, H\rangle$ and concurrent actions $(\alpha_1, \mathscr{W}_1)$ and $(\alpha_2, \mathscr{W}_2)$ in $\langle M, H\rangle$, such that $\langle M, H\rangle \overset{(\alpha_1, \mathscr{W}_1)}{\longmapsto} \langle M_1, H_1\rangle$ and $\langle M, H\rangle \overset{(\alpha_2, \mathscr{W}_2)}{\longmapsto} \langle M_2, H_2\rangle$. Then $\langle M_1, H_1\rangle \overset{(\alpha_2, \mathscr{W}_2)}{\longmapsto} \langle M', H'\rangle$ and $\langle M_2, H_2\rangle \overset{(\alpha_1, \mathscr{W}_1)}{\longmapsto} \langle M'', H''\rangle$, where $\langle M', H'\rangle \asymp \langle M'', H''\rangle$.

Proof: It is easy to see that since the two transitions involve distinct tokens then they can be executed in any order. If, additionally, $\alpha_1 \neq \alpha_2$ or $\alpha_1 = \alpha_2$ and they are both reverse transitions, then the effects imposed on the histories and the tokens of the transitions will be independent and the same in both cases, i.e. $\langle M', H'\rangle = \langle M', H'\rangle$. If instead $\alpha_1 = \alpha_2$ and α_1, α_2 are not both reverse transitions, then it is possible that distinct tokens will be assigned to the forward transition(s). Nonetheless, the sequence of actions executed by each token instance will be the same in both interleavings and, thus, the resulting states are causally equivalent. □

Proposition 3 (Reverse Transitions are Independent) Consider an MRPN $(P, T, \mathscr{A}, \mathscr{A}_V, \mathscr{B}, F)$, a state $\langle M, H\rangle$ and enabled reverse actions $(\underline{t_1}, \mathscr{R}_1)$ and $(\underline{t_2}, \mathscr{R}_2)$ where $(\underline{t_1}, \mathscr{R}_1) \neq (\underline{t_2}, \mathscr{R}_2)$. Then,$(\underline{t_1}, \mathscr{R}_1)$ and $(\underline{t_2}, \mathscr{R}_2)$ are concurrent.

Proof: It is straightforward to see that two distinct reverse transitions employ different tokens. This is because a token instance may only reverse the last transition occurrence in its history. Therefore $(\underline{t_1}, \mathscr{R}_1)$ and $(\underline{t_2}, \mathscr{R}_2)$ satisfy the requirement for being concurrent. □

We also define two transitions to be opposite in a certain state as follows:

Definition 12 Consider an MRPN $(P, T\mathscr{A}, \mathscr{A}_V, \mathscr{B}, F)$ and actions $(\alpha_1, \mathscr{W}_1)$ and $(\alpha_2, \mathscr{W}_2)$. Then $(\alpha_1, \mathscr{W}_1)$ and $(\alpha_2, \mathscr{W}_2)$ are said to be *opposite* if $\underline{\alpha_1} = \alpha_2$ and, if $\alpha_i = t$ for some t, for all $a \in \mathsf{pre}(t)$, $\mathsf{init}(\mathscr{W}_i(a)) = \mathscr{W}_{3-i}(a)$.

Note that this may arise exactly when the two actions are forward and reverse executions of the same transition and using the same token instances. We are now ready to define when two traces are causally equivalent.

Definition 13 Consider a reachable state $\langle M, H\rangle$. Then *causal equivalence on traces with respect to* $\langle M, H\rangle$, denoted by $\sigma_1 \asymp_{\langle M,H\rangle} \sigma_2$, is the least equivalence relation on traces such that (i) $\sigma_1 = \sigma;(\alpha_1, \mathscr{W}_1);(\alpha_2, \mathscr{W}_2);\sigma'$ where $\langle M, H\rangle \overset{\sigma}{\longmapsto} \langle M', H'\rangle$ and if $(\alpha_1, \mathscr{W}_1)$ and $(\alpha_2, \mathscr{W}_2)$ are concurrent in $\langle M', H'\rangle$ then $\sigma_2 = \sigma;(\alpha_2, \mathscr{W}_2);(\alpha_1, \mathscr{W}_1);\sigma'$, and (ii) if $(\alpha_1, \mathscr{W}_1)$ and $(\alpha_2, \mathscr{W}_2)$ are opposite transitions then $\sigma_2 = \sigma;\varepsilon;\sigma'$.

