EPTCS 322

Proceedings of the

Combined 27th International Workshop on Expressiveness in Concurrency

and 17th Workshop on Structural Operational Semantics

Online, 31 August 2020

Edited by: Ornela Dardha and Jurriaan Rot

Published: 27th August 2020
DOI: 10.4204/EPTCS.322
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Preface

This volume contains the proceedings of EXPRESS/SOS 2020, the Combined 27th International Workshop on Expressiveness in Concurrency and the 17th Workshop on Structural Operational Semantics. Following a long tradition, EXPRESS/SOS 2020 was held as one of the affiliated workshops of the 31st International Conference on Concurrency Theory (CONCUR 2020), originally planned to be held in Vienna, Austria. Due to the covid-19 pandemic, it was instead held online as a virtual workshop.

The EXPRESS/SOS workshop series aims at bringing together researchers interested in the formal semantics of systems and programming concepts, and in the expressiveness of computational models. In particular, topics of interest for the workshop include (but are not limited to):

- expressiveness and rigorous comparisons between models of computation (process algebras, event structures, Petri nets, rewrite systems);
- expressiveness and rigorous comparisons between programming languages and models (distributed, component-based, object-oriented, service-oriented);
- logics for concurrency (modal logics, probabilistic and stochastic logics, temporal logics and resource logics);
- analysis techniques for concurrent systems;
- theory of structural operational semantics (metatheory, category-theoretic approaches, congruence results);
- 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 year, the Program Committee selected 6 submissions for inclusion in the scientific program - five full papers and the following short paper:

- *Process, Systems and Tests: Three Layers in Concurrent Computation*, by Clément Aubert and Daniele Varacca.

This volume contains revised versions of the given full papers, as well as (extended) abstracts associated to the following three invited presentations, which nicely complemented the scientific program:

- *Quantitative Algebraic Reasoning: an Overview*, by Giorgio Bacci (Aalborg University, Denmark)
- *Divergence-Preserving Branching Bisimilarity*, by Bas Luttik (TU Eindhoven, The Netherlands)
- *Multiparty Session Programming with Global Protocol Combinators*, by Rumyana Neykova (Brunel University London, UK)

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 2020 organizing committee 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.

Ornela Dardha and Jurriaan Rot,
August 2020

Program Committee

- Pedro R. D'Argenio, University of Cordoba, Argentina
- Stephanie Balzer, Carnegie Mellon University, US
- Valentina Castiglioni, Reykjavik University, Iceland
- Ornela Dardha (co-chair), University of Glasgow, UK
- Mariangiola Dezani-Ciancaglini, University of Torino, Italy
- Rob van Glabbeek, Data61, CSIRO, Australia
- Sophia Knight, University of Minnesota Duluth, US
- Uwe Nestmann, TU Berlin, Germany
- Catuscia Palamidessi, Inria and École Polytechnique, France
- Marco Peressotti, University of Southern Denmark
- Jorge A. Pérez, University of Groningen, The Netherlands
- Jurriaan Rot (co-chair), Radboud University, The Netherlands
- Ivano Salvo, Sapienza, Rome, Italy

Additional reviewers

- Johannes Åman Pohjola
- Frank Valencia

Quantitative Algebraic Reasoning: an Overview

Giorgio Bacci
Aalborg University, Denmark
Department of Computer Science
grbacci@cs.aau.dk

Some Context and Motivations. Moggi's approach [6, 7] of incorporating computational effects in higher-order functional programming languages via the use of monads posed the basis for a unified category theoretic semantics for computational effects such as *nondeterminism*, *probabilistic nondeterminism*, *side-effects*, *exceptions*, etc. His semantics makes a careful systematic distinction between computational effects and values. A computation may be *pure*, in which case it terminates and returns a value, or *effectful*, in which case it performs a side-effect encapsulated into a monad structure.

The first programming language integrating this idea was Haskell, offering pre-built definitions in its core library; but with the influence of functional programming into other paradigms, formulations of monads (in spirit if not in name) can be found also in popular languages such as Python, Scala, and F#.

Moggi's work was followed up by the program of Plotkin and Power [9, 8, 10] on understanding computational effects as arising from operations and equations (see also the survey of Hyland and Power [3]). The original insight by Plotkin and Power was that many computational effects are naturally described by algebraic theories, and that computations should be described as operations on an algebra. The most profound result in [9] is a generalisation of the correspondence between finitary monads and Lawvere theories from **Set** to a category with finite products $\mathscr{C}$ and a strong monad T on $\mathscr{C}$. This result characterises *generic algebraic effects*, that is, computational effects described by operations and equations and interpreted on categories possibly richer than **Set**. A key motivating example from [8] shows how one is allowed to consider computational effects resulting from the solutions of recursive operations by interpreting them in the category of ω-cpo's.

With the emergence of probabilistic programming, which is characterised by the use of probabilistic nondeterminism as build-in computational effect, more emphasis has been put on *quantitative reasoning*. One thinks in terms of "how close are two programs?" rather than "are they completely indistinguishable?". This concept is captured by a *metric* and was first advocated in [2]. To address this need, Mardare et al. [4] proposed a version of equational reasoning, which they call *quantitative equational logic*, that captures such metric reasoning principles and, crucially, characterises algebraic effects on the category of metric spaces.

Abstract. In this talk I will review the basic definitions, constructions and key results presented in three works, [4, 5, 1], which constitute the first steps of the more ambitious the program of understanding the algebraic properties of computational effects on categories enriched over (extended) metric spaces. During the talk I will provide several motivating examples to offer a general pragmatic picture of how algebraic reasoning works and how it can be used. Finally, I will conclude by presenting a list of open problems and initial ideas on how to address them.

O. Dardha and J. Rot (Eds.): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics (EXPRESS/SOS 2020).
EPTCS 322, 2020, pp. 1–2, doi:10.4204/EPTCS.322.1

References

[1] Giorgio Bacci, Radu Mardare, Prakash Panangaden & Gordon D. Plotkin (2018): *An Algebraic Theory of Markov Processes*. In: *LICS*, ACM, pp. 679–688, doi:10.1145/3209108.3209177.

[2] Alessandro Giacalone, Chi-Chang Jou & Scott A. Smolka (1990): *Algebraic Reasoning for Probabilistic Concurrent Systems*. In: *Programming Concepts and Methods*, IFIP TC2, North-Holland, pp. 443–458.

[3] Martin Hyland & John Power (2007): *The Category Theoretic Understanding of Universal Algebra: Lawvere Theories and Monads*. *Electron. Notes Theor. Comput. Sci.* 172, pp. 437–458, doi:10.1016/j.entcs.2007.02.019.

[4] Radu Mardare, Prakash Panangaden & Gordon D. Plotkin (2016): *Quantitative Algebraic Reasoning*. In: *LICS*, ACM, pp. 700–709, doi:10.1145/2933575.2934518.

[5] Radu Mardare, Prakash Panangaden & Gordon D. Plotkin (2017): *On the axiomatizability of quantitative algebras*. In: *LICS*, IEEE Computer Society, pp. 1–12, doi:10.1109/LICS.2017.8005102.

[6] Eugenio Moggi (1988): *The Partial Lambda Calculus*. Ph.D. thesis, University of Edinburgh. College of Science and Engineering. School of Informatics.

[7] Eugenio Moggi (1991): *Notions of Computation and Monads*. *Inf. Comput.* 93(1), pp. 55–92, doi:10.1016/0890-5401(91)90052-4.

[8] Gordon D. Plotkin & John Power (2001): *Adequacy for Algebraic Effects*. In: *FoSSaCS*, *Lecture Notes in Computer Science* 2030, Springer, pp. 1–24, doi:10.1007/3-540-45315-6_1.

[9] Gordon D. Plotkin & John Power (2001): *Semantics for Algebraic Operations*. In: *MFPS*, *Electronic Notes in Theoretical Computer Science* 45, Elsevier, pp. 332–345, doi:10.1016/S1571-0661(04)80970-8.

[10] Gordon D. Plotkin & John Power (2002): *Notions of Computation Determine Monads*. In: *FoSSaCS*, *Lecture Notes in Computer Science* 2303, Springer, pp. 342–356, doi:10.1007/3-540-45931-6_24.

Divergence-Preserving Branching Bisimilarity

Bas Luttik
Eindhoven University of Technology
The Netherlands
s.p.luttik@tue.nl

This note considers the notion of divergence-preserving branching bisimilarity. It briefly surveys results pertaining to the notion that have been obtained in the past one-and-a-half decade, discusses its role in the study of expressiveness of process calculi, and concludes with some suggestions for future work.

1 Introduction

Branching bisimilarity was proposed by van Glabbeek and Weijland as an upgrade of (strong) bisimilarity that facilitates abstraction from internal activity [16]. It preserves the branching structure of processes more strictly than Milner's *observation equivalence* [21], which, according to van Glabbeek and Weijland, makes it, e.g., better suited for verification purposes. A case in point is the argument by Graf and Sifakis that there is no temporal logic with an *eventually* operator that is adequate for observation equivalence in the sense that two processes satisfy the same formulas if, and only if, they are observationally equivalent [17]. The crux is that observation equivalence insufficiently takes into account the intermediate states of an internal computation. Indeed, branching bisimilarity requires a stronger correspondence between the intermediate states of an internal computation.

Branching bisimilarity is also not compatible with a temporal logic that includes an eventually operator, because it abstracts to some extent from *divergence* (i.e., infinite internal computations). Thus, a further upgrade is necessary, removing the abstraction from divergence. De Nicola and Vaandrager show that *divergence-sensitive* branching bisimilarity coincides with the equivalence induced by satisfaction of formulas of the temporal logic CTL^*_{-X} [6]. (CTL^* [8] is an expressive state-based logic that includes both linear time and branching time modalities; CTL^*_{-X} refers to the variant of CTL^* obtained by omitting the next-state modality, which is incompatible with abstraction from internal activity.)

Divergence-sensitive branching bisimilarity still has one drawback when it comes to verification: it identifies deadlock and livelock and, as an immediate consequence, is not compatible with parallel composition. It turns out that the notion of *divergence-preserving* branching bisimilarity[1], which is the topic of this note, has all the right properties: it is the coarsest equivalence that is compatible with parallel composition, preserves CTL^*_{-X} formulas, and distinguishes deadlock and livelock [15]. Moreover, on finite processes divergence-preserving branching bisimilarity can be decided efficiently [18].

In [16], a coloured-trace characterisation of divergence-preserving branching bisimilarity is provided. In [13], relational and modal characterisations of the notion are given. For some time it was simply assumed that these three characterisations of the notion coincide, but this was only proved in [14]. To establish that the relational characterisation coincides with the coloured-trace and modal characterisations, it needs to be proved that the relational characterisation yields an equivalence relation that satisfies

[1] For stylistic reasons we prefer the term "divergence-preserving branching bisimilarity" over "branching bisimilarity with explicit divergence", which is used in earlier articles on the topic.

O. Dardha and J. Rot (Eds.): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics (EXPRESS/SOS 2020).
EPTCS 322, 2020, pp. 3–11, doi:10.4204/EPTCS.322.2

the so-called stuttering property, and this is surprisingly involved. A similar phenomenon is observed in the proof that a rooted version of divergence-preserving branching bisimilarity is compatible with the recursion construct $\mu X.\_$ [12]. Due to the divergence condition, Milner's ingenious argument in [23] that strong bisimilarity is compatible with recursion required several novel twists.

In this note we shall give a survey of results pertaining to divergence-preserving branching bisimilarity that were obtained in the past one-and-a-half decade. in Section 2 we shall present and discuss a relational characterisation of the notion. In Section 3 we comment on modal characterisations of the notion, and discuss the relationship with the temporal logic CTL^*_{-X}. In Section 4 we briefly discuss to what extent the notion is compatible with familiar process algebraic operators. In Section 5, we explain how it plays a role in expressiveness results. In Section 6 we arrive at some conclusions and mention some ideas for future work.

2 Relational characterisation

We presuppose a set $\mathscr{A}_\tau$ of *actions* including a special element τ, and we presuppose a *labelled transition system* $(S, \longrightarrow)$ with labels from $\mathscr{A}_\tau$, i.e., S is a set of *states* and $\longrightarrow \subseteq S \times \mathscr{A}_\tau \times S$ is a *transition relation* on S. Let $s, s' \in S$ and $\alpha \in \mathscr{A}_\tau$; we write $s \xrightarrow{\alpha} s'$ for $(s, \alpha, s') \in \longrightarrow$ and we abbreviate the statement '$s \xrightarrow{\alpha} s'$ or ($\alpha = \tau$ and $s = s'$)' by $s \xrightarrow{(\alpha)} s'$. We denote by $\rightarrow^+$ the transitive closure of the binary relation $\xrightarrow{\tau}$, and by $\twoheadrightarrow$ its reflexive-transitive closure. A *process* is given by a state s in a labelled transition system, and encompasses all the states and transitions reachable from s.

Definition 1. A symmetric binary relation $\mathscr{R}$ on S is a *branching bisimulation* if it satisfies the following condition for all $s, t \in S$ and $\alpha \in \mathscr{A}_\tau$:

(T) if $s \mathscr{R} t$ and $s \xrightarrow{\alpha} s'$ for some state s', then there exist states t' and t'' such that $t \twoheadrightarrow t'' \xrightarrow{(\alpha)} t'$, $s \mathscr{R} t''$ and $s' \mathscr{R} t'$.

We say that a branching bisimulation $\mathscr{R}$ *preserves (internal) divergence* if it satisfies the following condition for all $s, t \in S$:

(D) if $s \mathscr{R} t$ and there is an infinite sequence of states $(s_k)_{k\in\omega}$ such that $s = s_0$, $s_k \xrightarrow{\tau} s_{k+1}$ and $s_k \mathscr{R} t$ for all $k \in \omega$, then there is a state t' such that $t \longrightarrow^+ t'$, and $s_k \mathscr{R} t'$ for some $k \in \omega$.

States s and t are *divergence-preserving branching bisimilar* (notation: $s \leftrightarrows\!\!\!\!\!\!\leftrightarrow^{\Delta}_b t$) if there is a divergence-preserving branching bisimulation $\mathscr{R}$ such that $s \mathscr{R} t$.

The divergence condition (D) in the definition above is slightly weaker than the divergence condition used in the relation characterisation of divergence-preserving branching bisimilarity presented in [13], which actually requires that t admits an infinite sequence of τ-transitions and every state on this sequence is related to some state on the infinite sequence of τ-transitions from s. Nevertheless, as is established in [14], the notion of divergence-preserving branching bisimilarity defined here is equivalent to the one defined in [13]. In [14] it is also proved that $\leftrightarrows\!\!\!\!\!\!\leftrightarrow^{\Delta}_b$ is an equivalence, that the relation $\leftrightarrows\!\!\!\!\!\!\leftrightarrow^{\Delta}_b$ is itself a divergence-preserving branching bisimulation, and that it satisfies the so-called *stuttering property*: if $t_0 \xrightarrow{\tau} \cdots t_n$, $s \leftrightarrows\!\!\!\!\!\!\leftrightarrow^{\Delta}_b t_0$ and $s \leftrightarrows\!\!\!\!\!\!\leftrightarrow^{\Delta}_b t_n$, then $s \leftrightarrows\!\!\!\!\!\!\leftrightarrow^{\Delta}_b t_i$ for all $0 \leq i \leq n$.

Let us say that a state s is *divergent* if there exists an infinite sequence of states $(s_k)_{k\in\omega}$ such that $s = s_0$ and $s_k \xrightarrow{\tau} s_{k+1}$ for all $k \in \omega$. It is a straightforward consequence of the definition that divergence-preserving branching bisimilarity relates divergent states to divergent states only, i.e., that we have the following proposition.

Proposition 2. *If $s \leftrightarrows\!\!\!\!\!\!\leftrightarrow^{\Delta}_b t$, then s is divergent only if t is divergent.*

Proof. Suppose that $s \leftrightarrows_b^\Delta t$ and s is divergent. Then there exists an infinite sequence of states $(s_k)_{k\in\omega}$ such that $s = s_0$ and $s_k \xrightarrow{\tau} s_{k+1}$ for all $k \in \omega$. We inductively construct an infinite sequence of states $(t_\ell)_{\ell\in\omega}$ such that $t = t_0$, $t_\ell \xrightarrow{\tau} t_{\ell+1}$, together with a mapping $\sigma : \omega \to \omega$ such that $s_{\sigma(\ell)} \leftrightarrows_b^\Delta t_\ell$ for all $\ell \in \omega$;

- We define $t_0 = t$ and $\sigma(0) = 0$; note that $s_{\sigma(0)} = s \leftrightarrows_b^\Delta t = t_0$.
- Suppose that the sequence $(t_\ell)_{\ell\in\omega}$ and the mapping σ have been defined up to ℓ. Then, in particular, $s_{\sigma(\ell)} \leftrightarrows_b^\Delta t_\ell$. We distinguish two cases:

  If $s_{\sigma(\ell)+k} \leftrightarrows_b^\Delta t_\ell$ for all $k \in \omega$, then by (D) there exists $t_{\ell+1}$ such that $t_\ell \xrightarrow{\tau} t_{\ell+1}$ and $s_{\sigma(\ell)+k} \leftrightarrows_b^\Delta t_{\ell+1}$ for some $k \in \omega$; we can then define $\sigma(\ell+1) = k$.

  Otherwise, there exists some $k \in \omega$ such that $s_{\sigma(\ell)+k} \leftrightarrows_b^\Delta t_\ell$ and $s_{\sigma(\ell)+k+1} \not\leftrightarrows_b^\Delta t_\ell$. Since $s_{\sigma(\ell)+k} \xrightarrow{\tau} s_{\sigma(\ell)+k+1}$ it follows by (T) that there exist t_ℓ'' and $t_{\ell+1}$ such that $t_\ell \twoheadrightarrow t_\ell'' \xrightarrow{(\tau)} t_{\ell+1}$, $s_{\sigma(\ell)+k} \leftrightarrows_b^\Delta t_\ell''$ and $s_{\sigma(\ell)+k+1} \leftrightarrows_b^\Delta t_{\ell+1}$. Clearly, we have that $t_\ell \neq t_{\ell+1}$, so $t_\ell \longrightarrow^+ t_{\ell+1}$ and we can define $\sigma(\ell+1) = \sigma(\ell)+k+1$.

From the existence of an infinite sequence of states $(t_\ell)_{\ell\in\omega}$ such that $t = t_0$ and $t_\ell \xrightarrow{\tau} t_{\ell+1}$ it follows that t is divergent, as was to be shown. □

As the following example illustrates, however, a symmetric binary relation on S relating states that satisfies (T) of Definition 1 and relates divergent states to divergent states only is *not necessarily* included in a divergence-preserving branching bisimulation relation. In other words, a symmetric binary relation on S that satisfies (T) and only relates divergent states to divergent states may relate states that are not divergence-preserving branching bisimilar.

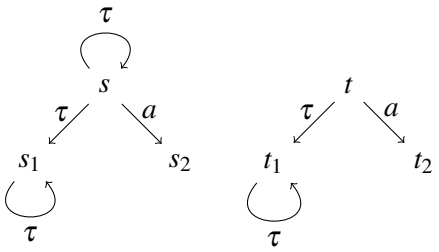


Figure 1: An example transition system illustrating that (D) cannot be replaced by the requirement that $\mathscr{R}$ relates divergent states to divergent states.

Example 3. Consider the transition system depicted in Figure 1. The symmetric closure of the relation $\mathscr{R} = \{(s,t),(s_1,t_2),(s_2,t_2)\}$ satisfies (T) and it relates divergent states to divergent states only. It does not, however, satisfy (D), for $s \mathscr{R} t$ and defining $s_k = s$ for all $k \in \omega$ we get an infinite sequence of states $(s_k)_{k\in\omega}$ such that $s_k \xrightarrow{\tau} s_{k+1}$ and $s_k \mathscr{R} t$ for all $k \in \omega$, while there does not exist a t' such that $t \longrightarrow^+ t'$ and $s_k \mathscr{R} t'$ for some $k \in \omega$. Note that s admits a complete path at which a is continuously (weakly) enabled, whereas t does not admit such a complete path.

3 Modal characterisations

As shown in [13], to get an (action-based) modal logic that is adequate for branching bisimilarity one could take an adaptation of standard Hennessy-Milner logic replacing, for all actions $\alpha \in \mathscr{A}_\tau$ in the

usual unary may and must modalities $\langle a \rangle$ and $[a]$ by a binary *just-before* modality a. A state s satisfies the formula $\varphi\, a\, \psi$ if, and only if, there exist states s'' and s' such that $s \longrightarrow\!\!\!\!\!\rightarrow s'' \xrightarrow{(\alpha)} s'$, φ holds in s'' and ψ holds in s'. To get an adequate logic for divergence-preserving branching bisimilarity, it suffices to add a unary *divergence modality* Δ such that s satisfies $\Delta\varphi$ if, and only if, there exists an infinite sequence of states $(s_k)_{k\in\omega}$ such that $s \longrightarrow\!\!\!\!\!\rightarrow s_0$, $s_k \xrightarrow{\tau} s_{k+1}$ and φ holds in s_k for all $k \in \omega$.

Let Φ be the class of formulas generated by the following grammar:

$$\varphi ::= \neg\varphi \mid \bigwedge \Phi' \mid \varphi\, \alpha\, \varphi \mid \Delta\varphi \qquad (\alpha \in \mathscr{A}_\tau,\ \varphi \in \Phi,\ \Phi' \subseteq \Phi)\ .$$

We then have that states s and t are divergence-proving branching bisimilar if, and only if, s and t satisfy exactly the same formula in Φ [14]. We may restrict the cardinality of Φ' in conjunctions to the cardinality of the set of states S.