We may now establish the Parabolic Lemma, which states that causal equivalence allows the permutation of reverse and forward transitions that have no causal relations between them. Therefore, computations are allowed to reach for the maximum freedom of choice going backward and then continue forward.

Lemma 2 (Parabolic Lemma) Consider an MRPN $(P,T,\mathscr{A},\mathscr{A}_V,\mathscr{B},F)$, a reachable state $\langle M,H\rangle$, and an execution $\langle M,H\rangle \xmapsto{\sigma} \langle M',H'\rangle$. Then there exist traces r,r' both forward such that $\sigma \asymp_{\langle M,H\rangle} \underline{r};r'$ and $\langle M,H\rangle \xmapsto{\underline{r};r'} \langle M'',H''\rangle$ where $\langle M',H'\rangle \asymp \langle M'',H''\rangle$.

Proof Following [17], given the satisfaction of the Square Property (Proposition 2) and the independence of reverse transitions (Proposition 3), we conclude that the lemma holds. A proof from first principles may also be found in [30]. □

We conclude with Theorem 1 stating that two computations beginning in the same state lead to equivalent states if and only if the two computations are causally equivalent. This guarantees the consistency of the approach since reversing transitions in causal order is in a sense equivalent to not executing the transitions in the first place. Reversal does not give rise to previously unreachable states, on the contrary, it gives rise to causally-equivalent states due to different keys being possibly assigned to concurrent transitions.

Theorem 1 Consider an MRPN $(P,T,\mathscr{A},\mathscr{A}_V,\mathscr{B},F)$, a reachable state $\langle M,H\rangle$, and traces σ_1, σ_2 such that $\langle M,H\rangle \xmapsto{\sigma_1} \langle M_1,H_1\rangle$ and $\langle M,H\rangle \xmapsto{\sigma_2} \langle M_2,H_2\rangle$. Then, $\sigma_1 \asymp_{\langle M,H\rangle} \sigma_2$ if and only if $\langle M_1,H_1\rangle \asymp \langle M_2,H_2\rangle$.

Proof: Following [17], given the satisfaction of the Parabolic Lemma and the fact that the model does not allow infinite reverse computations, we conclude that the theorem holds. A proof from first principles may also be found in [30]. □

4 Multi Tokens versus Single Tokens

We now proceed to define Single Reversing Petri Nets as MRPNs where each token type corresponds to exactly one token instance.

Definition 14 A *Single Reversing Petri Net* (SRPN) $(P,T,\mathscr{A},\mathscr{A}_V,\mathscr{B},F)$ is an MRPN where for all $A \in \mathscr{A}$, $|A| = 1$.

Forward and causal-order reversal for SRPNs is defined as for MRPNs. Consequently, SPRNs are special instances of MRPNs. In the sequel, we will show that for each MRPN there is an "equivalent" SRPN. To achieve this, similarly to [31], we will employ Labelled Transition Systems defined as follows:

Definition 15 A labelled transition system (LTS) is a tuple $(Q,E,\rightarrow,I)$ where:

- Q is a countable set of states,
- E is a countable set of actions,
- $\rightarrow \subseteq Q \times E \times Q$ is the step transition relation, where we write $p \xrightarrow{u} q$ for $(p,u,q) \in \rightarrow$, and
- $I \in Q$ is the initial state.

For the purposes of our comparison, we will employ LTSs in the context of isomorphism of reachable parts:

Definition 16 Two LTSs $L_1 = (Q_1,E_1,\rightarrow_1,I_1)$ and $L_2 = (Q_2,E_2,\rightarrow_2,I_2)$ are isomorphic, written $L_1 \cong L_2$, if they differ only in the names of their states and events, i.e. if there are bijections $\gamma : Q_1 \rightarrow Q_2$ and $\eta : E_1 \rightarrow E_2$ such that $\gamma(I_1) = I_2$, and, for $p,q \in Q_1$, $u \in E_1 : \gamma(p) \xrightarrow{\eta(u)}_2 \gamma(q)$ iff $p \xrightarrow{u}_1 q$.

The set $\mathscr{R}(Q)$ of reachable states in $L = (Q, E, \rightarrow, I)$ is the smallest set such that I is reachable and whenever p is reachable and $p \xrightarrow{u} q$ then q is reachable. The reachable part of L is the LTS $\mathscr{R}(L) = (R(Q), E, \rightarrow_{\mathscr{R}}, I)$, where $\rightarrow_{\mathscr{R}}$ is the part of the transition relation restricted to reachable states. We write $L_1 \cong_{\mathscr{R}} L_2$ if $\mathscr{R}(L_1)$ and $\mathscr{R}(L_2)$ are isomorphic. To check $L_1 \cong_{\mathscr{R}} L_2$ it suffices to restrict to subsets of Q_1 and Q_2 that contain all reachable states, and construct an isomorphism between the resulting LTSs.

We proceed to give a translation from MRPNs to SPRNs. First, we present how an LTS can be associated with an MRPN/SRPN structure.

Definition 17 Let $N = (P, T, \mathscr{A}, \mathscr{A}_V, \mathscr{B}, F)$ be an MRPN (or SRPN) with initial marking M_0. Then $\mathscr{H}(N, M_0) = ((P \rightarrow 2^{\mathscr{A}_I \cup \mathscr{B}_I}) \times (T \rightarrow 2^{\mathbb{N}}), Act, \longmapsto, \langle M_0, H_0 \rangle)$ is the LTS associated with N.

We may now establish that for any MRPN there exists an SPRN with an isomorphic LTS.

Theorem 2 For every MRPN $N = (P, T, \mathscr{A}, \mathscr{A}_V, \mathscr{B}, F)$ with initial marking M_0 there exists an SRPN $N' = (P, T', \mathscr{A}', \mathscr{A}'_V, \mathscr{B}', F')$ with initial marking M'_0 such that $\mathscr{H}(N, M_0) \cong_{\mathscr{R}} \mathscr{H}(N', M'_0)$.

Proof: Let $N = (P, T, \mathscr{A}, \mathscr{A}_V, \mathscr{B}, F)$ be an MRPN with initial state $\langle M_0, H_0 \rangle$. We introduce the notation $\mathscr{W}\downarrow$ where for any type-respecting assignment $\mathscr{W}$, $\mathscr{W}\downarrow(a) = \mathscr{W}(a)\downarrow$, that is $\mathscr{W}$ assigns to a variable in the range of $\mathscr{W}$ the token instance associated to it by $\mathscr{W}$ but with its history removed. Furthermore, if $f = \mathscr{W}\downarrow$ we write $f_s(a) = a_i$ if $f(a) = (A, i)$. We construct $N' = (P, T', \mathscr{A}', \mathscr{A}'_V, \mathscr{B}', F')$ with initial state $\langle M'_0, H'_0 \rangle$ as follows:

$$
\begin{array}{rcl}
\mathscr{A}' & = & \{A_i \mid \exists (A, i, []) \in M_0(x) \text{ for some } x \in P\} \\
\mathscr{A}'_V & = & \{a_i \mid A_i \in \mathscr{A}'\} \\
\mathscr{B}' & = & \{(A_i, B_j) \mid A_i, B_j \in \mathscr{A}', (A, B) \in \mathscr{B}\} \\
T' & = & \{t_{\mathscr{W}\downarrow} \mid t \in T, \mathscr{W} : \mathsf{pre}(t) \rightarrow \mathscr{A}_I \text{ is a type-respecting assignment } \} \\
F'(x_1, x_2) & = & \{f_s(a) \mid \exists i \in \{1,2\}, x_i = t_f \in T', \text{ and } a \in \mathsf{pre}(t)\} \\
 & \cup & \{(f_s(a), f_s(b)) \mid \exists i \in \{1,2\}, x_i = t_f \in T', (a, b) \in \mathsf{pre}(t)\} \\
M'_0(x) & = & \{(A_i, 1, []) \mid (A, i, []) \in M_0(x)\},\ \forall x \in P \\
H'_0(t) & = & \emptyset,\ \forall t \in T'
\end{array}
$$

The above construction, projects each type A in N to a set of types A_i in N' such that, $A_i \in \mathscr{A}'$ for each instance $(A, i, [])$ of type A in M_0. Type A_i contains exactly one element, initially named $(A_i, 1, [])$. Furthermore, for each transition $t \in T$, we create a set of transitions of the form $t_f \in T'$, to associate all possible ways in which token/bond instances may be taken as input by t with a distinct transition that takes as input the combination of types projected to by the instances.