Example 4. Consider again the transition system depicted in Figure 1. States s and t are not divergence-preserving branching bisimilar. The formula $\Delta(\top\, a\, \top)$ (in which $\top$ abbreviates $\bigwedge\emptyset$) expresses the existence of a divergence on which the action a is continuously enabled. It is satisfied by state s, but not by t.

There is also an intuitive correspondence between branching bisimilarity and the state-based temporal logic CTL^*_{-X} (CTL^* without the next-state modality) [7]. The standard semantics of CTL^*_{-X} is, however, with respect to Kripke structures, in which states rather than transitions have labels and the transition relation is assumed to be total. To formalise the correspondence, De Nicola and Vaandrager devised a framework of translations between labelled transition systems and Kripke structures [6]. The main idea of the translation from labelled transition systems to Kripke structures is that

1. every transition $s \xrightarrow{a} t$ ($a \neq \tau$) is replaced by two transitions $s \longrightarrow t_a$ and $t_a \longrightarrow t$, where t_a is a fresh state that is labelled with $\{a\}$;
2. every transition $s \xrightarrow{\tau} t$ gives rise to a transition $s \longrightarrow t$; and
3. for every state s without outgoing transitions (i.e., every deadlock state of the labelled transition system) a transition $s \longrightarrow s$ is added to satisfy the totality requirement of Kripke structures.

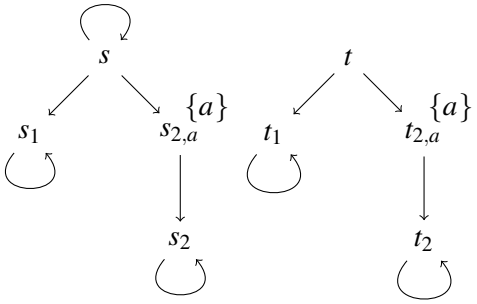


Figure 2: Result of apply De Nicola and Vaandrager's translation to the labelled transition system in Figure 1.

Example 5. If we apply the translation sketched above to the labelled transition system depicted in Figure 1, then we get the Kripke structure depicted in Figure 2. Note that by clause 3 of the translation state s_2 gets a transition to itself, whereas it is a deadlock state in the orginal transition system. Clearly, there is no CTL_{-X} formula that distinguishes, e.g., between s_1 and s_2, although in the labelled transition system depicted in Figure 1 these states are not divergence-preserving branching bisimilar.

De Nicola and Vaandrager propose a notion of *divergence-sensitive branching bisimilarity* on finite LTSs and establish that two states in an LTS are divergence-sensitive branching bisimilar if, and only if, in the Kripke resulting from the translation sketched above they satisfy the same CTL^*_{-X} formulas. Divergence-sensitive branching bisimilarity coincides with divergence-preserving branching bisimilarity on deadlock-free LTSs. In fact, the only difference between divergence-sensitive branching bisimilarity and divergence-preserving branching bisimilarity is that the latter distinguishes between deadlock and livelock states, whereas the former does not.

To preserve the distinction between deadlock and livelock, a modified translation is proposed in [15], obtained from the translation sketched above by replacing clause 3 by

3′. add a fresh state d labelled with $\{\delta\}$, and for every state s without outgoing transitions a transition $s \longrightarrow d$.

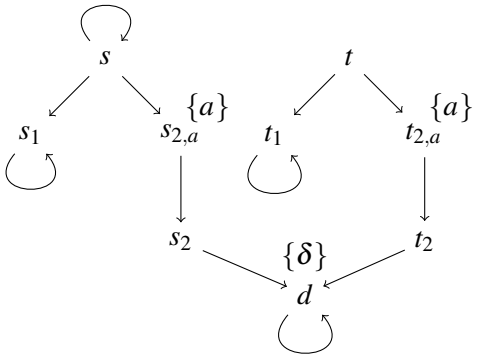


Figure 3: Result of apply the deadlock preserving translation to the labelled transition system in Figure 1.

Example 6. Applying the modified translation on the labelled transition in Figure 1, we get the Kripke structure in Figure 3. Note that s_1 does not satisfy the CTL^*_{-X} formula $\mathsf{EF}\,\delta$, while s_2 does.

Two states in a labelled transition system are divergence-preserving branching bisimilar if they satisfy the same CTL^*_{-X} formulas in the Kripke structure that results from the modified transition [15].

4 Congruence

An important reason to prefer divergence-preserving branching bisimilarity over divergence-sensitive branching bisimilarity is that the former is compatible with parallel composition, whereas the latter is not.

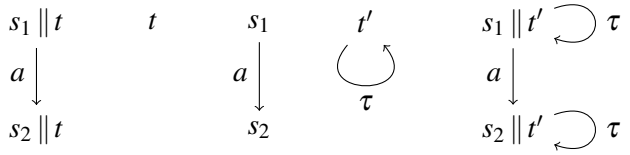


Figure 4: Divergence-sensitive branching bisimilarity is not compatible with parallel composition.

Example 7. Consider the transition system in Figure 4. States $s_1 \parallel t$ and $s_2 \parallel t$ represent the parallel compositions of states s_1 and t, and of states s_2 and t, respectively. Similarly, states $s_1 \parallel t'$ and $s_2 \parallel t'$ represent the parallel compositions of states s_1 and t', and of states s_2 and t', respectively. Recall that divergence-sensitive branching bisimilarity does not distinguish deadlock (state t) and livelock (state t'), so we have that t and t' are divergence-sensitive branching bisimilar. States $s_1 \parallel t$ and $s_1 \parallel t'$ are, however, not divergence-sensitive branching bisimilar. Note that $s_1 \parallel t'$ has a complete path on which a is continuously enabled, whereas $s_1 \parallel t$ does not have such a complete path, and so these two states do not satisfy the same CTL^*_{-X} formulas.

Divergence-preserving branching bisimilarity is the coarsest equivalence included in divergence-sensitive branching bisimilarity that is compatible with parallel composition [15]. Hence, it is also the coarsest congruence for parallel composition relating only processes that satisfy the same CTL^*_{-X} formulas.

It is well-known that branching bisimilarity is not compatible with non-deterministic choice, and that the coarsest behavioural equivalence that is included in branching bisimilarity and that is compatible with non-deterministic choice, is obtained by adding a so-called *root condition*. The same holds for divergence-preserving branching bisimilarity.

Definition 8. Let $\mathscr{R}$ be a divergence-preserving branching bisimulation. We say that $\mathscr{R}$ satisfies the *root condition* for s and t if, whenever

(R1) if $s \xrightarrow{\alpha} s'$ for some state s', then there exists a state t' such that $t \xrightarrow{\alpha} t'$ and $s' \mathscr{R} t'$.

(R2) if $t \xrightarrow{\alpha} t'$ for some state t', then there exists a state s' such that $s \xrightarrow{\alpha} s'$ and $s' \mathscr{R} t'$.

States s and t are *rooted divergence-preserving branching bisimilar* if there is a divergence-preserving branching bisimulation relation $\mathscr{R}$ satisfying the root condition for s and t such that that $s \mathscr{R} t$.

In [11], formats for transition system specifications are presented that guarantee that divergence-preserving branching bisimilarity and its rooted variant are compatible with the operators defined by the transition system specification. These formats relax the requirements of the branching bisimulation and rooted branching bisimulation formats of [10]. The relaxation of the formats is meaningful: the process-algebraic operations for *priority* [1] and *sequencing* [5, 4, 3], with which (rooted) branching bisimilarity is *not* compatible, are in the rooted divergence-preserving branching bisimulation format. So, in contrast to its divergence-insensitive variant, rooted divergence-preserving branching bisimilarity is compatible with priority and sequencing.

The structural operational rule for the recursion operator $\mu X.\_$, which was considered in the context of observation equivalence by Milner [22] and in the context of divergence-sensitive variants of observation equivalence by Lohrey, D'Argenio and Hermanns [19], is not in the format for rooted divergence-preserving branching bisimilarity. Nevertheless, rooted divergence preserving branching bisimilarity is compatible also with this operator [12]. The proof of this fact requires an adaption of the up-to technique used by Milner in his argument that (strong) bisimilarity is compatible with recursion [23].

5 Expressiveness of process calculi

Phillips showed that abstraction from divergence can be exploited to prove that every recursively enumerable transition system is branching bisimilar to a boundedly branching computable transition system [25][2]. In contrast, there exist recursively enumerable transition systems that are not divergence-

[2] Phillips actually claimed the correspondence modulo observation equivalence, but it is easy to see that his proof also works modulo branching bisimilarity.

preserving branching bisimilar to a computable transition system (cf., e.g., Example 3.6 in [2]). Hence, in a theory that aims to integrate computability and concurrency, divergence preservation is important.

In [2], interactivity is added to Turing machines by associating an action with every computation step. This so-called *reactive* Turing machine has a transition system semantics and can be studied from a concurrency-theoretic perspective. A transition system is called *executable* if it is behaviourally equivalent to the transition system associated with a reactive Turing machine. The notion of executability provides a way to characterise the absolute expressiveness of a process calculus. If every transition system that can be specified in the calculus is executable, then the calculus is said to be executable. Conversely, if every executable transition system can be specified in the calculus, then the calculus is said to be behaviourally complete.

A calculus with constants for deadlock and successful termination, unary action prefixes, binary operations for non-deterministic choice, sequencing and ACP-style parallel composition, iteration and nesting is both executable and behaviourally complete up to divergence-preserving branching bisimilarity [3]. The π-calculus is also behaviourally complete up to divergence-preserving branching bisimilarity. Since it allows the specification of transition systems with unbounded branching, it is, however, not executable up to divergence-preserving branching bisimilarity; it is nominally orbit-finitely executable up to the divergence-insensitive variant of branching bisimilarity [20].

The aforementioned results illustrate the role of divergence in the consideration of the absolute expressiveness of process calculi. Preservation of divergence is also widely accepted as an important criterion when comparing the relative expressiveness of process calculi [24].

6 Conclusions

We have discussed ther relational and modal characterisations of divergence-preserving branching bisimilarity, commented on its compatibility with respect to process algebraic operations and on its role in the study of the absolute expressiveness. We conclude by briefly mentioning some directions for future work.

Sound and complete axiomatisations for the divergence-sensitive spectrum of observation congruence for basic CCS with recursion are provided in [19]. The congruence result in [12] can serve as a stepping stone for providing similar sound and complete axiomatisations for divergence-preserving branching bisimilarity. Then, it would also be interesting to consider the axiomatisation of divergence-preserving branching bisimilarity for full CCS with recursion, although that would first require a non-trivial extension of the congruence result.

Ad hoc up-to techniques for divergence-preserving branching bisimilarity have already been used, e.g., in the congruence proof in [12] and in proof that the π-calculus is behaviourally complete [20]. Recently, several more generic up-to techniques for branching bisimilarity were proved sound [9]. An interesting direction for future work would be to consider extending those up-to techniques for divergence-preserving branching bisimilarity too.

References

[1] J. C. M. Baeten, J. A. Bergstra & J. W. Klop (1986): *Syntax and defining equations for an interrupt mechanism in process algebra*. Fundamenta Informaticae 9(2), pp. 127–167.

[2] J. C. M. Baeten, B. Luttik & P. van Tilburg (2013): *Reactive Turing machines*. Inf. Comput. 231, pp. 143–166, doi:10.1016/j.ic.2013.08.010.

[3] J. C. M. Baeten, B. Luttik & F. Yang (2017): *Sequential Composition in the Presence of Intermediate Termination (Extended Abstract)*. In K. Peters & S. Tini, editors: Proceedings Combined 24th International Workshop on *Expressiveness in Concurrency* and 14th Workshop on *Structural Operational Semantics*, EXPRESS/SOS 2017, Berlin, Germany, 4th September 2017, *EPTCS* 255, pp. 1–17, doi:10.4204/EPTCS.255.1.

[4] A. Belder, B. Luttik & J. C. M. Baeten (2019): *Sequencing and Intermediate Acceptance: Axiomatisation and Decidability of Bisimilarity*. In M. Roggenbach & A. Sokolova, editors: *8th Conference on Algebra and Coalgebra in Computer Science, CALCO 2019, June 3-6, 2019, London, United Kingdom*, *LIPIcs* 139, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 11:1–11:22, doi:10.4230/LIPIcs.CALCO.2019.11.

[5] B. Bloom (1994): *When is Partial Trace Equivalence Adequate?* *Formal Asp. Comput.* 6(3), pp. 317–338, doi:10.1007/BF01215409.

[6] R. De Nicola & F. W. Vaandrager (1995): *Three Logics for Branching Bisimulation*. *Journal of the ACM* 42(2), pp. 458–487, doi:10.1145/201019.201032.

[7] E. A. Emerson & E. M. Clarke (1982): *Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons*. *Science of Computer Programming* 2(3), pp. 241–266, doi:10.1016/0167-6423(83)90017-5.

[8] E. A. Emerson & J.Y. Halpern (1986): *'Sometimes' and 'Not Never' revisited: on branching time versus linear time temporal logic*. *Journal of the ACM* 33(1), pp. 151–178, doi:10.1145/4904.4999.

[9] R. Erkens, J. Rot & B. Luttik (2020): *Up-to Techniques for Branching Bisimilarity*. In A. Chatzigeorgiou, R. Dondi, H. Herodotou, C. A. Kapoutsis, Y. Manolopoulos, G. A. Papadopoulos & F. Sikora, editors: *SOFSEM 2020: Theory and Practice of Computer Science - 46th International Conference on Current Trends in Theory and Practice of Informatics, SOFSEM 2020, Limassol, Cyprus, January 20-24, 2020, Proceedings*, *Lecture Notes in Computer Science* 12011, Springer, pp. 285–297, doi:10.1007/978-3-030-38919-2_24.

[10] W. J. Fokkink, R. J. van Glabbeek & P. de Wind (2012): *Divide and congruence: From decomposition of modal formulas to preservation of branching and η-bisimilarity*. *Inf. Comput.* 214, pp. 59–85, doi:10.1016/j.ic.2011.10.011.

[11] W. J. Fokkink, Glabbeek R. J. van & B. Luttik (2019): *Divide and congruence III: From decomposition of modal formulas to preservation of stability and divergence*. *Inf. Comput.* 268, doi:10.1016/j.ic.2019.104435.

[12] R. J. van Glabbeek, B. Luttik & L. Spaninks (2020): *Rooted Divergence-Preserving Branching Bisimilarity is a Congruence*. *CoRR* abs/1801.01180. Available at http://arxiv.org/abs/1801.01180. Submitted.

[13] R. J. van Glabbeek (1993): *The Linear Time – Branching Time Spectrum II; The semantics of sequential systems with silent moves (extended abstract)*. In E. Best, editor: Proceedings 4^{th} International Conference on *Concurrency Theory*, CONCUR'93, Hildesheim, Germany, August 1993, LNCS 715, Springer, pp. 66–81, doi:10.1007/3-540-57208-2_6.

[14] R. J. van Glabbeek, B. Luttik & N. Trčka (2009): *Branching Bisimilarity with Explicit Divergence*. *Fundamenta Informaticae* 93(4), pp. 371–392, doi:10.3233/FI-2009-109.

[15] R. J. van Glabbeek, B. Luttik & N. Trčka (2009): *Computation Tree Logic with Deadlock Detection*. *Logical Methods in Computer Science* 5(4), doi:10.2168/LMCS-5(4:5)2009.

[16] R. J. van Glabbeek & W. P. Weijland (1996): *Branching time and abstraction in bisimulation semantics*. *Journal of the ACM* 43(3), pp. 555–600, doi:10.1145/233551.233556.

[17] S. Graf & J. Sifakis (1987): *Readiness Semantics for Regular Processes with Silent Actions*. In T. Ottmann, editor: *Automata, Languages and Programming, 14th International Colloquium, ICALP87, Karlsruhe, Germany, July 13-17, 1987, Proceedings*, *Lecture Notes in Computer Science* 267, Springer, pp. 115–125, doi:10.1007/3-540-18088-5_10.

[18] J. F. Groote, D. N. Jansen, J. J. A. Keiren & A. J. Wijs (2017): *An O(m log n) Algorithm for Computing Stuttering Equivalence and Branching Bisimulation*. *ACM Trans. Comput. Logic* 18(2), doi:10.1145/3060140.

[19] M. Lohrey, P. R. D'Argenio & H. Hermanns (2005): *Axiomatising divergence*. *Inf. Comput.* 203(2), pp. 115–144, doi:10.1016/j.ic.2005.05.007.

[20] B. Luttik & F. Yang (2020): *The π-Calculus is Behaviourally Complete and Orbit-Finitely Executable*. *CoRR* abs/1410.4512v8. Available at http://arxiv.org/abs/1410.4512.

[21] R. Milner (1980): *A Calculus of Communicating Systems*. *Lecture Notes in Computer Science* 92, Springer, doi:10.1007/3-540-10235-3.

[22] R. Milner (1989): *A Complete Axiomatisation for Observational Congruence of Finite-State Behaviors*. *Inf. Comput.* 81(2), pp. 227–247, doi:10.1016/0890-5401(89)90070-9.

[23] R. Milner (1990): *Operational and Algebraic Semantics of Concurrent Processes*. In Jan van Leeuwen, editor: *Handbook of Theoretical Computer Science (Vol. B)*, MIT Press, Cambridge, MA, USA, pp. 1201–1242. Available at http://dl.acm.org/citation.cfm?id=114891.114910.

[24] K. Peters (2019): *Comparing Process Calculi Using Encodings*. In J. A. Pérez & J. Rot, editors: *Proceedings Combined 26th International Workshop on Expressiveness in Concurrency and 16th Workshop on Structural Operational Semantics, EXPRESS/SOS 2019, Amsterdam, The Netherlands, 26th August 2019*, *EPTCS* 300, pp. 19–38, doi:10.4204/EPTCS.300.2.

[25] I. Phillips (1993): *A Note on Expressiveness of Process Algebra*. In G. L. Burn, S. J. Gay & M. Ryan, editors: Proceedings of the First Imperial College Department of Computing Workshop on *Theory and Formal Methods*, Isle of Thorns Conference Centre, Chelwood Gate, Sussex, UK, 29-31 March 1993, Workshops in Computing, Springer, pp. 260–264.

Multiparty Session Programming with Global Protocol Combinators

Rumyana Neykova
Brunel University London

Multiparty Session Types (MPST) is a typing discipline for communication protocols. It ensures the absence of communication errors and deadlocks for well-typed communicating processes. The state-of-the-art implementations of the MPST theory rely on (1) *runtime linearity checks* to ensure correct usage of communication channels and (2) external domain-specific languages for specifying and verifying multiparty protocols.

In this talk I will present a library for programming with *global combinators* – a set of functions for writing and verifying multiparty protocols in OCaml, that overcomes the above limitations. Local behaviours for *all* processes in a protocol are inferred *at once* from a global combinator. We formalise global combinators and prove a sound realisability of global combinators – a well-typed global combinator derives a set of local types, by which typed endpoint programs can ensure type and communication safety. Our approach enables fully-static verification and implementation of the whole protocol, from the protocol specification to the process implementations, to happen in the same language.

I will show the implementation in OCaml and will discuss its expressive power and performance. Our work has several distinctive features. It is the first fully-static MPST implementation; it realises multiparty communication over binary channels; it is lightweight – verification of protocols is reduced to type checking; and expressive – we have implemented a plethora of protocols (e.g OAuth, DNS, SMTP). This talk is based on a joint work with Keigo Imai, Nobuko Yoshida, and Shoji Yuen.

O. Dardha and J. Rot (Eds.): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics (EXPRESS/SOS 2020).
EPTCS 322, 2020, pp. 12–12, doi:10.4204/EPTCS.322.3

Can determinism and compositionality coexist in RML?

Davide Ancona Viviana Mascardi
DIBRIS, University of Genova, Italy
{Davide.Ancona,Viviana.Mascardi}@unige.it

Angelo Ferrando
University of Manchester, UK
angelo.ferrando@manchester.ac.uk

Runtime verification (RV) consists in dynamically verifying that the event traces generated by single runs of a system under scrutiny (SUS) are compliant with the formal specification of its expected properties. RML (Runtime Monitoring Language) is a simple but expressive Domain Specific Language for RV; its semantics is based on a trace calculus formalized by a deterministic rewriting system which drives the implementation of the interpreter of the monitors generated by the RML compiler from the specifications. While determinism of the trace calculus ensures better performances of the generated monitors, it makes the semantics of its operators less intuitive. In this paper we move a first step towards a compositional semantics of the RML trace calculus, by interpreting its basic operators as operations on sets of instantiated event traces and by proving that such an interpretation is equivalent to the operational semantics of the calculus.

1 Introduction

RV [35, 27, 13] consists in dynamically verifying that the event traces generated by single runs of a SUS are compliant with the formal specification of its expected properties.

The RV process needs as inputs the SUS and the specification of the properties to be verified, usually defined with either a domain specific (DSL) or a programming language, to denote the set of valid event traces; RV is performed by monitors, automatically generated from the specification, which consume the observed events of the SUS, emit verdicts and, in case they work online while the SUS is executing, feedback useful for error recovery.