We now proceed to define bijections γ and η for establishing the homomorphism between the two LTSs. To simplify the proof, we assume that during the execution of transitions the enabling assignment is recorded both in the transition histories, i.e. given a transition t we have $(k, \mathscr{W}) \in H(t)$ signifying that the k^{th} occurrence of t was executed with enabling assignment $\mathscr{W}$ and also in a token instance (A, i, xs) elements of xs have the form $(k, t, v, \mathscr{W}\downarrow)$ again recording the assignment that enabled the specific execution of the transition occurrence. In this setting, it is easy to associate each token instance of N to a token instance of N' as follows, where we write $st(A_i)$ for the equivalent token instance of A_i in the SPRN N':

$st((A, i, xs)) = (A_i, 1, ys)$

where, if $xs = [(k^i, t^i, v^i, f^i)]_{1 \leq i \leq n}$ then $ys = [(|\{(k, t^i, v, f) \mid \exists k, v, f \text{ s.t.} (k, t^i, v, f) \in ys\}|, t^i_f, f_s(a))]_{1 \leq i \leq n}$.

For any reachable state $\langle M,H\rangle$ in LTS $\mathscr{H}(N,M_0)$, we define $\gamma(\langle M,H\rangle) = \langle M',H'\rangle$ such that for all $x \in P$ and $t_f \in T'$

$$\begin{aligned} M'(x) &= \{st(A_i) \mid A_i \in M(x)\} \cup \{(st(A_i), st(B_j)) \mid (A_i,B_j) \in M(x)\} \\ H'(t_f) &= \{1,\ldots,k \mid k = |\{(i,\mathscr{R}) \in H(t) \mid \mathscr{R}\downarrow = f\}|\} \end{aligned}$$

Furthermore, given an action $(t,\mathscr{W})$, we write

$$\eta((t,\mathscr{W})) = (t_{\mathscr{W}\downarrow}, \mathscr{W}')$$

where if $\mathscr{W}(a) = A_i$ then $\mathscr{W}'(a_i) = st(A_i)$.

Based on these, we may confirm that there exists an isomorphism between the LTSs $\mathscr{H}(N,M_0)$ and $\mathscr{H}(N',M_0')$ as follows. Suppose $\langle M_m,H_m\rangle$ is a reachable state of $\mathscr{H}(N,M_0)$ with $\gamma(\langle M_m,H_m\rangle) = \langle M_s,H_s\rangle$. Two cases exist:

- Suppose $\langle M_m,H_m\rangle \xrightarrow{(t,\mathscr{S})} \langle M_m',H_m'\rangle$. This implies that t is a forward-enabled transition with forward-enabling assignment $\mathscr{S}$. Consider $\eta(t,\mathscr{S}) = (t_{\mathscr{S}\downarrow},\mathscr{S}')$, as defined above. It is easy to see that $t_{\mathscr{S}\downarrow}$ is also a forward-enabled transition in $\langle M_s,H_s\rangle$ with forward-enabling assignment $\mathscr{S}'$. Furthermore, if $\langle M_s,H_s\rangle \xrightarrow{(t_{\mathscr{S}\downarrow},\mathscr{S}')} \langle M_s',H_s'\rangle$, then

$$\begin{aligned} M_s'(x) &= (M_s(x) - \bigcup_{a\in F'(x,t_{\mathscr{S}\downarrow})} \mathsf{con}(\mathscr{S}'(a), M'(x))) \\ &\cup \bigcup_{a\in F'(t_{\mathscr{S}\downarrow},x)} \mathsf{con}(\mathscr{S}'(a), \mathsf{comp_f}(t_{\mathscr{S}\downarrow},\mathscr{S}',M_s)) \oplus (\mathscr{S}',t_{\mathscr{S}},k) \\ &= (M_s(x) - \bigcup_{a\in F(x,t)} \{st(A_i), (st(A_i),st(B_j)) \mid A_i,(A_i,B_j) \in \mathsf{con}(\mathscr{S}(a), M_m(x))) \\ &\cup \bigcup_{a\in F(t,x)} \{st(A_i), (st(A_i),st(B_j)) \mid A_i,(A_i,B_j) \in \\ &\qquad\qquad \mathsf{con}(\mathscr{S}(a), \mathsf{comp_f}(t,\mathscr{S},M_m)) \oplus (\mathscr{S},t,k)\} \end{aligned}$$

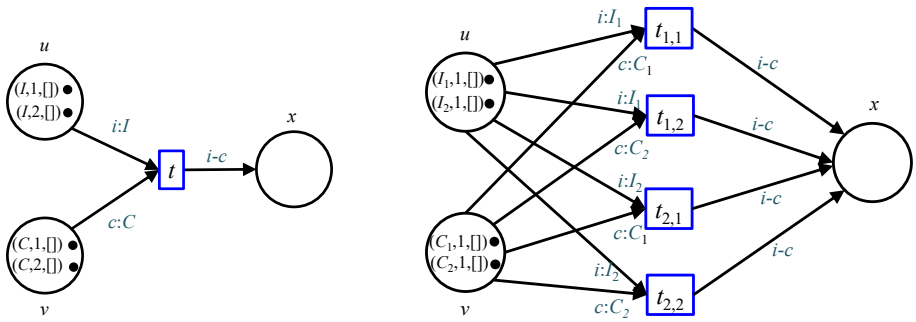
  where $k = max(\{0\} \cup \{k' | k' \in H(t)\}) + 1$ and

$$H'(t) = \begin{cases} H(t) \cup \{k\}, & \text{if } t = t_{\mathscr{S}\downarrow} \\ H(t'), & \text{otherwise} \end{cases}$$

  We may see that $\gamma(\langle M_m,H_m\rangle) = \langle M_s,H_s\rangle$, and the result follows. Reversing the arguments, we may also prove the opposite direction.

- Suppose $\langle M_m,H_m\rangle \overset{(t,\mathscr{R})}{\rightsquigarrow} \langle M_m',H_m'\rangle$. This implies that t is a reverse enabled transition with enabling assignment $\mathscr{R}$. Consider $\eta(t,\mathscr{R}) = (t_{\mathscr{R}\downarrow},\mathscr{R}')$, as defined above. It is easy to see that $t_{\mathscr{R}\downarrow}$ is also a reverse-enabled transition in $\langle M_s,H_s\rangle$ with reverse-enabling assignment $\mathscr{R}'$. Furthermore, if $\langle M_s,H_s\rangle \overset{(t_{\mathscr{R}\downarrow},\mathscr{R}')}{\rightsquigarrow} \langle M_s',H_s'\rangle$, then using similar arguments as in the previous case we may confirm that $\gamma(\langle M_m,H_m\rangle) = \langle M_s,H_s\rangle$. The same holds for the opposite direction. This completes the proof. □

In Fig. 6 we present an MRPN N and its respective SRPN N'. From N we obtain N' by constructing the new token types I_1,I_2,C_1,C_2 and exactly one token instance of each of these types. The places are the same in both RPN models. The transitions required for the SRPN are dependent on the types of the



(a) MRPN N (b) Equivalent SRPN N'

Figure 6: Translating MRPNs to SRPNs

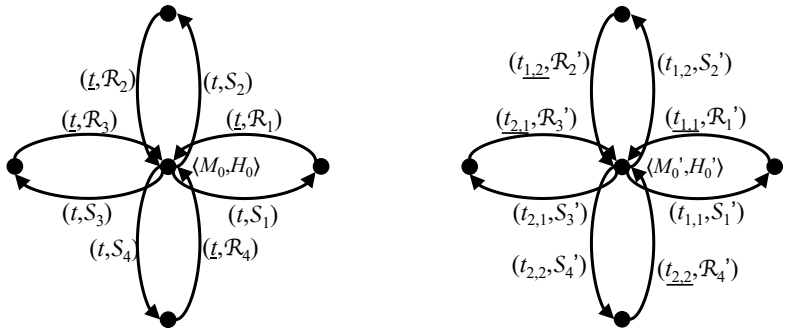
variables required for each MRPN transition and the token instances representing that type. Specifically for each token-instance combination that may fire a transition in the MRPN, a respective transition is required in the SRPN. In the example, two token instances of type I can be instantiated to variable i and two token instances of type C can be instantiated to variable v. This yields four combinations of token instances resulting in four different transitions.