RV is complimentary to other verification methods: analogously to formal verification, it uses a specification formalism, but, as opposite to it, scales well to real systems and complex properties and it is not exhaustive as happens in software testing; however, it also exhibits several distinguishing features: it is quite useful to check control-oriented properties [2], and offers opportunities for fault protection when the monitor runs online. Many RV approaches adopt a DSL language to specifiy properties to favor portability and reuse of specifications and interoperability of the generated monitors and to provide stronger correctness guarantees: monitors automatically generated from a higher level DSL are more reliable than ad hoc code implemented in a ordinary programming language to perform RV.

RML[1] [28] is a simple but expressive DSL for RV which can be used in practice for RV of complex non Context-Free properties, as FIFO properties, which can be verified by the generated monitors in time linear in the size of the inspected trace; the language design and implementation is based on previous work on trace expressions and global types [7, 17, 4, 9], which have been adopted for RV in several contexts. Its semantics is based on a trace calculus formalized by a rewriting system which drives the implementation of the interpreter of the monitors generated by the RML compiler from the specifications; to allow better performances, the rewriting system is fully deterministic [11] by adopting a left-preferential evaluation strategy for binary operators and, thus, no monitor backtracking is needed and exponential

[1] https://rmlatdibris.github.io

O. Dardha and J. Rot (Eds.): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics (EXPRESS/SOS 2020).
EPTCS 322, 2020, pp. 13–32, doi:10.4204/EPTCS.322.4

explosion of the space allocated for the states of the monitor is avoided. A similar strategy is followed by mainstream programming languages in predefined libraries for regular expressions for efficient incremental matching of input sequences, to avoid the issue of Regular expression Denial of Service (ReDoS) [24]: for instance, given the regular expression `a?(ab)?` (optionally `a` concatenated with optionally `ab`) and the input sequence `ab`, the Java method `lookingAt()` of class `java.util.regex.Matcher` matches `a` instead of the entire input sequence `ab` because the evaluation of concatenation is deterministically left-preferential.

As explained more in details in Section 5, with respect to other existing RV formalisms, RML has been designed as an extension of regular expressions and deterministic context-free grammars, which are widely used in RV because they are well-understood among software developers as opposite to other more sophisticated approaches, as temporal logics. As shown in previous papers [7, 17, 4, 9], the calculus at the basis of RML allows users to define and efficiently check complex parameterized properties and it has been proved to be more expressive than LTL [8].

Unfortunately, while determinism ensures better performances, it makes the compositional semantics of its operators less intuitive; for instance, the example above concerning the regular expression `a?(ab)?` with deterministic left-preferential concatenation applies also to RML, which is more expressive than regular expressions: the compositional semantics of concatenation does not correspond to standard language concatenation, because `a?` and `(ab)?` denote the formal languages $\{\lambda, a\}$ and $\{\lambda, ab\}$, respectively, where λ denotes the empty string, while, if concatenation is deterministically left-preferential, then the semantics of `a?(ab)?` is $\{\lambda, a, aab\}$ which does not coincide with the language $\{\lambda, a, ab, aab\}$ obtained by concatenating $\{\lambda, a\}$ with $\{\lambda, ab\}$. In Section 4 we show that the semantics of left-preferential concatenation can still be given compositionally, although the corresponding operator is more complicate than standard language concatenation. Similar results follow for the other binary operators of RML (union, intersection and shuffle); in particular, the compositional semantics of left-preferential shuffle is more challenging. Furthermore, the fact that RML supports parametricity makes the compositional semantics more complex, since traces must be coupled with the corresponding substitutions generated by event matching. To this aim, as a first step towards a compositional semantics of the RML trace calculus, we provide an interpretation of the basic operators of the RML trace calculus as operations on sets of instantiated event traces, that is, pairs of trace of events and substitutions computed to bind the variables occurring in the event type patterns used in the specifications and to associate them with the data values carried by the matched events. Furthermore we prove that such an interpretation is equivalent to the original operational semantics of the calculus based on the deterministic rewriting system.

The paper is structured as follows: Section 2 introduces the basic definitions which are used in the subsequent technical sections, Section 3 formalizes the RML trace calculus and its operational semantics, while Section 4 introduces the semantics based on sets of instantiated event traces and formally proves its equivalence with the operational semantics; finally, Section 5 is devoted to the related work and Section 6 draws conclusions and directions for further work. For space limitations, some proof details can be found in the extended version [10] of this paper.

2 Technical background

This section introduces some basic definitions and propositions used in the next technical sections.

Partial functions: Let $f : D \to C$ be a partial function; then $dom(f) \subseteq D$ denotes the set of elements $d \in D$ s.t. $f(d)$ is defined (hence, $f(d) \in C$).

A partial function over natural numbers $f : \mathbb{N} \to N$, with $N \subseteq \mathbb{N}$, is *strictly increasing* iff for all $n_1, n_2 \in dom(f)$, $n_1 < n_2$ implies $f(n_1) < f(n_2)$. From this definition one can easily deduce that a strictly increasing partial function over natural numbers is always injective, and, hence, it is bijective iff it is surjective.

Proposition 2.1 *Let $f : \mathbb{N} \to N$, with $N \subseteq \mathbb{N}$, be a strictly increasing partial function. Then for all $n_1, n_2 \in dom(f)$, if $f(n_1) < f(n_2)$, then $n_1 < n_2$.*

Proposition 2.2 *Let $f : \mathbb{N} \to N$, with $N \subseteq \mathbb{N}$, be a strictly increasing partial function satisfying the following conditions:*

1. *f is surjective (hence, bijective);*
2. *for all $n \in \mathbb{N}$, if $n+1 \in dom(f)$, then $n \in dom(f)$;*
3. *for all $n \in \mathbb{N}$, if $n+1 \in N$, then $n \in N$;*

Then, for all $n \in \mathbb{N}$, if $n \in dom(f)$, then $f(n) = n$, hence f is the identity over $dom(f)$, and $dom(f) = N$.

Event traces: Let $\mathscr{E}$ denotes a possibly infinite set $\mathscr{E}$ of events, called the *event universe*. An event trace over the event universe $\mathscr{E}$ is a partial function $\bar{e} : \mathbb{N} \to \mathscr{E}$ s.t. for all $n \in \mathbb{N}$, if $n+1 \in dom(\bar{e})$, then $n \in dom(\bar{e})$. We call $\bar{e}$ *finite/infinite* iff $dom(\bar{e})$ is finite/infinite, respectively; when $\bar{e}$ is finite, its length $|\bar{e}|$ coincides with the cardinality of $dom(\bar{e})$, while $|\bar{e}|$ is undefined for infinite traces $\bar{e}$. From the definitions above one can easily deduce that if $\bar{e}$ is finite, then $dom(\bar{e}) = \{n \in \mathbb{N} \mid n < |\bar{e}|\}$. We denote with λ the unique trace over $\mathscr{E}$ s.t. $|\lambda| = 0$; when not ambiguous, we denote with e the trace $\bar{e}$ s.t. $|\bar{e}| = 1$ and $\bar{e}(0) = e$.

For simplicity, in the rest of the paper we implicitly assume that all considered event traces are defined over the same event universe.

Concatenation: The concatenation $\bar{e}_1 \cdot \bar{e}_2$ of event trace $\bar{e}_1$ and $\bar{e}_2$ is the trace $\bar{e}$ s.t.

- if $\bar{e}_1$ is infinite, then $\bar{e} = \bar{e}_1$;
- if $\bar{e}_1$ is finite, then $\bar{e}(n) = \bar{e}_1(n)$ for all $n \in dom(\bar{e}_1)$, $\bar{e}(n + |\bar{e}_1|) = \bar{e}_2(n)$ for all $n \in dom(\bar{e}_2)$, and if $\bar{e}_2$ is finite, then $dom(\bar{e}) = \{n \mid n < |\bar{e}_1| + |\bar{e}_2|\}$.

From the definition above one can easily deduce that λ is the identity of $\cdot$, and that $\bar{e}_1 \cdot \bar{e}_2$ is infinite iff $\bar{e}_1$ or $\bar{e}_2$ is infinite. The trace $\bar{e}_1$ is a prefix of $\bar{e}_2$, denoted with $\bar{e}_1 \lhd \bar{e}_2$, iff there exists $\bar{e}$ s.t. $\bar{e}_1 \cdot \bar{e} = \bar{e}_2$. If T_1 and T_2 are two sets of event traces over $\mathscr{E}$, then $T_1 \cdot T_2$ is the set $\{\bar{e}_1 \cdot \bar{e}_2 \mid \bar{e}_1 \in T_1, \bar{e}_2 \in T_2\}$. We write $\bar{e}_1 \lhd T$ to mean that there exists $\bar{e}_2 \in T$ s.t. $\bar{e}_1 \lhd \bar{e}_2$.

Shuffle: The shuffle $\bar{e}_1 \,|\, \bar{e}_2$ of event trace $\bar{e}_1$ and $\bar{e}_2$ is the set of traces T s.t. $\bar{e} \in T$ iff $dom(\bar{e})$ can be partitioned into N_1 and N_2 in such a way that there exist two strictly increasing and bijective[2] partial functions $f_1 : dom(\bar{e}_1) \to N_1$ and $f_2 : dom(\bar{e}_2) \to N_2$ s.t.

$$\bar{e}_1(n_1) = \bar{e}(f_1(n_1)) \text{ and } \bar{e}_2(n_2) = \bar{e}(f_2(n_2)), \text{ for all } n_1 \in dom(\bar{e}_1),\ n_2 \in dom(\bar{e}_2).$$

From the definition above, the definition of λ and Proposition 2.2 one can deduce that $\lambda \,|\, \bar{e} = \bar{e} \,|\, \lambda = \{\bar{e}\}$; it is easy to show that for all $\bar{e} \in \bar{e}_1 \,|\, \bar{e}_2$, $\bar{e}$ is infinite iff $\bar{e}_1$ or $\bar{e}_2$ is infinite, and $|\bar{e}| = n$ iff $|\bar{e}_1| = n_1$, $|\bar{e}_2| = n_2$ and $n = n_1 + n_2$.

If T_1 and T_2 are two sets of event traces over $\mathscr{E}$, then $T_1 \,|\, T_2$ is the set $\bigcup_{\bar{e}_1 \in T_1, \bar{e}_2 \in T_2} (\bar{e}_1 \,|\, \bar{e}_2)$.

[2] Actually, the sufficient condition is surjectivity, but bijectivity can be derived from the fact that the functions are strictly increasing over natural numbers.

Left-preferential shuffle: The left-preferential shuffle $\bar{e}_1 \leftarrow| \bar{e}_2$ of event trace $\bar{e}_1$ and $\bar{e}_2$ is the set of traces $T \subseteq \bar{e}_1 \mid \bar{e}_2$ s.t. $\bar{e} \in T$ iff $dom(\bar{e})$ can be partitioned into N_1 and N_2 in such a way that there exist two strictly increasing and bijective partial functions $f_1 : dom(\bar{e}_1) \to N_1$ and $f_2 : dom(\bar{e}_2) \to N_2$ s.t.

- $\bar{e}_1(n_1) = \bar{e}(f_1(n_1))$ and $\bar{e}_2(n_2) = \bar{e}(f_2(n_2))$, for all $n_1 \in dom(\bar{e}_1)$, $n_2 \in dom(\bar{e}_2)$;
- for all $n_2 \in dom(\bar{e}_2)$, if $m = \min\{n_1 \in dom(\bar{e}_1) \mid f_2(n_2) < f_1(n_1)\}$, then $\bar{e}_1(m) \neq \bar{e}_2(n_2)$.

In the definition above, if[3] $\{n_1 \in dom(\bar{e}_1) \mid f_2(n_2) < f_1(n_1)\} = \emptyset$, then the second condition trivially holds.

As an example, if we have two traces of events $\bar{e}_1 = e_1 \cdot e_2$, and $\bar{e}_2 = e_2 \cdot e_3$, by applying the left-preferential shuffle we obtain the set of traces $\bar{e}_1 \leftarrow| \bar{e}_2 = \{e_1 \cdot e_2 \cdot e_2 \cdot e_3, e_2 \cdot e_3 \cdot e_1 \cdot e_2, e_2 \cdot e_1 \cdot e_3 \cdot e_2, e_2 \cdot e_1 \cdot e_2 \cdot e_3\}$. With respect to $\bar{e}_1 \mid \bar{e}_2$, the trace $e_1 \cdot e_2 \cdot e_3 \cdot e_2$ has been excluded, since this can be obtained only when the first occurrence of e_2 belongs to $\bar{e}_2$; formally, this correponds to the functions $f_1 : \{0,1\} \to \{0,3\}$ and $f_2 : \{0,1\} \to \{1,2\}$ s.t. $f_1(0) = 0$, $f_1(1) = 3$, $f_2(0) = 1$, $f_2(1) = 2$, which satisfy the first item of the definition, but not the second, because $\min\{n_1 \in \{0,1\} \mid f_2(0) = 1 < f_1(n_1)\} = 1$ and $\bar{e}_1(1) = e_2 = \bar{e}_2(0)$; the functions f_1' and f_2' s.t. $f_1'(0) = 0$, $f_1'(1) = 1$, $f_2'(0) = 3$, $f_2'(1) = 2$ satisfy both items, but f_2' is **not** strictly increasing.

Generalized left-preferential shuffle: Given a set of event traces T, the generalized left-preferential shuffle $\bar{e}_1 \leftarrow|_T \bar{e}_2$ of event trace $\bar{e}_1$ and $\bar{e}_2$ w.r.t. T is the set of traces $T' \subseteq \bar{e}_1 \leftarrow| \bar{e}_2$ s.t. $\bar{e} \in T'$ iff $dom(\bar{e})$ can be partitioned into N_1 and N_2 in such a way that there exist two strictly increasing and bijective partial functions $f_1 : dom(\bar{e}_1) \to N_1$ and $f_2 : dom(\bar{e}_2) \to N_2$ s.t.

- $\bar{e}_1(n_1) = \bar{e}(f_1(n_1))$ and $\bar{e}_2(n_2) = \bar{e}(f_2(n_2))$, for all $n_1 \in dom(\bar{e}_1)$, $n_2 \in dom(\bar{e}_2)$;
- for all $n_2 \in dom(\bar{e}_2)$, if $m = \min\{n_1 \in dom(\bar{e}_1) \mid f_2(n_2) < f_1(n_1)\}$, then $\bar{e}'(m) \neq \bar{e}_2(n_2)$ for all $\bar{e}' \in T$ s.t. $m \in dom(\bar{e}')$.

From the definitions of the shuffle operators above one can easily deduce that $\bar{e}_1 \leftarrow|_\emptyset \bar{e}_2 = \bar{e}_1 \mid \bar{e}_2$ and $\bar{e}_1 \leftarrow|_{\{\bar{e}_1\}} \bar{e}_2 = \bar{e}_1 \leftarrow| \bar{e}_2$, for all event traces $\bar{e}_1$, $\bar{e}_2$. This generalisation of the left-preferential shuffle is needed to define the compositional semantics of the shuffle in Section 4. Let us consider $T_1 = \{e_1 \cdot e_2, e_3 \cdot e_4\}$ and $T_2 = \{e_1 \cdot e_5\}$; one might be tempted to define $T_1 \leftarrow| T_2$ as the set $\{\bar{e} \mid \bar{e}_1 \in T_1, \bar{e}_2 \in T_2, \bar{e} \in \bar{e}_1 \leftarrow| \bar{e}_2\}$, which corresponds to $\{e_1 \cdot e_2 \cdot e_1 \cdot e_5, e_1 \cdot e_1 \cdot e_2 \cdot e_5, e_1 \cdot e_1 \cdot e_5 \cdot e_2, e_3 \cdot e_4 \cdot e_1 \cdot e_5, e_3 \cdot e_1 \cdot e_4 \cdot e_5, e_3 \cdot e_1 \cdot e_5 \cdot e_4, e_1 \cdot e_5 \cdot e_3 \cdot e_4, e_1 \cdot e_3 \cdot e_4 \cdot e_5, e_1 \cdot e_3 \cdot e_5 \cdot e_4\}$. But, the last three traces, where e_1 is consumed from T_2 as first event, are not correct, because the event e_1 in T_1 must take the precedence. Thus, the correct definition is given by $\{\bar{e} \mid \bar{e}_1 \in T_1, \bar{e}_2 \in T_2, \bar{e} \in \bar{e}_1 \leftarrow|_{T_1} \bar{e}_2\}$, which does not contain the three traces mentioned above.

3 The RML trace calculus

In this section we define the operational semantics of the trace calculus on which RML is based on. An RML specification is compiled into a term of the trace calculus, which is used as an Intermediate Representation, and then a SWI-Prolog[4] monitor is generated; its execution employs the interpreter of the trace calculus, whose SWI-Prolog implementation is directly driven by the reduction rules defining the labeled transition system of the calculus.

Syntax. The syntax of the calculus is defined in Figure 1. The main basic building block of the calculus

[3] This happens iff in $\bar{e}$ all events of $\bar{e}_1$ precede position n_2, hence, event $\bar{e}_2(n_2)$.

[4] `http://www.swi-prolog.org/`

$$
\begin{array}{lcll}
v & ::= & l \mid \{k_1{:}v_1, \ldots, k_n{:}v_n\} \mid [v_1, \ldots, v_n] & \text{(data value)} \\
b & ::= & x \mid l \mid \{k_1{:}b_1, \ldots, k_n{:}b_n\} \mid [b_1, \ldots, b_n] & \text{(basic data expression)} \\
\theta & ::= & \tau(b_1, \ldots, b_n) & \text{(event type pattern)} \\
t & ::= & \varepsilon & \text{(empty trace)} \\
 & & \theta & \text{(single event)} \\
 & & \mid t_1 \cdot t_2 & \text{(concatenation)} \\
 & & \mid t_1 \wedge t_2 & \text{(intersection)} \\
 & & \mid t_1 \vee t_2 & \text{(union)} \\
 & & \mid t_1 \mid t_2 & \text{(shuffle)} \\
 & & \mid \{\text{let } x; t\} & \text{(parametric expression)}
\end{array}
$$

Figure 1: Syntax of the RML trace calculus: θ is defined inductively, t is defined coinductively on the set of cyclic terms.

is provided by the notion of *event type pattern*, an expression consisting of a name τ of an *event type*, applied to arguments which are *basic data expressions* denoting either variables or the data values (of primitive, array, or object type) associated with the events perceived by the monitor. An event type is a predicate which defines a possibly infinite set of events; an event type pattern specifies the set of events that are expected to occur at a certain point in the event trace; since event type patterns can contain variables, upon a successful match a substitution is computed to bind the variables of the pattern with the data values carried by the matched event.

RML is based on a general object model where events are represented as JavaScript object literals; for instance, the event type `open(`*fd*`)` of arity 1 may represent all events stating 'function call `fs.open` has returned file descriptor *fd*' and having shape `{event:'func_post', name:'fs.open', res:`*fd*`}`. The argument *fd* consists of the file descriptor (an integer value) returned by a call to `fs.open`. The definition is parametric in the variable *fd* which can be bound only when the corresponding event is matched with the information of the file descriptor associated with the property `res`; for instance, `open(42)` matches all events of shape `{event:'func_post', name:'fs.open', res:42}`, that is, all returns from call to `fs.open` with value 42.

Despite RML offers to the users the possibility to define the event types that are used in the specification, for simplicity the calculus is independent of the language used to define event types; correspondingly, the definition of the rewriting system of the calculus is parametric in the relation *match* assigning a semantics to event types (see below).

A specification is represented by a trace expression t built on top of the constant ε (denoting the singleton set with the empty trace), event type patterns θ (denoting the sets of all traces of length 1 with events matching θ), the binary operators (able to combine together sets of traces) of concatenation (juxtaposition), intersection ($\wedge$), union ($\vee$) and shuffle ($|$), and a let-construct to define the scope of variables used in event type patterns.

Differently from event type patterns, which are inductively defined terms, trace expressions are assumed to be cyclic (a.k.a. regular or rational) [23, 29, 5, 6] to provide an abstract support to recursion, since no explicit constructor is needed for it: the depth of a tree corresponding to a trace expression is allowed to be infinite, but the number of its different subtrees must be finite. This condition is proved to be equivalent [23] to requiring that a trace expression can always be defined by a *finite* set[5] of possibly recursive syntactic equations.

[5]The internal representation of cyclic terms in SWI-Prolog is indeed based on such approach.

$$\text{(e-}\varepsilon\text{)}\frac{}{\vdash E(\varepsilon)} \qquad \text{(e-al)}\frac{\vdash E(t_1) \quad \vdash E(t_2)}{\vdash E(t_1\ op\ t_2)}\ op\in\{|,\cdot,\wedge\} \qquad \text{(e-or-l)}\frac{\vdash E(t_1)}{\vdash E(t_1\vee t_2)} \qquad \text{(e-or-r)}\frac{\vdash E(t_2)}{\vdash E(t_1\vee t_2)}$$

$$\text{(e-par)}\frac{\vdash E(t)}{\vdash E(\{\text{let } x; t\})} \qquad \text{(single)}\frac{}{\theta \xrightarrow{e} \varepsilon;\sigma}\ \sigma{=}match(e,\theta) \qquad \text{(or-l)}\frac{t_1 \xrightarrow{e} t_1';\sigma}{t_1\vee t_2 \xrightarrow{e} t_1';\sigma} \qquad \text{(or-r)}\frac{t_1 \xrightarrow{e}\!\!\!\!\!\!/\ \quad t_2 \xrightarrow{e} t_2';\sigma}{t_1\vee t_2 \xrightarrow{e} t_2';\sigma}$$

$$\text{(and)}\frac{t_1 \xrightarrow{e} t_1';\sigma_1 \quad t_2 \xrightarrow{e} t_2';\sigma_2}{t_1\wedge t_2 \xrightarrow{e} t_1'\wedge t_2';\sigma}\ \sigma{=}\sigma_1\cup\sigma_2 \qquad \text{(shuffle-l)}\frac{t_1 \xrightarrow{e} t_1';\sigma}{t_1\,|\,t_2 \xrightarrow{e} t_1'\,|\,t_2;\sigma} \qquad \text{(shuffle-r)}\frac{t_1 \xrightarrow{e}\!\!\!\!\!\!/\ \quad t_2 \xrightarrow{e} t_2';\sigma}{t_1\,|\,t_2 \xrightarrow{e} t_1\,|\,t_2';\sigma}$$

$$\text{(cat-l)}\frac{t_1 \xrightarrow{e} t_1';\sigma}{t_1\cdot t_2 \xrightarrow{e} t_1'\cdot t_2;\sigma} \qquad \text{(cat-r)}\frac{t_1 \xrightarrow{e}\!\!\!\!\!\!/\ \quad t_2 \xrightarrow{e} t_2';\sigma}{t_1\cdot t_2 \xrightarrow{e} t_2';\sigma}\vdash E(t_1) \qquad \text{(par-t)}\frac{t \xrightarrow{e} t';\sigma}{\{\text{let } x; t\} \xrightarrow{e} \sigma_{|x}t';\sigma_{\setminus x}}\ x{\in}dom(\sigma)$$

$$\text{(par-f)}\frac{t \xrightarrow{e} t';\sigma}{\{\text{let } x; t\} \xrightarrow{e} \{\text{let } x; t'\};\sigma}\ x{\notin}dom(\sigma) \qquad \text{(n-}\varepsilon\text{)}\frac{}{\varepsilon \xrightarrow{e}\!\!\!\!\!\!/\ } \qquad \text{(n-single)}\frac{}{\theta \xrightarrow{e}\!\!\!\!\!\!/\ }\ match(e,\theta)\ \text{undef} \qquad \text{(n-or)}\frac{t_1 \xrightarrow{e}\!\!\!\!\!\!/\ \quad t_2 \xrightarrow{e}\!\!\!\!\!\!/\ }{t_1\vee t_2 \xrightarrow{e}\!\!\!\!\!\!/\ }$$

$$\text{(n-and-l)}\frac{t_1 \xrightarrow{e}\!\!\!\!\!\!/\ }{t_1\wedge t_2 \xrightarrow{e}\!\!\!\!\!\!/\ } \qquad \text{(n-and-r)}\frac{t_2 \xrightarrow{e}\!\!\!\!\!\!/\ }{t_1\wedge t_2 \xrightarrow{e}\!\!\!\!\!\!/\ } \qquad \text{(n-and)}\frac{t_1 \xrightarrow{e} t_1';\sigma_1 \quad t_2 \xrightarrow{e} t_2';\sigma_2}{t_1\wedge t_2 \xrightarrow{e}\!\!\!\!\!\!/\ }\ \sigma_1\cup\sigma_2\ \text{undef}$$

$$\text{(n-shuffle)}\frac{t_1 \xrightarrow{e}\!\!\!\!\!\!/\ \quad t_2 \xrightarrow{e}\!\!\!\!\!\!/\ }{t_1\,|\,t_2 \xrightarrow{e}\!\!\!\!\!\!/\ } \qquad \text{(n-cat-l)}\frac{t_1 \xrightarrow{e}\!\!\!\!\!\!/\ }{t_1\cdot t_2 \xrightarrow{e}\!\!\!\!\!\!/\ }\not\vdash E(t_1) \qquad \text{(n-cat-r)}\frac{t_1 \xrightarrow{e}\!\!\!\!\!\!/\ \quad t_2 \xrightarrow{e}\!\!\!\!\!\!/\ }{t_1\cdot t_2 \xrightarrow{e}\!\!\!\!\!\!/\ } \qquad \text{(n-par)}\frac{t \xrightarrow{e}\!\!\!\!\!\!/\ }{\{\text{let } x; t\} \xrightarrow{e}\!\!\!\!\!\!/\ }$$

Figure 2: Transition system for the trace calculus.

Since event type patterns are inductive terms, the definition of free variables for them is standard.

Definition 3.1 *The set of free variables $pfv(\theta)$ occurring in an event type pattern θ is inductively defined as follows:*

$$pfv(x) = \{x\} \qquad pfv(l) = \emptyset$$
$$pfv(\tau(b_1,\ldots,b_n)) = pfv(\{k_1{:}b_1,\ldots,k_n{:}b_n\}) = pfv([b_1,\ldots,b_n]) = \textstyle\bigcup_{i=1\ldots n} pfv(b_i)$$

Given their cyclic nature, a similar inductive definition of free variables for trace expressions does not work; for instance, if $t = \texttt{open}(fd)\cdot t$, a definition of *fv* given by induction on trace expressions would work only for non-cyclic terms and would be undefined for $fv(t)$. Unfortunately, neither a coinductive definition could work correctly since the set S returned by $fv(t)$ has to satisfy the equation $S = \{fd\}\cup S$ which has infinitely many solutions; hence, while an inductive definition of *fv* leads to a partial function which is undefined for all cyclic terms, a coinductive definition results in a non-functional relation *fv*; luckily, such a relation always admits the "least solution" which corresponds to the intended semantics.

Fact 3.1 *Let p be the predicate on trace expressions and set of variables, coinductively defined as follows:*

$$\frac{}{p(\varepsilon,\emptyset)} \qquad \frac{}{p(\theta,S)}\ pfv(\theta){=}S \qquad \frac{p(t,S)}{p(\{\text{let } x; t\}, S\setminus\{x\})} \qquad \frac{p(t_1,S_1) \quad p(t_2,S_2)}{p(t_1\ op\ t_2, S_1\cup S_2)}\ op\in\{|,\cdot,\wedge,\vee\}$$

Then, for any trace expression t, if $L = \bigcap\{S \mid p(t,S)\ holds\}$, then $p(t,L)$ holds.

Proof: By case anaysis on t and coinduction on the definition of $p(t,S)$. □

Definition 3.2 *The set of free variables $fv(t)$ occurring in a trace expression is defined by $fv(t) = \bigcap\{S \mid p(t,S)\ holds\}$.*

Semantics. The semantics of the calculus depends on three judgments, inductively defined by the inference rules in Figure 2. Events e range over a fixed universe of events $\mathscr{E}$. The judgment $\vdash E(t)$ is derivable iff t accepts the empty trace λ and is auxiliary to the definition of the other two judgments $t_1 \xrightarrow{e} t_2;\sigma$ and $t \xrightarrow{e}\!\!\!\!\!\!/\;$; the rules defining it are straightforward and are independent from the remaining judgments, hence a stratified approach is followed and $\vdash E(t)$ and its negation $\nvdash E(t)$ are safely used in the side conditions of the rules for $t_1 \xrightarrow{e} t_2;\sigma$ and $t \xrightarrow{e}\!\!\!\!\!\!/\;$ (see below).

The judgment $t_1 \xrightarrow{e} t_2;\sigma$ defines the single reduction steps of the labeled transition system on which the semantics of the calculus is based; $t_1 \xrightarrow{e} t_2;\sigma$ is derivable iff the event e can be consumed, with the generated substitution σ, by the expression t_1, which then reduces to t_2. The judgment $t \xrightarrow{e}\!\!\!\!\!\!/\;$ is derivable iff there are no reduction steps for event e starting from expression t and is needed to enforce a deterministic semantics and to guarantee that the rules are monotonic and, hence, the existence of the least fixed-point; the definitions of the two judgments are mutually recursive.

Substitutions are finite partial maps from variables to data values which are produced by successful matches of event type patterns; the domain of σ and the empty substitution are denoted by $dom(\sigma)$ and $\emptyset$, respectively, while $\sigma_{|x}$ and $\sigma_{\setminus x}$ denote the substitutions obtained from σ by restricting its domain to $\{x\}$ and removing x from its domain, respectively. We simply write $t_1 \xrightarrow{e} t_2$ to mean $t_1 \xrightarrow{e} t_2;\emptyset$. Application of a substitution σ to an event type patter θ is denoted by $\sigma\theta$, and defined by induction on θ:

$$\sigma x = \sigma(x) \text{ if } x \in dom(\sigma),\ \sigma x = x \text{ otherwise} \qquad \sigma l = l$$
$$\sigma\{k_1{:}b_1,\ldots,k_n{:}b_n\} = \{k_1{:}\sigma b_1,\ldots,k_n{:}\sigma b_n\} \qquad \sigma[b_1,\ldots,b_n] = [\sigma b_1,\ldots,\sigma b_n]$$
$$\sigma\tau(b_1,\ldots,b_n) = \tau(\sigma b_1,\ldots,\sigma b_n)$$

Application of a substitution σ to a trace expression t is denoted by σt, and defined by coinduction on t:

$$\sigma\varepsilon = \varepsilon \qquad \sigma\theta = \sigma\tau(b_1,\ldots,b_n) \text{ if } \theta = \tau(b_1,\ldots,b_n)$$
$$\sigma(t_1\, op\, t_2) = \sigma t_1\, op\, \sigma t_2 \text{ for } op \in \{\cdot,\wedge,\vee,|\} \qquad \sigma\{\text{let } x;\, t\} = \{\text{let } x;\, \sigma_{\setminus x} t\}$$

Since the calculus does not cover event type definitions, the semantics of event types is parametric in the auxiliary partial function *match*, used in the side condition of rules (prefix) and (n-prefix): $match(e,\theta)$ returns the substitution σ iff event e matches event type $\sigma\theta$ and fails (that is, is undefined) iff there is no substitution σ for which e matches $\sigma\theta$. The substitution is expected to be the most general one and, hence, its domain to be included in the set of free variables in θ (see Def. 3.1).

As an example of how *match* could be derived from the definitions of event types in RML, if we consider again the event type `open(`*fd*`)` and *e*=`{event:'func_post', name:'fs.open', res:42}`, then $match(e,$`open(fd)`$) = \{$`fd`$\mapsto 42\}$, while $match(e,$`open(23)`$)$ is undefined.

Except for intersection, which is intrinsically deterministic since both operands need to be reduced, the rules defining the semantics of the other binary operators depend on the judgment $t \xrightarrow{e}\!\!\!\!\!\!/\;$ to force determinism; in particular, the judgment is used to ensure a left-to-right evaluation strategy: reduction of the right operand is possible only if the left hand side cannot be reduced.

The side condition of rule (and) uses the partial binary operator $\cup$ to merge substitutions: $\sigma_1 \cup \sigma_2$ returns the union of σ_1 and σ_2, if they coincide on the intersection of their domains, and is undefined otherwise.

Rule (cat-r) uses the judgment $E(t_1)$ in its side condition: event e consumed by t_2 can also be consumed by $t_1 \cdot t_2$ only if e is not consumed by t_1 (premise $t_1 \xrightarrow{e}\!\!\!\!\!\!/\;$ forcing left-to-right deterministic reduction), and the empty trace is accepted by t_1 (side condition $\vdash E(t_1)$).

Rule (par-t) can be applied when variable x is in the domain of the substitution σ generated by the reduction step from t to t': the substitution $\sigma_{|x}$ restricted to x is applied to t', and x is removed from the

domain of σ, together with its corresponding declaration. If x is not in the domain of σ (rule (par-f)), no substitution and no declaration removal is performed.

The rules defining $t \overset{e}{\not\to}$ are complementary to those for $t \overset{e}{\to} t'$, and the definition of $t \overset{e}{\not\to}$ depends on the judgment $t \overset{e}{\to} t'$ because of rule (n-and): there are no reduction steps for event e starting from expression $t_1 \wedge t_2$, even when $t_1 \overset{e}{\to} t_1'; \sigma_1$ and $t_2 \overset{e}{\to} t_2'; \sigma_2$ are derivable, if the two generated substitutions σ_1 and σ_2 cannot be successfully merged together; this happens when there are two event type patterns that match event e for two incompatible values of the same variable.

Let us consider an example of a cyclic term with the let-construct: $t = \{\text{let}\, fd; open(fd) \cdot close(fd) \cdot t\}$. The trace expression declares a local variable *fd* (the file descriptor), and requires that two immediately subsequent open and close events share the same file descriptor. Since the recursive occurrence of t contains a nested let-construct, the subsequent open and close events can involve a different file descriptor, and this can happen an infinite number of times. In terms of derivation, starting from t, if the event `{event:'func_post', name:'fs.open', res:42}` is observed, which matches `open(42)`, then the substitution $\{fd \mapsto 42\}$ is computed. As a consequence, the residual term $close(42) \cdot t$ is obtained, by substituting *fd* with 42 and removing the let-block. After that, the only valid event which can be observed is `{event:'func_pre',name:'close',args:[42]}`, matching $close(fd)$. Thus, after this rewriting step we get t again; the behavior continues as before, but a different file descriptor can be matched because of the let-block which hides the outermost declaration of *fd*; indeed, the substitution is not propagated inside the nested let-block. Differently from t, the term $\{\text{let}\, fd; t'\}$ with $t' = open(fd) \cdot close(fd) \cdot t'$ would require all open and close events to match a unique global file descriptor. As further explained in Section 5, such example shows how the let-construct is a solution more flexible than the mechanism of trace slicing used in other RV tools to achieve parametricty.

The following lemma can be proved by induction on the rules defining $t \overset{e}{\to} t'; \sigma$.

Lemma 3.1 *If $t \overset{e}{\to} t'; \sigma$ is derivable, then $dom(\sigma) \cup fv(t') \subseteq fv(t)$.*

Since trace expressions are cyclic, they can only contain a finite set of free variables, therefore the domains of all substitutions generated by a possibly infinite sequence of consecutive reduction steps starting from t are all contained in $fv(t)$.

3.1 Semantics based on the transition system

The reduction rules defined above provide the basis for the semantics of the calculus; because of computed substitutions and free variables, the semantics of a trace expression is not just a set of event traces: every accepted trace must be equipped with a substitution specifying how variables have been instantiated during the reduction steps. We call it an *instantiated event trace*; this can be obtained from the pairs of event and substitution traces yield by the possibly infinite reduction steps, by considering the disjoint union of all returned substitutions. Such a notion is needed[6] to allow a compositional semantics. The notion of substitution trace can be given in an analogous way as done for event traces in Section 2. By the considerations related to Lemma 3.1, the substitution associated with an instantiated event trace has always a finite domain, even when the trace is infinite; this means that the substitution is always fully defined after a finite number of reduction steps.

Definition 3.3 *A* concrete instantiated event trace *over the event universe $\mathscr{E}$ is a pair $(\bar{e}, \overline{\sigma})$ of event traces over $\mathscr{E}$, and substitution traces s.t. either $\bar{e}$ and $\overline{\sigma}$ are both infinite, or they are both finite and have the same length, all the substitutions in $\overline{\sigma}$ have mutually disjoint domains and $\bigcup\{dom(\sigma') \mid \sigma' \in \bar{\sigma}\}$ is finite.*

[6] See the example in Section 4.

An abstract instantiated event trace *(instantiated event trace, for short) over $\mathscr{E}$ is a pair $(\bar{e},\sigma)$ where $\bar{e}$ is an event trace over $\mathscr{E}$ and σ is a substitution. We say that $(\bar{e},\sigma)$ is derived from the concrete instantiated event trace $(\bar{e},\bar{\sigma})$, written $(\bar{e},\bar{\sigma}) \rightsquigarrow (\bar{e},\sigma)$, iff $\sigma = \bigcup\{\sigma' \mid \sigma' \in \bar{\sigma}\}$.*

In the rest of the paper we use the meta-variable $\mathscr{I}$ to denote sets of instantiated event traces. We use the notations $\mathscr{I}\downarrow_1$ and $\mathscr{I}\downarrow_2$ to denote the two projections $\{\bar{e} \mid (\bar{e},\sigma) \in \mathscr{I}\}$ and $\{\sigma \mid (\bar{e},\sigma) \in \mathscr{I}\}$, respectively; we write $\bar{e} \lhd \mathscr{I}$ to mean $\bar{e} \lhd \mathscr{I}\downarrow_1$. The notation $\mathscr{I}\downarrow_\omega$ denotes the set $\{(\bar{e},\sigma) \mid (\bar{e},\sigma) \in \mathscr{I}, \bar{e}$ *infinite*$\}$ restricted to infinite traces.

We can now define the semantics of trace expressions.

Definition 3.4 *The concrete semantics $[\![t]\!]_c$ of a trace expression t is the set of concrete instantiated event traces coinductively defined as follows:*

- $(\lambda,\lambda) \in [\![t]\!]_c$ *iff* $\vdash E(t)$ *is derivable;*
- $(e\cdot\bar{e},\sigma\cdot\bar{\sigma}) \in [\![t]\!]_c$ *iff* $t \xrightarrow{e} t';\sigma$ *is derivable and* $(\bar{e},\bar{\sigma}) \in [\![\sigma t']\!]_c$.

The (abstract) semantics $[\![t]\!]$ of a trace expression t is the set of instantiated event traces $\{(\bar{e},\sigma) \mid (\bar{e},\bar{\sigma}) \in [\![t]\!]_c, (\bar{e},\bar{\sigma}) \rightsquigarrow (\bar{e},\sigma)\}$.

The following propositions show that the concrete semantics of a trace expression t as given in Definition 3.4 is always well-defined.

Proposition 3.1 *If $(\bar{e},\bar{\sigma}) \in [\![t]\!]_c$ and $\bar{e}$ is finite, then $|\bar{e}| = |\bar{\sigma}|$.*

Proposition 3.2 *If $(\bar{e},\bar{\sigma}) \in [\![t]\!]_c$ and $\bar{e}$ is infinite, then $\bar{\sigma}$ is infinite as well.*

Proposition 3.3 *If $(\bar{e},\bar{\sigma}) \in [\![t]\!]_c$, then for all $n,m \in \mathbb{N}$, $n \neq m$ implies $dom(\bar{\sigma}(n)) \cap dom(\bar{\sigma}(m)) = \emptyset$.*

Proposition 3.4 *If $(\bar{e},\bar{\sigma}) \in [\![t]\!]_c$, then for all $n \in \mathbb{N}$ $dom(\bar{\sigma}(n)) \subseteq fv(t)$.*

4 Towards a compositional semantics

In this section we show how each basic trace expression operator can be interpreted as an operation over sets of instantiated event traces and we formally prove that such an interpretation is equivalent to the semantics derived from the transition system of the calculus in Definition 3.4, if one considers only contractive terms.

4.1 Composition operators

Left-preferential union: The left-preferential union $\mathscr{I}_1 \overleftarrow{\bigvee} \mathscr{I}_2$ of sets of instantiated event traces $\mathscr{I}_1$ and $\mathscr{I}_2$ is defined as follows: $\mathscr{I}_1 \overleftarrow{\bigvee} \mathscr{I}_2 = \mathscr{I}_1 \bigcup \{(\bar{e},\sigma) \in \mathscr{I}_2 \mid \bar{e} = \lambda \text{ or } (\bar{e} = e\cdot\bar{e}', e \not\lhd \mathscr{I}_1)\}$.

In the deterministic left-preferential version of union, instantiated event traces in $\mathscr{I}_2$ are kept only if they start with an event which is not the first element of any of the traces in $\mathscr{I}_1$ (the condition vacuously holds for the empty trace); since reduction steps can involve only one of the two operands at time, no restriction on the substitutions of the instantiated event traces is required.

Left-preferential concatenation: The left-preferential concatenation $\mathscr{I}_1 \overset{\leftarrow}{\cdot} \mathscr{I}_2$ of sets of instantiated event traces $\mathscr{I}_1$ and $\mathscr{I}_2$ is defined as follows: $\mathscr{I}_1 \overset{\leftarrow}{\cdot} \mathscr{I}_2 = \mathscr{I}_1{\downarrow_\omega} \cup \{(\bar{e}_1 \cdot \bar{e}_2, \sigma) \mid (\bar{e}_1, \sigma_1) \in \mathscr{I}_1, (\bar{e}_2, \sigma_2) \in \mathscr{I}_2, \sigma = \sigma_1 \cup \sigma_2, (\bar{e}_2 = \lambda \text{ or } (\bar{e}_2 = e \cdot \bar{e}_3, (\bar{e}_1 \cdot e) \notin \mathscr{I}_1))\}$.

The left operand $\mathscr{I}_1{\downarrow_\omega}$ of the union corresponds to the fact that in the deterministic left-preferential version of concatenation, all infinite instantiated event traces in $\mathscr{I}_1$ belong to the semantics of concatenation. The right operand of the union specifies the behavior for all finite instantiated event traces $\bar{e}_1$ in $\mathscr{I}_1$; in such cases, the trace in $\mathscr{I}_1 \overset{\leftarrow}{\cdot} \mathscr{I}_2$ can continue with $\bar{e}_2$ in $\mathscr{I}_2$ if $\bar{e}_1$ is not allowed to continue in $\mathscr{I}_1$ with the first event e of $\bar{e}_2$ ($(\bar{e}_1 \cdot e) \notin \mathscr{I}_1$, the condition vacuously holds if $\bar{e}_2$ is the empty trace). Since the reduction steps corresponding to $\bar{e}_2$ follow those for $\bar{e}_1$, the overall substitution σ must meet the constraint $\sigma = \sigma_1 \cup \sigma_2$ ensuring that σ_1 and σ_2 match on the shared variables of the two operands.