In Fig. 7 we may see the isomorphic LTSs of the two RPNs, where

$$\begin{aligned} t_{1,1} &= t_{\mathscr{S}_1\downarrow} & \underline{t_{1,1}} &= \underline{t_{\mathscr{R}_1\downarrow}} \\ t_{1,2} &= t_{\mathscr{S}_2\downarrow} & \underline{t_{1,2}} &= \underline{t_{\mathscr{R}_2\downarrow}} \\ t_{2,1} &= t_{\mathscr{S}_3\downarrow} & \underline{t_{2,1}} &= \underline{t_{\mathscr{R}_3\downarrow}} \\ t_{2,2} &= t_{\mathscr{S}_4\downarrow} & \underline{t_{2,3}} &= \underline{t_{\mathscr{R}_4\downarrow}} \end{aligned}$$

and the enabling assignments of the actions in the two LTSs are

$$\begin{aligned} \mathscr{S}_1(i) &= (I,1,[]), & \mathscr{S}_1(c) &= (C,1,[]) \\ \mathscr{R}_1(i) &= (I,1,[(t,1,i)]), & \mathscr{R}_1(c) &= (C,1,[(t,1,c)]) \\ \mathscr{S}_2(i) &= (I,1,[]), & \mathscr{S}_2(c) &= (C,2,[]) \\ \mathscr{R}_2(i) &= (I,1,[(t,1,i)]), & \mathscr{R}_2(c) &= (C,2,[(t,1,c)]) \\ \mathscr{S}_3(i) &= (I,2,[]), & \mathscr{S}_3(c) &= (C,1,[]) \\ \mathscr{R}_3(i) &= (I,2,[(t,1,i)]), & \mathscr{R}_3(c) &= (C,1,[(t,1,c)]) \\ \mathscr{S}_4(i) &= (I,2,[]), & \mathscr{S}_4(c) &= (C,2,[]) \\ \mathscr{R}_4(i) &= (I,2,[(t,1,i)]), & \mathscr{R}_4(c) &= (C,2,[(t,1,c)]) \end{aligned}$$



(a) LTS of net of Fig. 6(a) (b) LTS of net of Fig. 6(b)

Figure 7: Isomorphic LTSs of an MRPN and its SPRN translation

and

$$\begin{aligned}
\mathscr{S}_1'(i) &= (I_1, 1, []), & \mathscr{S}_1(c) &= (C_1, 1, []) \\
\mathscr{R}_1(i) &= (I_1, 1, [(t, 1, i)]), & \mathscr{R}_1(c) &= (C_1, 1, [(t, 1, c)]) \\
\mathscr{S}_2(i) &= (I_1, 1, []), & \mathscr{S}_2(c) &= (C_2, 1, []) \\
\mathscr{R}_2(i) &= (I_1, 1, [(t, 1, i)]), & \mathscr{R}_2(c) &= (C_2, 1, [(t, 1, c)]) \\
\mathscr{S}_3(i) &= (I_2, 1, []), & \mathscr{S}_3(c) &= (C_1, 1, []) \\
\mathscr{R}_3(i) &= (I_2, 1, [(t, 1, i)]), & \mathscr{R}_3(c) &= (C_1, 1, [(t, 1, c)]) \\
\mathscr{S}_4(i) &= (I_2, 1, []), & \mathscr{S}_4(c) &= (C_2, 1, []) \\
\mathscr{R}_4(i) &= (I_2, 1, [(t, 1, i)]), & \mathscr{R}_4(c) &= (C_2, 1, [(t, 1, c)])
\end{aligned}$$

5 Conclusions

This paper presents an extension of RPNs with multiple tokens of the same type based on the individual token interpretation. The individuality of tokens is enabled by recording their causal path, while the semantics allows identical tokens to fire any eligible transition when going forward, but only the transitions they have been previously involved in when going backward. We have presented a semantics for causal-order reversibility, which unlike the semantics presented in [24] is purely local and requires no global control. Another contribution of the paper is a result illustrating that introducing multiple tokens in the model does not increase its expressive power. Indeed, for every MRPN we may construct an equivalent SRPN, which preserves its computation. In related work [30], MRPNs have also been associated with backtracking and out-of-causal-order semantics and it was shown that in all settings MRPNs are equivalent to the original RPN model.

In our current work we are developing a tool for simulating and verifying RPN models [9], which we aim to apply towards the analysis of resource-aware systems. Our experience in applying RPNs in the context of wireless communications [25] has illustrated that resource management can be studied and understood in terms of RPNs since, along with their visual nature, they offer a number of features, such as token persistence, that is especially relevant in these contexts. In future work, we would like to further apply our framework in the specific fields as well as in the field of long-running transactions.