Intersection: The intersection $\mathscr{I}_1 \bigwedge \mathscr{I}_2$ of sets of instantiated event traces $\mathscr{I}_1$ and $\mathscr{I}_2$ is defined as follows: $\mathscr{I}_1 \bigwedge \mathscr{I}_2 = \{(\bar{e}, \sigma) \mid (\bar{e}, \sigma_1) \in \mathscr{I}_1, (\bar{e}, \sigma_2) \in \mathscr{I}_2, \sigma = \sigma_1 \cup \sigma_2\}$.

Since intersection is intrinsically deterministic, its semantics throws no surprise.

Left-preferential shuffle: The left-preferential shuffle $\mathscr{I}_1 \overset{\leftarrow}{|} \mathscr{I}_2$ of sets of instantiated event traces $\mathscr{I}_1$ and $\mathscr{I}_2$ is defined as follows: $\mathscr{I}_1 \overset{\leftarrow}{|} \mathscr{I}_2 = \{(\bar{e}, \sigma) \mid (\bar{e}_1, \sigma_1) \in \mathscr{I}_1, (\bar{e}_2, \sigma_2) \in \mathscr{I}_2, \sigma = \sigma_1 \cup \sigma_2, \bar{e} \in \bar{e}_1 \leftarrow|_{\mathscr{I}_1{\downarrow_1}} \bar{e}_2\}$.

The definition is based on the generalized left-preferential shuffle defined in Section 2; an event in $\bar{e}_2$ at a certain position n can contribute to the shuffle only if no trace in $\mathscr{I}_1{\downarrow_1}$ could contribute with the same event at the same position n. Since the reduction steps corresponding to $\bar{e}_1$ and $\bar{e}_2$ are interleaved, the overall substitution σ must meet the constraint $\sigma = \sigma_1 \cup \sigma_2$ ensuring that σ_1 and σ_2 match on the shared variables of the two operands.

Variable deletion: The deletion $\mathscr{I}_{\backslash x}$ of x from the set of instantiated event traces $\mathscr{I}$ is defined as follows: $\mathscr{I}_{\backslash x} = \{(\bar{e}, \sigma_{\backslash x}) \mid (\bar{e}, \sigma) \in \mathscr{I}\}$.

As expected, variable deletion only affects the domain of the computed substitution.

The definitions above show that instantiated event traces are needed to allow a compositional semantics; let us consider the following simplified variation of the example given in Section 3: $t' = \{\text{let } fd; open(fd) \cdot close(fd)\}$. If we did not keep track of substitutions, then the compositional semantics of $open(fd)$ and $close(fd)$ would contain all traces of length 1 matching $open(fd)$ and $close(fd)$, respectively, for any value fd, and, hence, the semantics of $open(fd) \cdot close(fd)$ could not constrain *open* and *close* events to be on the same file descriptor. Indeed, such a constraint is obtained by checking that the substitution of the event trace matching $open(fd)$ can be successfully merged with the substitution of the event trace matching $close(fd)$, so that the two substitutions agree on fd.

4.2 Contractivity

Contractivity is a condition on trace expressions which is statically enforced by the RML compiler; such a requirement avoids infinite loops when an event does not match the specification and the generated monitor would try to build an infinite derivation. Although the generated monitors could dynamically check potential loops dynamically, a syntactic condition enforced statically by the compiler allows monitors to be relieved of such a check, and, thus, to be more efficient.

Contractivity can be seen as a generalization of absence of left recursion in grammars [37]; loops in cyclic terms are allowed only if they are all guarded by a concatenation where the left operand t

cannot contain the empty trace (that is, $\nvdash E(t)$ holds), and the loop continues in the right operand of the concatenation. If such a condition holds, then it is not possible to build infinite derivations for $t_1 \xrightarrow{e} t_2$.

Interestingly enough, such a condition is also needed to prove that the interpretation of operators as given in Section 4.1 is equivalent to the semantics given in Definition 3.4. Indeed, the equivalence result proved in Theorem 4.1 is based on Lemma 4.1 stating that for all contractive term t_1 and event e, there exist t_2 and σ s.t. $t_1 \xrightarrow{e} t_2;\sigma$ is derivable if and only if $t_1 \not\xrightarrow{e}$ is not derivable; such a claim does not hold for a non contractive term as $t = t \vee t$, because for all e, t' and σ, $t \xrightarrow{e} t';\sigma$ and $t \not\xrightarrow{e}$ are not derivable. This is due to the fact that both judgments are defined by an inductive inference system. Intuitively, from a contractive term we cannot derive a new term without passing through at least one concatenation. For instance, considering the term $t = e \cdot t$, we have contractivity because we have to consume e before going inside the loop. But, if we swap the operands, we obtain instead $t = t \cdot e$, where contractivity does not hold; in fact, deriving the concatenation we go first inside the head, but it is cyclic. Since the $\rightarrow$ and $\not\rightarrow$ judgements are defined inductively, both are not derivable because a finite derivation tree cannot be derived for neither of them.

Definition 4.1 *Syntactic contexts $\mathscr{C}$ are inductively defined as follows:*

$$\mathscr{C} \quad ::= \quad \Box \mid \mathscr{C}\, op\, t \mid t\, op\, \mathscr{C} \mid \{\text{let } x; \mathscr{C}\} \quad \textit{with } op \in \{\wedge, \vee, |, \cdot\}$$

Definition 4.2 *A syntactic context $\mathscr{C}$ is contractive if one of the following conditions hold:*

- *$\mathscr{C} = \{\text{let } x; \mathscr{C}'\}$ and $\mathscr{C}'$ is contractive;*
- *$\mathscr{C} = \mathscr{C}'\, op\, t$, $\mathscr{C}'$ is contractive and $op \in \{\cdot, \wedge, \vee, |\}$;*
- *$\mathscr{C} = t\, op\, \mathscr{C}'$, $\mathscr{C}'$ is contractive and $op \in \{\wedge, \vee, |\}$;*
- *$\mathscr{C} = t \cdot \mathscr{C}'$, $\vdash E(t)$ and $\mathscr{C}'$ is contractive;*
- *$\mathscr{C} = t \cdot \mathscr{C}'$ and $\nvdash E(t)$.*

Definition 4.3 *A term is part of t iff it belongs to the least set $partof(t)$ matching the following definition:*

$$partof(\varepsilon) = partof(\theta) = \emptyset \quad partof(\{\text{let } x; t\}) = \{t\} \cup partof(t)$$
$$partof(t_1\, op\, t_2) = \{t_1, t_2\} \cup partof(t_1) \cup partof(t_2) \text{ for } op \in \{|, \cdot, \wedge, \vee\}$$

Because trace expressions can be cyclic, the definition of *partof* follows the same pattern adopted for *fv*. One can prove that a term t is cyclic iff there exists $t' \in partof(t)$ s.t. $t' \in partof(t')$.

Definition 4.4 *A term t is* contractive *iff the following conditions old:*

- *for any syntactic context $\mathscr{C}$, if $t = \mathscr{C}[t]$ then $\mathscr{C}$ is contractive;*
- *for any term t', if $t' \in partof(t)$, then t' is contractive.*

4.3 Main Theorem

We first list all the auxiliary lemmas used to prove Theorem 4.1.

Lemma 4.1 *For all contractive term t_1 and event e, there exist t_2 and σ s.t. $t_1 \xrightarrow{e} t_2;\sigma$ is derivable if and only if $t_1 \not\xrightarrow{e}$ is not derivable.*

Lemma 4.2 *If $(\bar{e}, \bar{\sigma}) \rightsquigarrow (\bar{e}, \sigma)$, then $(\bar{e}, \bar{\sigma}_{\setminus x}) \rightsquigarrow (\bar{e}, \sigma_{\setminus x})$.*

Where $\bar{\sigma}_{\setminus x}$ denotes the substitution sequence where x is removed from the domain of each substitution in $\bar{\sigma}$.

Lemma 4.3 *Given a substitution function σ and a term t, we have that $\sigma t = \sigma_{\setminus x}\sigma_{|x}t = \sigma_{|x}\sigma_{\setminus x}t$, for every $x \in dom(\sigma)$.*

Lemma 4.4 *Let t be a term, σ_1 be a substitution function s.t. $dom(\sigma_1) = \{x\}$; we have that:*

$$\forall_{(\bar{e},\sigma_2)\in[\![t]\!]}.((\sigma_1 \cup \sigma_2 \text{ is defined}) \implies (\bar{e}, \sigma_{2\setminus x}) \in [\![\sigma_1 t]\!])$$

Lemma 4.5 *Let t be a term, σ_1 be a substitution function s.t. $dom(\sigma_1) = \{x\}$; we have that:*

$$\forall_{(\bar{e},\sigma_2)\in[\![\sigma_1 t]\!]}.((\sigma_1 \cup \sigma_2 \text{ is defined}) \implies (\bar{e}, \sigma_2) \in [\![t]\!])$$

Lemma 4.6 $t \overset{e}{\not\to} \iff e \not\triangleleft [\![t]\!]$.

Lemma 4.7 *If $(\bar{e}, \sigma) \in [\![t]\!]$, then $(\bar{e}, \emptyset) \in [\![\sigma t]\!]$.*

Lemma 4.8 *If $(\bar{e}, \bar{\sigma}) \in [\![t]\!]_c$ and $\bar{e}$ is infinite, then $(\bar{e}, \bar{\sigma}) \in [\![t \cdot t']\!]_c$ for every t'.*

Lemma 4.9 *If $e \cdot \bar{e} \in \bar{e}_1 \leftarrow|_T \bar{e}_2$, then $\bar{e}_1 = e \cdot \bar{e}'_1$, or $\bar{e}_2 = e \cdot \bar{e}'_2$ and $e \not\triangleleft \bar{e}_1$.*

Lemma 4.10 *If $(\bar{e}, \bar{\sigma}) \in [\![t]\!]_c$ and $E(t')$, then $(\bar{e}, \bar{\sigma}) \in [\![t \cdot t']\!]_c$.*

Lemma 4.11 *Given $(\bar{e}_1, \bar{\sigma}_1) \in [\![t_1]\!]_c$, $t_2 \xrightarrow{e_2} t_2^1; \sigma_2^1$ and $(\bar{e}_2, \bar{\sigma}_2^2) \in [\![\sigma_2^1 t_2^1]\!]_c$ with $\bar{\sigma}_2 = \sigma_2^1 \cdot \bar{\sigma}_2^2$. If $\bar{e}_1 = e_1 \cdot \ldots \cdot e_n$ is finite, $t_1 \xrightarrow{e_1} t_1^1; \sigma_1^1$, $t_1^1 \xrightarrow{e_2} t_1^2; \sigma_1^2$, ..., $t_1^{n-1} \xrightarrow{e_n} t_1^n; \sigma_1^n$, with $\sigma_1 = \sigma_1^1 \cdot \ldots \cdot \sigma_1^n$ and $t_1^n \overset{e_2}{\not\to}$, then $(\bar{e}_1 \cdot e_2 \cdot \bar{e}_2, \bar{\sigma}_1 \cdot \bar{\sigma}_2) \in [\![t_1 \cdot t_2]\!]_c$.*

In Theorem 4.1 we claim that for every operator of the trace calculus, the compositional semantics is equivalent to the abstract semantics. To prove such claim, we need to show that, for each operator, every trace belonging to the compositional semantics belongs to the abstract semantics, which means we only consider correct traces (*soundness*); and, every trace belonging to the abstract semantics belongs to the compositional semantics, which means we consider all the correct traces (*completeness*).

Each operator requires a customised proofs, but in principle, all the proofs follow the same reasoning. Both soundness and completeness proof start expanding the compositional semantics definition in terms of its concrete semantics, which in turn is rewritten in terms of the operational semantics. At this point, the compositional operator's operands can be separately analysed in order to be recombined with the corresponding trace calculus operator. Finally, the proofs are concluded going backwards from the operational semantics to the abstract one, through the concrete semantics. For all the operators, except $\vee$ and $\wedge$, the proofs are given by coinduction over the terms structure. In every proof which is not analysed separately ($\iff$ cases), we implicitly apply Lemma 4.1.

Theorem 4.1 *The following claims hold for all contractive terms t_1 and t_2:*

- $[\![t_1 \vee t_2]\!] = [\![t_1]\!] \overset{\leftarrow}{\bigvee} [\![t_2]\!]$
- $[\![t_1 \cdot t_2]\!] = [\![t_1]\!] \overset{\leftarrow}{\cdot} [\![t_2]\!]$
- $[\![t_1 \wedge t_2]\!] = [\![t_1]\!] \bigwedge [\![t_2]\!]$
- $[\![t_1 \,|\, t_2]\!] = [\![t_1]\!] \overset{\leftarrow}{|} [\![t_2]\!]$
- $[\![\{\text{let } x; t_1\}]\!] = [\![t_1]\!]_{\setminus x}$

The proofs for the union, intersection, shuffle and let cases are omitted and can be found in the extended version [10]. We decided not to report them due to space constraints. In the proofs that follow, we prove composed implications such as $A_1 \vee \ldots \vee A_n \implies B$, by splitting them into n separate implications $A_1 \implies^1 B, \ldots, A_n \implies^n B$.

The first operator we are going to analyse is the concatenation, where we are going to show that $(\bar{e},\sigma) \in [\![t_1 \cdot t_2]\!] \iff (\bar{e},\sigma) \in [\![t_1]\!] \overset{\leftarrow}{\cdot} [\![t_2]\!]$.

The proof for the empty trace is trivial, and is constructed on top of the definition of the E predicate.

$$
\begin{aligned}
(\lambda,\emptyset) \in [\![t_1 \cdot t_2]\!] \quad &\iff \quad (\lambda,\lambda) \in [\![t_1 \cdot t_2]\!]_c \wedge (\lambda,\lambda) \rightsquigarrow (\lambda,\emptyset) \text{ (by definition of } [\![t]\!]) \\
&\iff \quad E(t_1 \cdot t_2) \text{ is derivable (by definition of } [\![t]\!]_c) \\
&\iff \quad E(t_1) \text{ is derivable } \wedge E(t_2) \text{ is derivable (by definition of } E(t)) \\
&\iff \quad (\lambda,\lambda) \in [\![t_1]\!]_c \wedge (\lambda,\lambda) \in [\![t_2]\!]_c \wedge (\lambda,\lambda) \rightsquigarrow (\lambda,\emptyset) \text{ (by definition of } [\![t]\!]_c) \\
&\iff \quad (\lambda,\emptyset) \in [\![t_1]\!] \wedge (\lambda,\emptyset) \in [\![t_2]\!] \text{ (by definition of } [\![t]\!]) \\
&\iff \quad (\lambda,\emptyset) \in ([\![t_1]\!] \overset{\leftarrow}{\cdot} [\![t_2]\!]) \text{ (by definition of } \overset{\leftarrow}{\cdot})
\end{aligned}
$$

When the trace is not empty, we present the procedure to prove *completeness* ($\implies$) and *soundness* ($\impliedby$), separately.

Let us start with *completeness*. To prove it, we have to show that the abstract semantics $[\![t_1 \cdot t_2]\!]$ (based on the original operational semantics) is included in the composition of the abstract semantics $[\![t_1]\!]$ and $[\![t_2]\!]$, using $\overset{\leftarrow}{\cdot}$ operator. More specifically, in the first part of the proof ($\implies^1$), the first event of the trace belongs to the head of the concatenation. Thus, the head is expanded through operational semantics, causing the term to be rewritten into a concatenation, where the head is substituted with a new term. Since the concrete semantics has been defined coinductively, we can conclude that the proof is satisfied by the so derived concatenation by coinduction. Finally, the proof is concluded recombining the new concatenation in terms of $\overset{\leftarrow}{\cdot}$. The second part of the proof ($\implies^2$) does not require coinduction, since the trace belongs to the tail of the concatenation. Through the operational semantics, the concatenation is rewritten into the new tail, and the proof is straightforwardly concluded following the abstract semantics.

$$
\begin{aligned}
(e \cdot \bar{e},\sigma) \in [\![t_1 \cdot t_2]\!] \quad &\implies \quad (e \cdot \bar{e},\bar{\sigma}) \in [\![t_1 \cdot t_2]\!]_c \wedge (e \cdot \bar{e},\bar{\sigma}) \rightsquigarrow (e \cdot \bar{e},\sigma) \text{ (by definition of } [\![t]\!]) \\
&\implies \quad t_1 \cdot t_2 \xrightarrow{e} t';\sigma' \text{ is derivable } \wedge (\bar{e},\bar{\sigma}') \in [\![\sigma' t']\!]_c \text{ (by definition of } [\![t]\!]_c) \\
&\implies \quad (t_1 \xrightarrow{e} t_1';\sigma' \text{ is derivable } \wedge t_1 \cdot t_2 \xrightarrow{e} t_1' \cdot t_2;\sigma' \text{ is derivable } \wedge \\
&\qquad (\bar{e},\bar{\sigma}') \in [\![\sigma'(t_1' \cdot t_2)]\!]_c) \vee \\
&\qquad (t_1 \not\xrightarrow{e} \wedge E(t_1) \wedge t_2 \xrightarrow{e} t_2';\sigma' \text{ is derivable } \wedge t_1 \cdot t_2 \xrightarrow{e} t_2';\sigma' \text{ is derivable} \wedge \\
&\qquad (\bar{e},\bar{\sigma}') \in [\![\sigma' t_2']\!]_c) \text{ (by operational semantics)} \\
&\implies^1 \quad t_1 \xrightarrow{e} t_1';\sigma' \text{ is derivable } \wedge t_1 \cdot t_2 \xrightarrow{e} t_1' \cdot t_2;\sigma' \text{ is derivable } \wedge \\
&\qquad (\bar{e},\sigma'') \in [\![\sigma'(t_1' \cdot t_2)]\!] \wedge (\bar{e},\bar{\sigma}') \rightsquigarrow (\bar{e},\sigma'') \wedge \sigma = \sigma'' \cup \sigma' \\
&\qquad \text{(by definition of } [\![t]\!]) \\
&\implies^1 \quad t_1 \xrightarrow{e} t_1';\sigma' \text{ is derivable } \wedge t_1 \cdot t_2 \xrightarrow{e} t_1' \cdot t_2 \text{ is derivable } \wedge \\
&\qquad (\bar{e},\sigma'') \in [\![\sigma' t_1']\!] \overset{\leftarrow}{\cdot} [\![\sigma' t_2]\!] \wedge (\bar{e},\bar{\sigma}') \rightsquigarrow (\bar{e},\sigma'') \wedge \sigma = \sigma'' \cup \sigma' \\
&\qquad \text{(by coinduction over } [\![t]\!]) \\
&\implies^1 \quad t_1 \xrightarrow{e} t_1';\sigma' \text{ is derivable } \wedge t_1 \cdot t_2 \xrightarrow{e} t_1' \cdot t_2 \text{ is derivable } \wedge
\end{aligned}
$$

$$
\begin{array}{rl}
 & (\bar{e}_1,\sigma_1'') \in [\![\sigma' t_1']\!] \wedge (\bar{e}_2,\sigma_2'') \in [\![\sigma' t_2]\!] \wedge \bar{e} = \bar{e}_1 \cdot \bar{e}_2 \wedge \\
 & (\bar{e}_2 = \lambda \vee (\bar{e}_2 = e' \cdot \bar{e}_3 \wedge \bar{e}_1 \cdot e' \not\in [\![\sigma' t_1']\!])) \text{ (by definition of } \overset{\leftarrow}{\cdot}) \\
\Longrightarrow^{1} & t_1 \xrightarrow{e} t_1'; \sigma' \text{ is derivable } \wedge (\bar{e}_1, \bar{\sigma}_1) \in [\![\sigma' t_1']\!]_c \wedge (\bar{e}_1, \bar{\sigma}_1) \rightsquigarrow (\bar{e}_1, \sigma_1'') \wedge \\
 & (\bar{e}_2,\sigma_2'') \in [\![\sigma' t_2]\!] \wedge \bar{e} = \bar{e}_1 \cdot \bar{e}_2 \wedge \\
 & (\bar{e}_2 = \lambda \vee (\bar{e}_2 = e' \cdot \bar{e}_3 \wedge \bar{e}_1 \cdot e' \not\in [\![\sigma' t_1']\!])) \text{ (by definition of } [\![t]\!]) \\
\Longrightarrow^{1} & (e \cdot \bar{e}_1, \sigma' \cdot \bar{\sigma}_1) \in [\![t_1]\!]_c \wedge (\bar{e}_1, \bar{\sigma}_1) \rightsquigarrow (\bar{e}_1, \sigma_1'') \wedge \\
 & (\bar{e}_2,\sigma_2'') \in [\![\sigma' t_2]\!] \wedge \bar{e} = \bar{e}_1 \cdot \bar{e}_2 \wedge \\
 & (\bar{e}_2 = \lambda \vee (\bar{e}_2 = e' \cdot \bar{e}_3 \wedge \bar{e}_1 \cdot e' \not\in [\![\sigma' t_1']\!])) \text{ (by definition of } [\![t]\!]_c) \\
\Longrightarrow^{1} & (e \cdot \bar{e}_1, \sigma_1'') \in [\![\sigma' t_1]\!] \wedge (\bar{e}_2, \sigma_2'' \cup \sigma') \in [\![t_2]\!] \wedge \bar{e} = \bar{e}_1 \cdot \bar{e}_2 \wedge \\
 & (\bar{e}_2 = \lambda \vee (\bar{e}_2 = e' \cdot \bar{e}_3 \wedge \bar{e}_1 \cdot e' \not\in [\![\sigma' t_1']\!])) \\
 & \text{(by definition of } [\![t]\!] \text{ and Lemma 4.7)} \\
\Longrightarrow^{1} & (e \cdot \bar{e}, \sigma) \in [\![t_1]\!] \overset{\leftarrow}{\cdot} [\![t_2]\!] \text{ (by definition of } \overset{\leftarrow}{\cdot}) \\
\Longrightarrow^{2} & (e \cdot \bar{e}, \bar{\sigma}) \in [\![t_2]\!]_c \wedge (\lambda, \lambda) \in [\![t_1]\!]_c \wedge t_1 \xrightarrow{e} \text{ (by definition of } [\![t]\!]_c) \\
\Longrightarrow^{2} & (e \cdot \bar{e}, \sigma) \in [\![t_2]\!] \wedge (\lambda, \emptyset) \in [\![t_1]\!] \wedge t_1 \xrightarrow{e} \text{ (by definition of } [\![t]\!]) \\
\Longrightarrow^{2} & (e \cdot \bar{e}, \sigma) \in [\![t_2]\!] \wedge (\lambda, \emptyset) \in [\![t_1]\!] \wedge (\lambda \cdot e) \not\in [\![t_1]\!] \text{ (by Lemma 4.6)} \\
\Longrightarrow^{2} & (e \cdot \bar{e}, \sigma) \in [\![t_1]\!] \overset{\leftarrow}{\cdot} [\![t_2]\!] \text{ (by definition of } \overset{\leftarrow}{\cdot})
\end{array}
$$