References

[1] Bogdan Aman & Gabriel Ciobanu (2018): *Bonding calculus*. *Natural Computing* 17(4), pp. 823–832, doi:10.1007/s11047-018-9709-7.

[2] Kamila Barylska, Anna Gogolinska, Lukasz Mikulski, Anna Philippou, Marcin Piatkowski & Kyriaki Psara (2018): *Reversing Computations Modelled by Coloured Petri Nets*. In: *Proceedings of ATAED 2018*, CEUR Workshop Proceedings 2115, pp. 91–111. Available at `http://ceur-ws.org/Vol-2115/ATAED2018-91-111.pdf`.

[3] Kamila Barylska, Anna Gogolinska, Lukasz Mikulski, Anna Philippou, Marcin Piatkowski & Kyriaki Psara (2022): *Formal Translation from Reversing Petri Nets to Coloured Petri Nets*. In: *Proceedings of RC 2022*, *LNCS* 13354, Springer, pp. 172–186, doi:10.1007/978-3-031-09005-9_12.

[4] Kamila Barylska, Maciej Koutny, Lukasz Mikulski & Marcin Piatkowski (2018): *Reversible computation vs. reversibility in Petri nets*. *Science of Computer Programming* 151, pp. 48–60, doi:10.1016/j.scico.2017.10.008.

[5] Kamila Barylska, Lukasz Mikulski, Marcin Piatkowski, Maciej Koutny & Evgeny Erofeev (2016): *Reversing Transitions in Bounded Petri Nets*. In: *Proceedings of CS&P 2016*, CEUR Workshop Proceedings 1698, CEUR-WS.org, pp. 74–85.

[6] Adel Benamira (2020): *Causal Reversibility in Individual Token Interpretation of Petri Nets*. *The Computer Science Journal* 21(4), doi:10.7494/csci.2020.21.4.3728.

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

[8] Vincent Danos & Jean Krivine (2005): *Transactions in RCCS*. In: *Proceedings of CONCUR 2005*, LNCS 3653, Springer, pp. 398–412, doi:10.1007/11539452_31.

[9] Yannis Dimopoulos, Eleftheria Kouppari, Anna Philippou & Kyriaki Psara (2020): *Encoding Reversing Petri Nets in Answer Set Programming*. In: *Proceedings of RC 2020*, LNCS 12227, Spinger, pp. 264–271, doi:10.1007/978-3-030-52482-1_17.

[10] Rob J. van Glabbeek (2005): *The Individual and Collective Token Interpretations of Petri Nets*. In: *Proceedings of CONCUR 2005*, LNCS 3653, Springer, pp. 323–337, doi:10.1007/11539452_26.

[11] Rob J. van Glabbeek, Ursula Goltz & Jens-Wolfhard Schicke (2021): *On Causal Semantics of Petri Nets*. *CoRR* abs/2103.00729. Available at `https://arxiv.org/abs/2103.00729`.

[12] Rob J. van Glabbeek & Gordon D. Plotkin (1995): *Configuration Structures*. In: *Proceedings of LICS 1995*, IEEE Computer Society, pp. 199–209, doi:10.1109/LICS.1995.523257.

[13] Jetty Kleijn & Maciej Koutny (2002): *Causality semantics of Petri nets with weighted inhibitor arcs*. In: *Proceedings of CONCUR 2002*, LNCS 2421, Springer, pp. 531–546, doi:10.1007/3-540-45694-5_35.

[14] Stefan Kuhn, Bogdan Aman, Gabriel Ciobanu, Anna Philippou, Kyriaki Psara & Irek Ulidowski (2020): *Reversibility in Chemical Reactions*. In: *Reversible Computation: Extending Horizons of Computing - Selected Results of the COST Action IC1405*, LNCS 1270, Spinger, pp. 151–176, doi:10.1007/978-3-030-47361-7_7.

[15] Stefan Kuhn & Irek Ulidowski (2018): *Local reversibility in a Calculus of Covalent Bonding*. *Science of Computer Programming* 151(Supplement C), pp. 18–47, doi:10.1016/j.scico.2017.09.008.

[16] Ivan Lanese, Claudio Antares Mezzina & Jean-Bernard Stefani (2016): *Reversibility in the higher-order π-calculus*. *Theoretical Computer Science* 625, pp. 25–84, doi:10.1016/j.tcs.2016.02.019.