We now prove *soundness*. To prove it, we show that the composition of abstract semantics $[\![t_1]\!]$ and $[\![t_2]\!]$ using the $\overset{\leftarrow}{\cdot}$ operator is included in the abstract semantics of the related concatenation term $[\![t_1 \cdot t_2]\!]$. The resulting proof is splitted in four separated cases. When the trace belongs to $[\![t_1]\!]$ is infinite ($\Longrightarrow^{1}$). The proof is based on the fact that an infinite trace concatenated to another trace is always equal to itself. In all the other cases, the proof can be fully derived by a direct application of the operational semantics.

$$
\begin{array}{rcl}
(e \cdot \bar{e}, \sigma) \in [\![t_1]\!] \overset{\leftarrow}{\cdot} [\![t_2]\!] & \Longrightarrow & (e \cdot \bar{e}) \in [\![t_1]\!]\downarrow_\omega \vee \\
 & & (e \cdot \bar{e} = \bar{e}_1 \cdot \bar{e}_2 \wedge (\bar{e}_1, \sigma_1) \in [\![t_1]\!] \wedge (\bar{e}_2, \sigma_2) \in [\![t_2]\!] \wedge \sigma = \sigma_1 \cup \sigma_2 \wedge \\
 & & (\bar{e}_2 = \lambda \vee (\bar{e}_2 = e' \cdot \bar{e}_3 \wedge \bar{e}_1 \cdot e' \not\in [\![t_1]\!]))) \text{ (by definition of } \overset{\leftarrow}{\cdot}) \\
 & \Longrightarrow & (e \cdot \bar{e}) \in [\![t_1]\!]\downarrow_\omega \vee \\
 & & (\bar{e}_1 = \lambda \wedge (\lambda, \emptyset) \in [\![t_1]\!] \wedge (e \cdot \bar{e}, \sigma) \in [\![t_2]\!] \wedge e \not\in [\![t_1]\!]) \vee \\
 & & (\bar{e}_2 = \lambda \wedge (e \cdot \bar{e}) \in [\![t_1]\!] \wedge (\lambda, \emptyset) \in [\![t_2]\!]) \vee \\
 & & (\bar{e}_1 = e \cdot \bar{e}_1' \wedge \bar{e}_2 = e' \cdot \bar{e}_3 \wedge \bar{e}_1 \cdot e' \not\in [\![t_1]\!] \wedge \\
 & & (e \cdot \bar{e}_1, \sigma_1) \in [\![t_1]\!] \wedge (e' \cdot \bar{e}_3) \in [\![t_2]\!] \wedge \sigma = \sigma_1 \cup \sigma_2)
\end{array}
$$

$$
\begin{array}{rcl}
(e \cdot \bar{e}, \sigma) \in [\![t_1]\!]\downarrow_\omega & \Longrightarrow^{1} & (e \cdot \bar{e}, \sigma) \in [\![t_1]\!] \wedge \bar{e} \text{ infinite (by definition of } \downarrow_\omega) \\
 & \Longrightarrow^{1} & (e \cdot \bar{e}, \bar{\sigma}) \in [\![t_1]\!]_c \wedge (e \cdot \bar{e}, \bar{\sigma}) \rightsquigarrow (e \cdot \bar{e}, \sigma) \wedge \\
 & & \bar{e} \text{ infinite (by definition of } [\![t]\!]) \\
 & \Longrightarrow^{1} & t_1 \xrightarrow{e} t_1'; \sigma' \text{ is derivable } \wedge (\bar{e}, \bar{\sigma}') \in [\![\sigma' t_1']\!]_c \wedge \\
 & & (e \cdot \bar{e}, \bar{\sigma}) \rightsquigarrow (e \cdot \bar{e}, \sigma) \wedge \bar{e} \text{ infinite (by definition of } [\![t]\!]_c) \\
 & \Longrightarrow^{1} & t_1 \xrightarrow{e} t_1'; \sigma' \text{ is derivable } \wedge (\bar{e}, \bar{\sigma}') \in [\![\sigma' (t_1' \cdot t_2)]\!]_c \wedge
\end{array}
$$

$$
\begin{array}{rll}
 & (e\cdot\bar{e},\bar{\sigma}) \rightsquigarrow (e\cdot\bar{e},\sigma) \wedge \bar{e} \text{ infinite (by Lemma 4.8)} \\
\Longrightarrow^{1} & t_1\cdot t_2 \xrightarrow{e} t_1'\cdot t_2;\sigma' \text{ is derivable} \wedge (\bar{e},\bar{\sigma}') \in [\![\sigma'(t_1'\cdot t_2)]\!]_c \wedge \\
 & (e\cdot\bar{e},\bar{\sigma}) \rightsquigarrow (e\cdot\bar{e},\sigma) \wedge \bar{e} \text{ infinite (by operational semantics)} \\
\Longrightarrow^{1} & (e\cdot\bar{e},\bar{\sigma}) \in [\![t_1\cdot t_2]\!]_c \wedge (e\cdot\bar{e},\bar{\sigma}) \rightsquigarrow (e\cdot\bar{e},\sigma) \text{ (by definition of } [\![t]\!]_c) \\
\Longrightarrow^{1} & (e\cdot\bar{e},\sigma) \in [\![t_1\cdot t_2]\!] \text{ (by definition of } [\![t]\!])
\end{array}
$$

$$
\begin{array}{rll}
(\bar{e}_1 = \lambda \wedge (\lambda,\emptyset) \in [\![t_1]\!] \wedge & & \\
(e\cdot\bar{e},\sigma) \in [\![t_2]\!] \wedge e \not\triangleleft [\![t_1]\!]) & \Longrightarrow^{2} & E(t_1) \text{ is derivable} \wedge (e\cdot\bar{e},\sigma) \in [\![t_2]\!] \wedge \\
 & & e \not\triangleleft [\![t_1]\!] \text{ (by definition of } [\![t]\!]) \\
 & \Longrightarrow^{2} & E(t_1) \text{ is derivable} \wedge t_2 \xrightarrow{e} t_2';\sigma' \text{ is derivable} \wedge \\
 & & (\bar{e},\bar{\sigma}') \in [\![\sigma' t_2']\!]_c \wedge e \not\triangleleft [\![t_1]\!] \wedge (e\cdot\bar{e},\bar{\sigma}) \rightsquigarrow (e\cdot\bar{e},\sigma) \\
 & & \text{(by definition of } [\![t]\!]_c) \\
 & \Longrightarrow^{2} & t_1\cdot t_2 \xrightarrow{e} t_2';\sigma' \text{ is derivable} \wedge (\bar{e},\bar{\sigma}') \in [\![\sigma' t_2']\!]_c \\
 & & \text{(by operational semantics)} \\
 & \Longrightarrow^{2} & (e\cdot\bar{e},\bar{\sigma}) \in [\![t_1\cdot t_2]\!]_c \wedge (e\cdot\bar{e},\bar{\sigma}) \rightsquigarrow (e\cdot\bar{e},\sigma) \\
 & & \text{(by definition of } [\![t]\!]_c) \\
 & \Longrightarrow^{2} & (e\cdot\bar{e},\sigma) \in [\![t_1\cdot t_2]\!] \text{ (by definition of } [\![t]\!])
\end{array}
$$

$$
(\bar{e}_2 = \lambda \wedge (e\cdot\bar{e}) \in [\![t_1]\!] \wedge (\lambda,\emptyset) \in [\![t_2]\!]) \quad \Longrightarrow^{3} \quad (e\cdot\bar{e},\sigma) \in [\![t_1\cdot t_2]\!] \text{ (by Lemma 4.10)}
$$

$$
\begin{array}{rll}
(\bar{e}_1 = e\cdot\bar{e}_1' \wedge \bar{e}_2 = e'\cdot\bar{e}_3 \wedge & & \\
\bar{e}_1\cdot e' \not\triangleleft [\![t_1]\!] \wedge (e\cdot\bar{e}_1,\sigma_1) \in [\![t_1]\!] \wedge & & \\
(e'\cdot\bar{e}_3,\sigma_2) \in [\![t_2]\!] \wedge \sigma = \sigma_1 \cup \sigma_2) & \Longrightarrow^{4} & t_1 \xrightarrow{e} t_1';\sigma_1' \text{ is derivable} \wedge (\bar{e}_1,\bar{\sigma}_1) \in [\![\sigma_1' t_1]\!]_c \wedge \\
 & & (\bar{e}_1,\bar{\sigma}_1) \rightsquigarrow (\bar{e}_1,\sigma_1'') \wedge t_2 \xrightarrow{e'} t_2';\sigma_2' \text{ is derivable} \wedge \\
 & & (\bar{e}_3,\bar{\sigma}_2') \in [\![\sigma_2' t_2']\!] \wedge (\bar{e}_2,\bar{\sigma}_2') \rightsquigarrow (\bar{e}_2,\sigma_2'') \wedge t_1 \xrightarrow{e'} \text{ is derivable} \\
 & & \wedge \sigma_1 = \sigma_1' \cup \sigma_1'' \wedge \sigma_2 = \sigma_2' \cup \sigma_2'' \text{ (by operational semantics)} \\
 & \Longrightarrow^{4} & (\bar{e}_1\cdot\bar{e}_2,\bar{\sigma}) \in [\![t_1\cdot t_2]\!]_c \text{ (by Lemma 4.11)}
\end{array}
$$

5 Related Work

Compositionality, determinism and events-based semantics are topics very central to concurrent systems. Winskel has introduced the notion of event structure [44] to model computational processes as sets of event occurrences together with relations representing their causal dependencies. Vaandrager [43] has proved that for concurrent deterministic systems it is sufficient to observe the beginning and end of events to derive its causal structure. Lynch and Tuttle have introduced input/output automata [36] to model concurrent and distributed discrete event systems with a trace semantics consisting of both finite and infinite sequences of actions.

The rest of this section describes some of the main RV techniques and state-of-the-art tools and compares them with respect to RML; more comprehensive surveys on RV can be found in literature [25, 30, 35, 41, 26, 13, 31] which mention formalisms for parameterised runtime verification that have not deliberately presented here for space limitation.

Monitor-oriented programming: Similarly as RML, which does not depend on the monitored system and its instrumentation, other proposals introduce different levels of separation of concerns. *Monitor-oriented programming* (*MOP* [19]) is an infrastructure for RV that is neither tied to any particular programming language nor to a single specification language. In order to add support for new logics, one has to develop an appropriate plug-in converting specifications to one of the format supported by the MOP instance of the language of choice; the main formalisms implemented in existing MOP include finite state machines, extended regular expressions, context-free grammars and temporal logics. Finite state machines (or, equivalently, regular expressions) can be easily translated to RML, have limited expressiveness, but are widely used in RV because they are well-understood among software developers as opposite to other more sophisticated approaches, as temporal logics. Extended regular expressions include intersection and complement; although such operators allow users to write more compact specifications, they do not increase the formal expressive power since regular languages are closed under both. Deterministic Context-Free grammars (that is, deterministic pushdown automata) can be translated in RML using recursion, concatenation, union, and the empty trace, while the relationship with Context-Free grammars (that is, pushdown automata) has not been fully investigated yet; as stated in the introduction, RML can express several non Context-Free properties, hence RML cannot be less expressive than Context-Free grammars, but we do not know whether Context-Free grammars are less expressive than RML.

Temporal logics: Since RV has its roots in model checking, it is not surprising that logic-based formalism previously introduced in the context of the latter have been applied to the former. *Linear Temporal Logic* (*LTL*) [38], is one of the most used formalism in verification.

Since the standard semantics of LTL is defined on infinite traces only, and RV monitors can only check finite trace prefixes (as opposed to static formal verification), a three-valued semantics for LTL, named LTL_3 has been proposed [15]. Beyond the basic "true" and "false" truth values, a third "inconclusive" one is considered (LTL specification syntax is unchanged, only the semantics is modified to take into account the new value). This allows one to distinguish the satisfaction/violation of the desired property ("false") from the lack of sufficient evidence among the events observed so far ("inconclusive"), making this semantics more suited to RV. Differently from LTL, the semantics of LTL_3 is defined on finite prefixes, making it more suitable for comparison with other RV formalisms. Further development of LTL_3 led to *RV-LTL* [14], a 4-valued semantics on which RML monitor verdicts are based on.

The expressive power of LTL is the same as of star-free ω-regular languages [39]. When restricted to finite traces, RML is much more expressive than LTL as any regular expression can be trivially translated to it; however, on infinite traces, the comparison is slightly more intricate since RML and LTL_3 have incomparable expressiveness [8]. There exist many extensions of LTL that deal with time in a more quantitative way (as opposed to the strictly qualitative approach of standard LTL) without increasing the expressive power, like *interval temporal logic* [18], *metric temporal logic* [42] and *timed LTL* [15]. Other proposals go beyond regularity [3] and even context-free languages [16].

Several temporal logics are embeddable in *recHML* [34], a variant of the *modal μ-calculus* [33]; this allows the formal study of monitorability [1] in a general framework, to derive results for free about any formalism that can be expressed in such calculi. It would be interesting to study whether the RML trace calculus could be derivable to get theoretical results that are missing from this presentation. Unfortunately, it is not clear whether our calculus and *recHML* are comparable at all. For instance, *recHML* is a fixed-point logic including both least and greatest fixpoint operators, while our calculus implicitly uses a greatest fixpoint semantics for recursion. Nonetheless, *recHML* does not include a shuffle operator, and we are not aware of a way to derive it from other operators.

Regardless of the formal expressiveness, RML and temporal logics are essentially different: RML is closer to formalisms with which software developers are more familiar, as regular expressions and Context-Free languages, but does not offer direct support for time; however, if the instrumentation provides timestamps, then some time-related properties can still be expressed exploiting parametricity.

State machines: As opposite to the language-based approach, as RML, specifications can be defined using *state machines* (a.k.a. automata or finite-state machines). Though the core concept of a finite set of states and a (possibly input-driven) transition function between them is always there, in the field of automata theory different formalizations and extensions bring the expressiveness anywhere from simple deterministic finite automata to Turing machines.

An example of such formalisms is *DATE* (Dynamic Automata with Timers and Events [21]), an extension of the finite-state automata computational model based on communicating automata with timers and transitions triggered by observed events. This is the basis of *LARVA* [22], a Java RV tool focused on control-flow and real-time properties, exploiting the expressiveness of the underlying system (DATE).

The main feature of LARVA that is missing in RML is the support for temporized properties, as observed events can trigger timers for other expected events. On the other hand, the parametric verification support of RML is more general. LARVA scope mechanism works at the object level, thus parametricity is based on *trace slicing* [31] and implemented by spawning new monitors and associating them with different objects. The RML approach is different as specifications can be parametric with respect to any observed data thanks to event type patterns and the let-construct to control the scope of the variables occurring in them. Limitations of the parametric trace slicing approach described above, as well as possible generalizations to overcome them, have been explored by [20, 12, 40].

Finally, the goals of the two tools are different: while RML strives to be system-independent, LARVA is devoted to Java verification, and the implementation relies on AspectJ [32] as an "instrumentation" layer allowing one to inject code (the monitor) to be executed at specific locations in the program.

6 Conclusion

We have moved a first step towards a compositional semantics of the RML trace calculus, by introducing the notion of instantiated event trace, defining the semantics of trace expressions in terms of sets of instantiated event traces and showing how each basic trace expression operator can be interpreted as an operation over sets of instantiated event traces; we have formally proved that such an interpretation is equivalent to the semantics derived from the transition system of the calculus if one considers only contractive terms.

For simplicity, here we have considered only the core of the calculus, but we plan to extend our result to the full calculus, which includes also the prefix closure operator and a top-level layer with constructs to support generic specifications [28]. Another interesting direction for further investigation consists in studying how the notion of contractivity influences the expressive power of the calculus and, hence, of RML; although we have failed so far to find a non-contractive term whose semantics is not equivalent to a corresponding contractive trace expression, we have not formally proved that contractivity does not limit the expressive power of the calculus.

References

[1] L. Aceto, A. Achilleos, A. Francalanza, A. Ingólfsdóttir & K. Lehtinen (2019): *Adventures in Monitorability: From Branching to Linear Time and Back Again*. *Proc. ACM Program. Lang.* 3(POPL), pp. 52:1–52:29, doi:10.1145/3290365.

[2] Wolfgang Ahrendt, Jesús Mauricio Chimento, Gordon J. Pace & Gerardo Schneider (2017): *Verifying data- and control-oriented properties combining static and runtime verification: theory and tools*. *Formal Methods in System Design* 51(1), pp. 200–265, doi:10.1007/s10703-017-0274-y.

[3] Rajeev Alur, Kousha Etessami & P. Madhusudan (2004): *A Temporal Logic of Nested Calls and Returns*. In: *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, pp. 467–481, doi:10.1007/978-3-540-24730-2_35.

[4] 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 & Nobuko Yoshida (2016): *Behavioral Types in Programming Languages*. *Foundations and Trends in Programming Languages* 3(2-3), pp. 95–230, doi:10.1561/2500000031.

[5] Davide Ancona & Andrea Corradi (2014): *Sound and Complete Subtyping between Coinductive Types for Object-Oriented Languages*. In: *ECOOP 2014*, pp. 282–307, doi:10.1007/978-3-662-44202-9_12.

[6] Davide Ancona & Andrea Corradi (2016): *Semantic subtyping for imperative object-oriented languages*. In: *OOPSLA 2016*, pp. 568–587, doi:10.1145/2983990.2983992.

[7] Davide Ancona, Sophia Drossopoulou & Viviana Mascardi (2012): *Automatic Generation of Self-monitoring MASs from Multiparty Global Session Types in Jason*. In: *Declarative Agent Languages and Technologies X - 10th International Workshop, DALT 2012, Valencia, Spain, June 4, 2012, Revised Selected Papers*, pp. 76–95, doi:10.1007/978-3-642-37890-4_5.

[8] Davide Ancona, Angelo Ferrando & Viviana Mascardi (2016): *Comparing Trace Expressions and Linear Temporal Logic for Runtime Verification*. In: *Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*, pp. 47–64, doi:10.1007/978-3-319-30734-3_6.

[9] Davide Ancona, Angelo Ferrando & Viviana Mascardi (2017): *Parametric Runtime Verification of Multiagent Systems*. In: *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017, São Paulo, Brazil, May 8-12, 2017*, pp. 1457–1459, doi:10.5555/3091125.3091328.

[10] Davide Ancona, Angelo Ferrando & Viviana Mascardi (2020): *Can determinism and compositionality coexist in RML? (extende version)*. Available at `https://arxiv.org/abs/2008.06453`.

[11] Davide Ancona, Luca Franceschini, Angelo Ferrando & Viviana Mascardi (2019): *A Deterministic Event Calculus for Effective Runtime Verification*. In Alessandra Cherubini, Nicoletta Sabadini & Simone Tini, editors: *Proceedings of the 20th Italian Conference on Theoretical Computer Science, ICTCS 2019, Como, Italy, September 9-11, 2019*, *CEUR Workshop Proceedings* 2504, CEUR-WS.org, pp. 248–260. Available at `http://ceur-ws.org/Vol-2504/paper28.pdf`.

[12] Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger & David E. Rydeheard (2012): *Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors*. In: *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, pp. 68–84, doi:10.1007/978-3-642-32759-9_9.

[13] Ezio Bartocci, Yliès Falcone, Adrian Francalanza & Giles Reger (2018): *Introduction to Runtime Verification*. In: *Lectures on Runtime Verification - Introductory and Advanced Topics*, pp. 1–33, doi:10.1007/978-3-319-75632-5_1.

[14] Andreas Bauer, Martin Leucker & Christian Schallhart (2007): *The Good, the Bad, and the Ugly, But How Ugly Is Ugly?* In Oleg Sokolsky & Serdar Taşıran, editors: *Runtime Verification*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 126–138, doi:10.1007/978-3-540-77395-5_11.

[15] Andreas Bauer, Martin Leucker & Christian Schallhart (2011): *Runtime verification for LTL and TLTL*. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 20(4), pp. 14:1–14:64, doi:10.1145/2000799.2000800.