[17] Ivan Lanese, Iain C. C. Phillips & Irek Ulidowski (2020): *An Axiomatic Approach to Reversible Computation*. In: *Proceedings of FOSSACS 2020*, *LNCS 12077* 12077, Springer, pp. 442–461, doi:10.1007/978-3-030-45231-5_23.

[18] Y. Edmund Lien (1976): *A note on transition systems*. *Information Sciences* 10(4), pp. 347–362, doi:10.1016/0020-0255(76)90054-2.

[19] Hernán C. Melgratti, Claudio Antares Mezzina, Iain Phillips, G. Michele Pinna & Irek Ulidowski (2020): *Reversible Occurrence Nets and Causal Reversible Prime Event Structures*. In: *Proceedings of RC 2020*, LNCS 12227, Spinger, pp. 19–36, doi:10.1007/978-3-030-52482-1_2.

[20] Hernán C. Melgratti, Claudio Antares Mezzina & Irek Ulidowski (2020): *Reversing Place Transition Nets*. *Logical Methods in Computer Science* 16(4). Available at `https://lmcs.episciences.org/6843`.

[21] Lukasz Mikulski & Ivan Lanese (2019): *Reversing Unbounded Petri Nets*. In: *Proceedings of PETRI NETS 2019*, LNCS 11522, Springer, pp. 213–233, doi:10.1007/978-3-030-21571-2_13.

[22] Anna Philippou & Kyriaki Psara (2018): *Reversible Computation in Petri Nets*. In: *Proceedings of RC 2018*, LNCS 11106, Springer, pp. 84–101, doi:10.1007/978-3-319-99498-7_6.

[23] Anna Philippou & Kyriaki Psara (2022): *A collective interpretation semantics for reversing Petri nets*. *Theoretical Computer Science* 924, pp. 148–170, doi:10.1016/j.tcs.2022.05.016.

[24] Anna Philippou & Kyriaki Psara (2022): *Reversible computation in nets with bonds*. *Journal of Logical and Algebraic Methods in Programming* 124, p. 100718, doi:10.1016/j.jlamp.2021.100718.

[25] Anna Philippou, Kyriaki Psara & Harun Siljak (2019): *Controlling Reversibility in Reversing Petri Nets with Application to Wireless Communications*. In: *Proceedings of RC 2019*, LNCS 11497, Springer, pp. 238–245, doi:10.1007/978-3-030-21500-2_15.

[26] Iain Phillips & Irek Ulidowski (2007): *Reversing algebraic process calculi*. *Journal of Logic and Algebraic Programming* 73(1-2), pp. 70–96, doi:10.1016/j.jlap.2006.11.002.

[27] Iain Phillips & Irek Ulidowski (2015): *Reversibility and asymmetric conflict in event structures*. *Journal of Logical and Algebraic Methods in Programming* 84(6), pp. 781–805, doi:10.1016/j.jlamp.2015.07.004.

[28] Iain Phillips, Irek Ulidowski & Shoji Yuen (2012): *A Reversible Process Calculus and the Modelling of the ERK Signalling Pathway*. In: *Proceedings of RC 2012*, LNCS 7581, Springer, pp. 218–232, doi:10.1007/978-3-642-36315-3_18.

[29] Iain Phillips, Irek Ulidowski & Shoji Yuen (2013): *Modelling of Bonding with Processes and Events*. In: *Proceedings of RC 2013*, LNCS 7947, Springer, pp. 141–154, doi:10.1007/978-3-642-38986-3_12.

[30] Kyriaki Psara (2020): *Reversible Computation in Petri Nets*. Ph.D. thesis, Department of Computer Science, University of Cyprus. Available at `https://arxiv.org/abs/2101.07066`.

[31] Wolfgang Reisig (1985): *Petri Nets with Individual Tokens*. *Theoretical Computer Science* 41, pp. 185–213, doi:10.1016/0304-3975(85)90070-2.

[32] Grzegorz Rozenberg & Joost Engelfriet (1996): *Elementary net systems*. In: *Advanced Course on Petri Nets*, LNCS 1491, Springer, pp. 12–121, doi:10.1007/3-540-65306-6_14.

[33] Irek Ulidowski, Iain Phillips & Shoji Yuen (2014): *Concurrency and Reversibility*. In: *Proceedings of RC 2014*, LNCS 8507, Springer, pp. 1–14, doi:10.1007/978-3-319-08494-7_1.