[16] Benedikt Bollig, Normann Decker & Martin Leucker (2012): *Frequency Linear-time Temporal Logic*. In: *Sixth International Symposium on Theoretical Aspects of Software Engineering, TASE 2012, 4-6 July 2012, Beijing, China*, pp. 85–92, doi:10.1109/TASE.2012.43.

[17] G. Castagna, M. Dezani-Ciancaglini & L. Padovani (2012): *On Global Types and Multi-Party Session*. *Logical Methods in Computer Science* 8(1), doi:10.2168/LMCS-8(1:24)2012.

[18] Antonio Cau & Hussein Zedan (1997): *Refining Interval Temporal Logic Specifications*. In: *Transformation-Based Reactive Systems Development, 4th International AMAST Workshop on Real-Time Systems and Concurrent and Distributed Software, ARTS'97, Palma, Mallorca, Spain, May 21-23, 1997, Proceedings*, pp. 79–94, doi:10.1007/3-540-63010-4_6.

[19] Feng Chen & Grigore Rosu (2007): *Mop: an efficient and generic runtime verification framework*. In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pp. 569–588, doi:10.1145/1297027.1297069.

[20] Feng Chen & Grigore Rosu (2009): *Parametric Trace Slicing and Monitoring*. In: *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pp. 246–261, doi:10.1007/978-3-642-00768-2_23.

[21] Christian Colombo, Gordon J. Pace & Gerardo Schneider (2008): *Dynamic Event-Based Runtime Monitoring of Real-Time and Contextual Properties*. In: *Formal Methods for Industrial Critical Systems, 13th International Workshop, FMICS 2008, L'Aquila, Italy, September 15-16, 2008, Revised Selected Papers*, pp. 135–149, doi:10.1007/978-3-642-03240-0_13.

[22] Christian Colombo, Gordon J. Pace & Gerardo Schneider (2009): *LARVA – Safer Monitoring of Real-Time Java Programs*. In: *SEFM 2009*, pp. 33–37, doi:10.1109/SEFM.2009.13.

[23] Bruno Courcelle (1983): *Fundamental Properties of Infinite Trees*. *Theor. Comput. Sci.* 25, pp. 95–169, doi:10.1016/0304-3975(83)90059-2.

[24] James C. Davis, Christy A. Coghlan, Francisco Servant & Dongyoon Lee (2018): *The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale*. In: *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pp. 246–256, doi:10.1145/3236024.3236027.

[25] Nelly Delgado, Ann Q. Gates & Steve Roach (2004): *A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools*. *IEEE Trans. Software Eng.* 30(12), pp. 859–872, doi:10.1109/TSE.2004.91.

[26] Yliès Falcone, Klaus Havelund & Giles Reger (2013): *A Tutorial on Runtime Verification*. In: *Engineering Dependable Software Systems*, pp. 141–175, doi:10.3233/978-1-61499-207-3-141.

[27] Yliès Falcone, Srdan Krstic, Giles Reger & Dmitriy Traytel (2018): *A Taxonomy for Classifying Runtime Verification Tools*. In: *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings*, pp. 241–262, doi:10.1007/978-3-030-03769-7_14.

[28] Luca Franceschini (March 2020): *RML: Runtime Monitoring Language*. Ph.D. thesis, DIBRIS - University of Genova. Available at `http://hdl.handle.net/11567/1001856`.

[29] A. Frisch, G. Castagna & V. Benzaken (2008): *Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types*. *J. ACM* 55(4), doi:10.1145/1391289.1391293.

[30] Klaus Havelund & Allen Goldberg (2005): *Verify Your Runs*. In: *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, pp. 374–383, doi:10.1007/978-3-540-69149-5_40.

[31] Klaus Havelund, Giles Reger, Daniel Thoma & Eugen Zalinescu (2018): *Monitoring Events that Carry Data*. In: *Lectures on Runtime Verification - Introductory and Advanced Topics*, pp. 61–102, doi:10.1007/978-3-319-75632-5_3.

[32] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm & William G. Griswold (2001): *An Overview of AspectJ*. In: *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*, pp. 327–353, doi:10.1007/3-540-45337-7_18.

[33] Dexter Kozen (1983): *Results on the Propositional mu-Calculus*. *Theor. Comput. Sci.* 27, pp. 333–354, doi:10.1016/0304-3975(82)90125-6.

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

[35] Martin Leucker & Christian Schallhart (2009): *A brief account of runtime verification*. *The Journal of Logic and Algebraic Programming* 78(5), pp. 293–303, doi:10.1016/j.jlap.2008.08.004.

[36] Nancy A. Lynch & Mark R. Tuttle (1987): *Hierarchical Correctness Proofs for Distributed Algorithms*. In Fred B. Schneider, editor: *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada, August 10-12, 1987*, ACM, pp. 137–151, doi:10.1145/41840.41852.

[37] RC Moore (2000): *Removing left recursion from context-free grammars*. *NAACL 2000: Proceedings of the 1st North American chapter of the Association for Computational Linguistics conference*. Available at `https://www.aclweb.org/anthology/A00-2033`.

[38] Amir Pnueli (1977): *The temporal logic of programs*. In: *18th Annual Symposium on Foundations of Computer Science, 1977*, IEEE, pp. 46–57, doi:10.1109/SFCS.1977.32.

[39] Amir Pnueli & Lenore D. Zuck (1993): *In and Out of Temporal Logic*. In: *Proceedings of the Eighth Annual Symposium on Logic in Computer Science (LICS '93), Montreal, Canada, June 19-23, 1993*, pp. 124–135, doi:10.1109/LICS.1993.287594.

[40] Giles Reger, Helena Cuenca Cruz & David E. Rydeheard (2015): *MarQ: Monitoring at Runtime with QEA*. In: *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pp. 596–610, doi:10.1007/978-3-662-46681-0_55.

[41] Oleg Sokolsky, Klaus Havelund & Insup Lee (2012): *Introduction to the special section on runtime verification*. *STTT* 14(3), pp. 243–247, doi:10.1007/s10009-011-0218-6.

[42] Prasanna Thati & Grigore Rosu (2005): *Monitoring Algorithms for Metric Temporal Logic Specifications*. *Electr. Notes Theor. Comput. Sci.* 113, pp. 145–162, doi:10.1016/j.entcs.2004.01.029.

[43] Frits W. Vaandrager (1991): *Determinism - (Event Structure Isomorphism = Step Sequence Equivalence)*. *Theor. Comput. Sci.* 79(2), pp. 275–294, doi:10.1016/0304-3975(91)90333-W.

[44] Glynn Winskel (1986): *Event Structures*. In Wilfried Brauer, Wolfgang Reisig & Grzegorz Rozenberg, editors: *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, Germany, 8-19 September 1986*, *Lecture Notes in Computer Science* 255, Springer, pp. 325–392, doi:10.1007/3-540-17906-2_31.

A process algebra with global variables

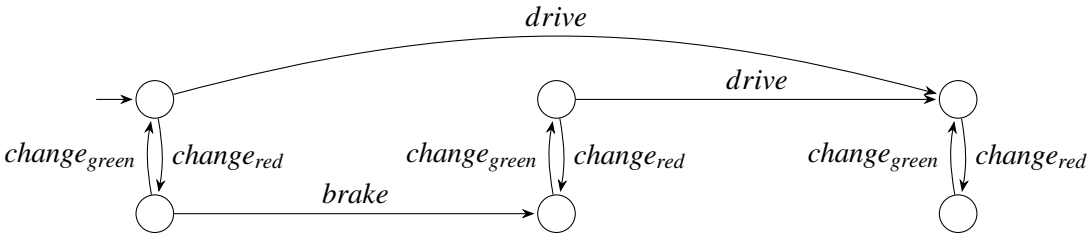
Mark Bouwman Bas Luttik Wouter Schols Tim A.C. Willemse
Eindhoven University of Technology
Eindhoven, The Netherlands
m.s.bouwman@tue.nl s.p.luttik@tue.nl w.r.m.schols@student.tue.nl t.a.c.willemse@tue.nl

In standard process algebra, parallel components do not share a common state and communicate through synchronisation. The advantage of this type of communication is that it facilitates compositional reasoning. For modelling and analysing systems in which parallel components operate on shared memory, however, the communication-through-synchronisation paradigm is sometimes less convenient. In this paper we study a process algebra with a notion of global variable. We also propose an extension of Hennessy-Milner logic with predicates to test and set the values of the global variables, and prove correspondence results between validity of formulas in the extended logic and stateless bisimilarity and between validity of formulas in the extended logic without the set operator and state-based bisimilarity. We shall also present a translation from the process algebra with global variables to a fragment of mCRL2 that preserves the validity of formulas in the extended Hennessy-Milner logic.

1 Introduction

Communication between parallel components in real world systems takes many forms: packets over a network, inter-process communication, communication via shared memory, communication over a bus, etcetera. Process algebras usually offer an abstract message passing feature. Not all forms of communication fit well in a message passing paradigm, in particular, global variables and other forms of shared memory do not fit in well. In some cases it would be desirable to have global variables as first class citizens. To illustrate this we introduce a small example.

Example 1. Consider a traffic light and a car approaching a junction. If the light is green the car performs an action *drive* and moves past the junction. If the light is red the car performs an action *brake* and stops. Once the traffic light is green again the car performs the action *drive*. The specification should result in the following LTS.



It would be natural to model the car and the traffic light as two parallel components. The car and the traffic light need to communicate so that the car can make a decision to drive or brake depending on the current state of the traffic light. Typically, such global information is modelled by introducing an extra parallel component that maintains the global information, in this case the colour of the traffic light. However, taking that approach we obtain a different LTS, which has an extra transition modelling

O. Dardha and J. Rot (Eds.): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics (EXPRESS/SOS 2020).
EPTCS 322, 2020, pp. 33–50, doi:10.4204/EPTCS.322.5

the communication of car and traffic light with the extra component. Moreover, one must take care that decisions to drive or brake are made on the basis of up-to-date information, e.g., by implementing a protocol that locks the additional component. In many cases it is realistic that the observed value is no longer up to date and in some cases we are also interested in analysing the consequences of this. In other cases however, we might want to abstract from such complications. When the information is constantly available to the observers, as is the case with a traffic light, we have even more reason to not introduce separate transitions communicating the global information.

In some process algebras it is possible to define a communication function that specifies that a *drive* action is the result of a communication between two actions of parallel components, e.g. a $drive\_if\_green$ action from the car and a *signal_green* action from the traffic light. This is somewhat unnatural as the traffic light does not really actively participate in the driving of the car. Moreover, if we introduce a second car that only wants to drive when the traffic light is red we would need to change the communication function, even though the communication of information does not essentially change. It would be better to let the colour of the traffic light be in a global variable. In that way the behaviour of the traffic light and cars acting upon information from the traffic light is more separated, we obtain a better separation of concerns.

In the early days of process algebra, doing away with global variables in favour of message passing and local variables was an important step to further develop the field [2]. Since then there have, nevertheless, been some efforts to reintroduce notions of globally available data.

In [1] propositional signals and a state operator are presented. The state operator tracks the value of some information. Based on the current value a number of propositions can be signalled to the rest of the system. In the example of the traffic lights the process modelling the traffic light could track the state of the light, emitting signals such as $lightGreen$ and $\neg lightRed$, which can in turn be used as conditions in the process modelling the car. In this approach the value of the global variable is not communicated directly, which restricts conditions based on global variables to propositional logic.

Other approaches, such as the one presented in [11, Chapter 19], model global variables as separate parallel processes and use a protocol to ensure only one process accesses a global variable at the time. This approach introduces extra internal steps, which increase the statespace. Moreover, it introduces divergence when a process locks a global variable, reads the value, concludes that it cannot make a step and unlocks the variable again.

Formalisms based on Concurrent Constraint Programming (CCP) [12] have global data at their core. In CCP a central store houses a set of constraints. Concurrent processes may *tell* a constraint, adding it to the global store or *ask* a constraint, checking whether it is entailed by the constraints in the store. An ask will block until other processes have added sufficient constraints to the store. Process calculi based on the coordination language LINDA [10] also use global data. In these process calculi there is a global set of data elements. Similarly to CCP, processes may tell a data element (adding it to the global set) or ask a data element (checking whether it is in the set). Additionally, processes may *get* an element, removing it from the data set. LINDA does not have a concept of variables, just a central set of data elements. Generally, process calculi based on CCP or LINDA do not allow asking a constraint/data element and acting upon the information in a single step.

The goal of our work is to propose and study (i) a process calculus with global variables (ii) a modal logic that can refer to the values of global variables (iii) an encoding in an existing process calculus and logic with tool support. In this paper we propose a simple process calculus where every component of the system can access the current value of global variables directly. We define appropriate notions of equivalence for our process calculus. Our first contribution is an extension of the Hennessy-Milner

Logic (HML) with two new operators that is strong enough to differentiate non-equivalent process expressions. Our second contribution is an encoding of our process algebra in mCRL2 and our extended logic in standard HML. This encoding is such that the translated formula holds for the translated process expression if and only if the original formula holds for the original process expression.

This paper is organised as follows. In Section 2 we define a simple process algebra with global variables. In Section 3 we give appropriate notions of equivalence for our process algebra. We continue by defining an extension of the Hennessy-Milner Logic in Section 4 and relating it to our equivalence notions in Section 5. In Section 6 we show how our process algebra with global variables can be encoded in mCRL2. Sections 7 and 8 discuss the results and conclude this work.

2 A simple process algebra with global variables

In this section we will introduce a process algebra with global variables and its semantics. For convenience we will, in this paper, assume a single data domain D. We will use Var to denote the finite set of global variable names.

We presuppose a set of actions Act and derive a set of transition labels $TL \stackrel{\text{def}}{=} Act \cup \{assign(v,d) \mid v \in Var \wedge d \in D\}$. We also presuppose a set of process names PN. The set of process expressions $\mathscr{P}$ is generated by the following grammar containing *action prefix, inaction, choice, parallelism, encapsulation, recursion and conditionals*:

$$P := \lambda.P \mid \delta \mid P+P \mid P\|P \mid \partial_B(P) \mid X \mid (v=d) \rightarrow P,$$

where $\lambda \in TL$, $B \subseteq Act$, $X \in PN$, $v \in Var$ and $d \in D$. Inaction is similar to the process constant 0 in, for example, CCS [8] and TCP [1]. Our process algebra supports recursion because we also define a recursive specification E defining the process names. Let a recursive equation be an equation of the form $X \stackrel{\text{def}}{=} t$ with $X \in PN$ and t a process expression in $\mathscr{P}$. A recursive specification contains one recursive equation $X \stackrel{\text{def}}{=} t$ for every $X \in PN$. Every recursive specification should be guarded. This means that every occurrence of X in t is in the scope of an action prefix. For communication between parallel processes we use an ACP style communication function. We presuppose a binary communication function on the set of actions, i.e., a partial function $\gamma : Act \times Act \rightharpoonup Act$ that is commutative and associative. We only allow handshakes (communication between two parties): if $\gamma(a,b) = c$ then $\gamma(c,d)$ is undefined for every d.

Let $\mathscr{V}$ be the set of all functions $Var \rightarrow D$, i.e. the set of all valuations. Let $V \in \mathscr{V}$; we denote by $V[v \mapsto d]$ the assignment defined, for all $v' \in Var$ by:

$$V[v \mapsto d](v') = \begin{cases} d & \text{if } v' = v \\ V(v') & \text{if } v' \neq v \end{cases}$$

In Definition 2 we give the usual definition of Labelled Transition Systems (LTSs).

Definition 2. A Labelled Transition System (LTS) is a tuple $(S, L, \rightarrow, s)$, where

- S is a set of states,
- L is a set of transition labels,
- $\rightarrow \subseteq S \times L \times S$ is the transition relation,
- $s \in S$ is the initial state.

$$\text{(PREF)}\ \frac{}{\langle a.P,V\rangle \xrightarrow{a} \langle P,V\rangle} \qquad \text{(ASGN)}\ \frac{}{\langle assign(v,d).P,V\rangle \xrightarrow{assign(v,d)} \langle P,V[v\mapsto d]\rangle}$$

$$\text{(CON)}\ \frac{\langle P,V\rangle \xrightarrow{\lambda} \langle P',V'\rangle}{\langle (v=d)\to P,V\rangle \xrightarrow{\lambda} \langle P',V'\rangle} V(v)=d \qquad \text{(REC)}\ \frac{\langle P,V\rangle \xrightarrow{\lambda} \langle P',V'\rangle}{\langle X,V\rangle \xrightarrow{\lambda} \langle P',V'\rangle} X \stackrel{\text{def}}{=} P$$

$$\text{(SUM-L)}\ \frac{\langle P,V\rangle \xrightarrow{\lambda} \langle P',V'\rangle}{\langle P+Q,V\rangle \xrightarrow{\lambda} \langle P',V'\rangle} \qquad \text{(SUM-R)}\ \frac{\langle Q,V\rangle \xrightarrow{\lambda} \langle Q',V'\rangle}{\langle P+Q,V\rangle \xrightarrow{\lambda} \langle Q',V'\rangle}$$

$$\text{(PAR-L)}\ \frac{\langle P,V\rangle \xrightarrow{\lambda} \langle P',V'\rangle}{\langle P\parallel Q,V\rangle \xrightarrow{\lambda} \langle P'\parallel Q,V'\rangle} \qquad \text{(PAR-R)}\ \frac{\langle Q,V\rangle \xrightarrow{\lambda} \langle Q',V'\rangle}{\langle P\parallel Q,V\rangle \xrightarrow{\lambda} \langle P\parallel Q',V'\rangle}$$

$$\text{(COMM)}\ \frac{\langle P,V\rangle \xrightarrow{a} \langle P',V\rangle \quad \langle Q,V\rangle \xrightarrow{b} \langle Q',V\rangle}{\langle P\parallel Q,V\rangle \xrightarrow{c} \langle P'\parallel Q',V\rangle} \gamma(a,b)=c$$

$$\text{(ENC)}\ \frac{\langle P,V\rangle \xrightarrow{\lambda} \langle P',V'\rangle}{\langle \partial_B(P),V\rangle \xrightarrow{\lambda} \langle \partial_B(P'),V'\rangle} \lambda \notin B$$

Table 1: Structural operational semantics.

We now want to associate an LTS with the process algebra. As the behaviour of a process expression depends on the valuation of global variables a state is a pair $\langle P,V\rangle$ of a process expression P and a valuation function V. The set of states is $\mathscr{P}\times\mathscr{V}$. The transition relation is the least relation on states satisfying the rules of the structural operational semantics (see Table 1).

Note that we only allow processes to synchronise on actions and not on assignments. This design decision was made since assignments change the valuation function, whereas actions cannot change the valuation. When two processes synchronise on assignments then it is not clear what the resulting effect on the value of the variable should be.

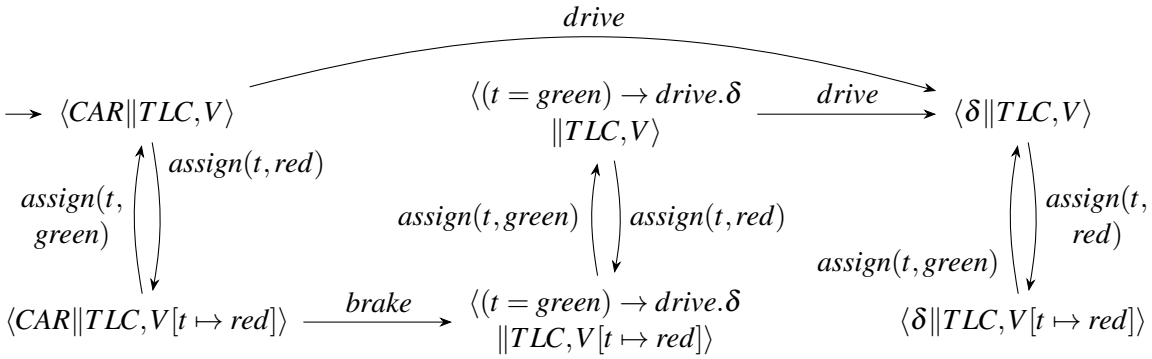
Example 3. Consider the interaction between a car and a traffic light controller (TLC). The TLC sets the colour of a traffic light which the driver of the car acts upon. There is one global variable t and the data domain consists of two elements $D=\{green,red\}$. The recursive specification consists of two process equations, given below.

$$CAR \stackrel{\text{def}}{=} ((t=green)\to drive.\delta) + ((t=red)\to brake.((t=green)\to drive.\delta))$$

$$TLC \stackrel{\text{def}}{=} ((t=green)\to assign(t,red).TLC)$$

$$+((t=red)\to assign(t,green).TLC)$$

Using the SOS we can derive an LTS with $\langle CAR||TLC,V\rangle$ as initial state, where $V(t)=green$. Note that this LTS is isomorphic to the LTS presented in Example 1. We only show the states reachable from the initial state. The initial state is marked with an arrow pointing to it.



3 Equivalence of process expressions

We will examine equivalence relations in the context of global variables. To start we note that we can examine equivalence on two levels: on the level of process expressions and on the level of pairs of process expression together with an initial valuation (from which we can derive an LTS). We begin by exploring the equivalence of process expressions.

We require of the equivalence relation on process expressions that if P and Q are equivalent then we can safely replace P with Q in any larger process expression. In other words, the equivalence relation on process expressions should be a congruence for the process algebra.

Typically, equivalence of process expressions is established by a notion of bisimilarity. Most variants of bisimulation only consider the labels on transitions. Strong bisimilarity (defined in Definition 4) is, however, not a congruence for our process algebra, which we will demonstrate with an example.

Definition 4. Strong bisimilarity: A relation $\mathscr{R} \subseteq S \times S$, where S is the set of states of an LTS, is a strong bisimulation relation if and only if for all states s and t and labels λ we have $(s,t) \in \mathscr{R}$ implies that

- for all states s': $s \xrightarrow{\lambda} s'$ implies there exists a state t' such that $t \xrightarrow{\lambda} t'$ and $(s',t') \in \mathscr{R}$,
- for all states t': $t \xrightarrow{\lambda} t'$ implies there exists a state s' such that $s \xrightarrow{\lambda} s'$ and $(s',t') \in \mathscr{R}$.

Two states s and t are strongly bisimilar, denoted by $s \underline{\leftrightarrow} t$, if and only if there exists a strong bisimulation relation $\mathscr{R}$ such that $(s,t) \in \mathscr{R}$.

Example 5. Consider process expressions $P = (v = 0) \rightarrow a.\delta$ and $Q = a.\delta$. Note that P and Q are simply abbreviations of process expressions, not process names. Let V map a global variable v to 0 and $D = \{0,1\}$. The reachable fragments of the LTSs with $\langle P,V\rangle$ and $\langle Q,V\rangle$ as initial state are shown in Figure 1.

Figure 1: Part of the transition system space of P and Q

Processes P and Q seem behaviourally equivalent looking at the reachable transitions, the states $\langle P,V\rangle$ and $\langle Q,V\rangle$ are in fact strongly bisimilar. The problem arises when we add a parallel component that can

assign a different value to the global variable. Let us consider the process expression $R = assign(v,1).\delta$. The reachable fragments of the LTSs with $P\|R$ and $Q\|R$ as initial state are shown in Figure 2. Clearly $P\|R$ and $Q\|R$ are not strongly bisimilar and therefore strong bisimilarity is not a congruence for our process algebra.

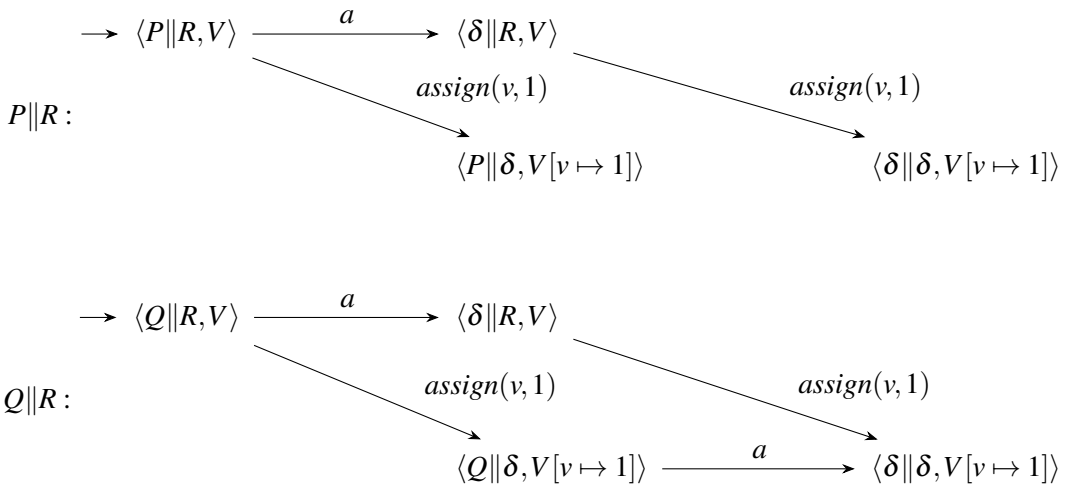


Figure 2: Part of the transition system space of P and Q

We will use the notion of stateless-bisimilarity, defined in [9], as an equivalence relation on process expressions. In essence, stateless-bisimilarity relates process expressions that behave the same under any valuation.

Definition 6. Stateless-bisimilarity: A relation $\mathscr{R}_{sl} \subseteq \mathscr{P} \times \mathscr{P}$ is a stateless bisimulation relation if and only if for all process expressions P and Q and labels λ we have $(P,Q) \in \mathscr{R}_{sl}$ implies that

- for all process expressions P' and valuation functions $V,V' \in \mathscr{V}$: $\langle P,V\rangle \xrightarrow{\lambda} \langle P',V'\rangle$ implies there exists a process expression Q' such that $\langle Q,V\rangle \xrightarrow{\lambda} \langle Q',V'\rangle$ and $(P',Q') \in \mathscr{R}_{sl}$,
- for all process expressions Q' and valuation functions $V,V' \in \mathscr{V}$: $\langle Q,V\rangle \xrightarrow{\lambda} \langle Q',V'\rangle$ implies there exists a process expression P' such that $\langle P,V\rangle \xrightarrow{\lambda} \langle P',V'\rangle$ and $(P',Q') \in \mathscr{R}_{sl}$.

Two process expressions P and Q are stateless bisimilar, denoted by $P \underline{\leftrightarrow}_{sl} Q$, if and only if there exists a stateless bisimulation relation $\mathscr{R}_{sl}$ such that $(P,Q) \in \mathscr{R}_{sl}$.

The deduction system of our process algebra is in process-tyft format from which it follows that stateless-bisimilarity is a congruence [9].

In the case that global variables cannot be changed by the environment and we have a specific initial valuation in mind we might not care about the behaviour under valuations that will never occur. To that end we use state-based bisimilarity [9], which is defined on states rather than process expressions.

Definition 7. State-based bisimilarity: A relation $\mathscr{R}_{sb} \subseteq (\mathscr{P} \times \mathscr{V}) \times (\mathscr{P} \times \mathscr{V})$ is a state-based bisimulation relation if and only if for all states $\langle P,V_1\rangle$ and $\langle Q,V_2\rangle$ and labels λ we have $(\langle P,V_1\rangle,\langle Q,V_2\rangle) \in \mathscr{R}_{sb}$ implies that $V_1 = V_2$ and

- for all process expressions P' and valuation functions V': $\langle P,V_1\rangle \xrightarrow{\lambda} \langle P',V'\rangle$ implies there exists a process expression Q' such that $\langle Q,V_2\rangle \xrightarrow{\lambda} \langle Q',V'\rangle$ and $(\langle P',V'\rangle,\langle Q',V'\rangle) \in \mathscr{R}_{sb}$,

- for all process expressions Q' and valuation functions V': $\langle Q,V_2\rangle \xrightarrow{\lambda} \langle Q',V'\rangle$ implies there exists a process expression P' such that $\langle P,V_1\rangle \xrightarrow{\lambda} \langle P',V'\rangle$ and $(\langle P',V'\rangle,\langle Q',V'\rangle) \in \mathscr{R}_{sb}$.

Two states $\langle P,V_1\rangle$ and $\langle Q,V_2\rangle$ are state-based bisimilar, denoted by $\langle P,V_1\rangle \underline{\leftrightarrow}_{sb} \langle Q,V_2\rangle$, if and only if there exists a state-based bisimulation relation $\mathscr{R}_{sb}$ such that $(\langle P,V_1\rangle,\langle Q,V_2\rangle) \in \mathscr{R}_{sb}$.

State-based bisimilarity is not a congruence for our process algebra, the problem shown in Example 5 applies.

There is a relation between stateless-bisimilarity and state-based bisimilarity. For any two process expressions P and Q we have that if $P \underline{\leftrightarrow}_{sl} Q$ then also $\langle P,V\rangle \underline{\leftrightarrow}_{sb} \langle Q,V\rangle$ for all valuations $V \in \mathscr{V}$ [9].

State-based bisimilarity distinguishes LTSs on the valuation that is in a state: two states that are strongly bisimilar may not be state-based bisimilar due to differences in valuations in reachable states. This takes into account that the value of global variables may be essential to the modelled system and may be visible to the environment.

4 Hennessy-Milner logic

In order to reason about properties of a process expression or system specification we define a logic. Standard Hennessy-Milner Logic (HML) [7] is insufficient for our purpose, for two reasons. The first reason is that we would like to conveniently refer to global variables in the logic. The second reason for extending the logic is that we want a correspondence between the logic and stateless bisimilarity. Process expressions $a.\delta$ and $(v = 0) \rightarrow a.\delta$ are not stateless bisimilar but in the case that we have a valuation function V that maps v to 0 then states $\langle a.\delta,V\rangle$ and $\langle (v = 0) \rightarrow a.\delta,V\rangle$ cannot be distinguished using HML.

We extend HML with two new operators. The first operator is the check operator $(v = e)$. This operator returns a boolean which is true if and only if a global variable v has value e. The second operator is the set operator $\downarrow (v := e)$. The set operator sets the value of a global variable v to e. This results in the following syntax for our logic:

$$\phi := true \mid false \mid (v = e) \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \langle T\rangle\phi \mid [T]\phi \mid \downarrow (v := e)\phi$$

where T is a nonempty finite set of transition labels. Depending on whether we include the check operator, the set operator or both, we will refer to the logic with HML^{check}, HML^{set} or HML$^{check+set}$, respectively.

The formula *true* holds in every state and $false$ holds in no states. The operators $\neg,\wedge,\vee$ have their usual meaning. The diamond operator $\langle T\rangle\phi$ is true in a state s if and only if a transition $s \xrightarrow{\lambda} s'$ exists where ϕ holds in s' and $\lambda \in T$. The box operator $[T]\phi$ holds in a state s if and only if for every state s' and transition label $\lambda \in T$ we have $s \xrightarrow{\lambda} s'$ implies ϕ holds in s'.

The check operator $(v = e)$ is true in a state $\langle P,V\rangle$ if and only if $V(v) = e$. The set operator $\downarrow$ indicates that $\downarrow (v := e)\phi$ is true in all states $\langle P,V\rangle$ if and only if ϕ is true in $\langle P,V[v \mapsto e]\rangle$. Note that the set operator allows us to reason about parts of an LTS that are not reachable from the initial state. Further note that the set operator allows us to distinguish $\langle a.\delta,V\rangle$ and $\langle (v = 0) \rightarrow a.\delta,V\rangle$ even if $V(v) = 0$ the formula $\downarrow (v := 1)\langle\{a\}\rangle true$ distinguishes them. We will use the notation $\downarrow (V)$, $V \in \mathscr{V}$, to set the value of all global variables to the value specified by V. This is a shorthand for a sequence of regular set operations. Note that the number of global variables is finite and the order of set operations is irrelevant in the sequence of set operations as each sets a different variable.

4.1 Semantics

In this section we will define semantic rules to obtain all states that satisfy a HML$^{check+set}$ formula. We have obtained the semantics of standard HML from [6]. Let ϕ be a modal formula, let $(S, L, \rightarrow, s)$ be an LTS. We inductively define the interpretation of ϕ, notation $[\![\phi]\!]$, where $[\![\phi]\!]$ contains all states $u \in S$ where ϕ is true. Note that the check and set operators are only defined for LTSs where states consist of both a process expression and a valuation.

$$
\begin{array}{lcl}
[\![true]\!] & = & S \\
[\![false]\!] & = & \emptyset \\
[\![v = e]\!] & = & \{\langle P, V\rangle \in S \mid V(v) = e\} \\
[\![\neg\phi]\!] & = & S \setminus [\![\phi]\!] \\
[\![\phi \wedge \phi']\!] & = & [\![\phi]\!] \cap [\![\phi']\!] \\
[\![\phi \vee \phi']\!] & = & [\![\phi]\!] \cup [\![\phi']\!] \\
[\![\langle T\rangle\phi]\!] & = & \{u \in S \mid \exists u' \in [\![\phi]\!], \lambda \in T : u \xrightarrow{\lambda} u'\} \\
[\![[T]\phi]\!] & = & \{u \in S \mid \forall u' \in S, \lambda \in T : (u \xrightarrow{\lambda} u') \Rightarrow u' \in [\![\phi]\!]\} \\
[\![\downarrow(v := e)\phi]\!] & = & \{\langle P, V\rangle \in S \mid \langle P, V[v \mapsto e]\rangle \in [\![\phi]\!]\}
\end{array}
$$

5 Relation logic and bisimilarity

There is a nice correspondence between strong bisimilarity and HML: two states in an LTS are strongly bisimilar if and only if they satisfy the same HML formulas [7]. This correspondence is often called the Hennessy-Milner theorem. We would like a similar correspondence between process expressions and states and the extended HML. First, we introduce the notion of an image-finite process. As an LTS contains all possible process expressions (and valuations) we want to impose image-finiteness only for reachable states and process expressions, so we start by defining reachability.

Definition 8. Reachability states: A state s' is reachable from a state s if there exist states $s_0, \ldots, s_n$ and labels $\lambda_1, \ldots, \lambda_n$ such that $s = s_0 \wedge s_0 \xrightarrow{\lambda_1} s_1 \wedge \cdots \wedge s_{n-1} \xrightarrow{\lambda_n} s_n \wedge s_n = s'$.

Definition 9. Reachability process expressions: Process expression P' is reachable from a process expression P if there exist processes $P_0, \ldots, P_n$ and labels $\lambda_1, \ldots, \lambda_n$ such that $P = P_0 \wedge \exists_{V_0, V_1} \langle P_0, V_0\rangle \xrightarrow{\lambda_1} \langle P_1, V_1\rangle \wedge \cdots \wedge \exists_{V_{n-1}, V_n} \langle P_{n-1}, V_{n-1}\rangle \xrightarrow{\lambda_n} \langle P_n, V_n\rangle \wedge P_n = P'$.

Definition 10. Image-finiteness: A state $\langle P, V\rangle$ is image finite if and only if the set $\{\langle P', V'\rangle \mid \langle P, V\rangle \xrightarrow{\lambda} \langle P', V'\rangle\}$ is finite for every label λ. A process expression P is image finite if and only if for every process expression P' reachable from P and every valuation V the state $\langle P', V\rangle$ is image-finite.

An example of a state that is not image finite is $\langle A, V\rangle$, with $A \stackrel{\text{def}}{=} a.\delta || A$.

We can now prove the following two correspondences on the level of process expressions and states.

Theorem 11. Let P and Q be two image-finite process expressions. Then $P \underline{\leftrightarrow}_{sl} Q$ if and only if for all valuations V and all HML$^{check+set}$ formulas ϕ we have that $\langle P, V\rangle \in [\![\phi]\!] \Leftrightarrow \langle Q, V\rangle \in [\![\phi]\!]$.

Proof. We prove the two implications separately. To prove the implication from left to right assume $P \underline{\leftrightarrow}_{sl} Q$. The proof that for some HML$^{check+set}$ formula ϕ we have that $\langle P, V\rangle \in [\![\phi]\!]$ if and only if $\langle Q, V\rangle \in [\![\phi]\!]$ is straightforward by induction on the structure of ϕ.

For the implication from right to left we assume that $\langle P,V\rangle$ and $\langle Q,V\rangle$ satisfy exactly the same formulae in HML$^{check+set}$. We shall prove that $\langle P,V\rangle \underline{\leftrightarrow}_{sl} \langle Q,V\rangle$. To this end, note that it is sufficient to show that the relation

$$\mathscr{R}_{sl} = \{(T,U) | T,U \in \mathscr{P} \text{ and } \forall_{V\in\mathscr{V}} \langle T,V\rangle \text{ and } \langle U,V\rangle \text{ satisfy the same HML}^{check+set} \text{ formulae}\}$$

is a stateless bisimulation relation. Assume that $T\mathscr{R}_{sl}U$ and $\langle T,\overline{V}\rangle \xrightarrow{\lambda} \langle T',V'\rangle$ for some valuation $\overline{V}$. We shall now argue that there is a process U' such that $\langle U,\overline{V}\rangle \xrightarrow{\lambda} \langle U',V'\rangle$ and $T'\mathscr{R}_{sl}U'$. Since $\mathscr{R}_{sl}$ is symmetric, this suffices to establish that $\mathscr{R}_{sl}$ is a stateless bisimulation relation.

Now assume, towards a contradiction, that there is no $\langle U',V'\rangle$ such that $\langle U,\overline{V}\rangle \xrightarrow{\lambda} \langle U',V'\rangle$ and for all valuations $V \in \mathscr{V}$, $\langle U',V\rangle$ satisfies the same HML$^{check+set}$ formulas as $\langle T',V\rangle$. Since $\langle U,\overline{V}\rangle$ is image finite, the set of processes that $\langle U,\overline{V}\rangle$ can reach by performing a λ-labelled transition is finite, say $\{\langle U_1,V_1\rangle,\ldots,\langle U_n,V_n\rangle\}$ with $n \in \mathbb{N}$. For every $i \in \{1\ldots n\}$, there exist a formula ϕ_i and valuation V_i' such that $\langle T',V_i'\rangle \in [\![\phi_i]\!]$ and $\langle U_i,V_i'\rangle \notin [\![\phi_i]\!]$ or valuations V_i and V' differ for variable v.

We are now in a position to construct a formula that is satisfied by $\langle T,\overline{V}\rangle$ but not by $\langle U,\overline{V}\rangle$, contradicting our assumption that $\langle T,\overline{V}\rangle$ and $\langle U,\overline{V}\rangle$ satisfy the same formulae.

$$\text{We define for each } i \in \{1\ldots n\} : refute(i) = \begin{cases} \downarrow(V_i')\phi_i & \text{if } \langle T',V_i'\rangle \in [\![\phi_i]\!] \text{ and } \langle U_i,V_i'\rangle \notin [\![\phi_i]\!] \\ (v = V'(v)) & \text{if the valuations of } V' \text{ and } V_i' \text{ differ for } v \end{cases}$$

The formula $\langle\lambda\rangle(refute(1) \wedge refute(2) \wedge \cdots \wedge refute(n))$ is satisfied by $\langle T,\overline{V}\rangle$ but not by $\langle U,\overline{V}\rangle$.

□

Theorem 12. Let $\langle P,V\rangle$ and $\langle Q,V\rangle$ be states in some LTS $(S,TL,\rightarrow,s)$ and let all states reachable from $\langle P,V\rangle$ and $\langle Q,V\rangle$ be image-finite. Then $\langle P,V\rangle \underline{\leftrightarrow}_{sb} \langle Q,V\rangle$ if and only if for all HML^{check} formulas ϕ we have that $\langle P,V\rangle \in [\![\phi]\!] \Leftrightarrow \langle Q,V\rangle \in [\![\phi]\!]$.

Proof. The proof is very similar to the one given for Theorem 11. We will only provide the distinguishing formula.

Let $\langle T,\overline{V}\rangle \xrightarrow{\lambda} \langle T',V'\rangle$ and let $\{\langle U_1,V_1\rangle,\ldots,\langle U_n,V_n\rangle\}$ be the set of states $\langle U,\overline{V}\rangle$ can reach with a λ-labelled transition. For every $i \in \{1\ldots n\}$, there exists a formula ϕ_i such that $\langle T',V'\rangle \in [\![\phi_i]\!]$ and $\langle U_i,V_i\rangle \notin [\![\phi_i]\!]$ or valuations V_i and V' differ for variable v.

$$\text{We define for each } i \in \{1\ldots n\} : refute(i) = \begin{cases} \phi_i & \text{if } \langle T',V'\rangle \in [\![\phi_i]\!] \text{ and } \langle U_i,V_i\rangle \notin [\![\phi_i]\!] \\ (v = V'(v)) & \text{if the valuations of } V' \text{ and } V_i \text{ differ for } v \end{cases}$$

The formula $\langle\lambda\rangle(refute(1) \wedge refute(2) \wedge \cdots \wedge refute(n))$ is satisfied by $\langle T,\overline{V}\rangle$ but not by $\langle U,\overline{V}\rangle$. □

Note that for process expressions we need the set operator in the logic, whereas for states we can only have the check operator on top of regular HML. Intuitively, we need the set operator on the level of process expressions to say something about the behaviour of the process for any valuation.

6 Translation to mCRL2

For process algebras without global variables it might be the case that global variables can be modelled using different language constructs. Modelling global variables then often requires a protocol to regulate the access to global variables. In this section we explore how a process expression in our process

algebra with global variables can be translated to mCRL2 without introducing extra internal activity. The resulting mCRL2 specification induces an LTS that is isomorphic to the LTS of the original process expression, save some selfloops signalling information on the valuation of that state. We also give a translation from HML^{check} to the modal μ-calculus, which is the logic that is used in mCRL2 to express properties. We show that a HML^{check} formula holds for the original process expression if and only if the translated formula holds for the translated process expression.

6.1 Introduction of mCRL2

We will introduce the syntax and semantics of the fragment of mCRL2 that is needed to encode global variables. In particular, we will introduce actions parametrised with data and multi-actions. We will not go into the details of the data language itself. It suffices that there exists a semantic interpretation function $[\![.]\!]$ that maps data expressions to elements of the data domain. We declare a set of data expressions $\mathscr{D}$ and a set of Boolean expressions $\mathscr{B}$ of which the interpretation is an element of D or $\{true, false\}$, respectively. We also presuppose an equality relation $\approx$ on data expressions. For more information on the syntax and semantics of mCRL2 we refer the reader to [6]. Multi-actions in combination with the allow operator were first proposed in [13].

We presuppose a set of *action names* $\underline{\Lambda}$, each with an associated arity. An *action label* $a(d_1,\ldots,d_n)$ consists of an action name $a \in \underline{\Lambda}$ of arity n and a list of data parameters $d_1,\ldots,d_n$. We denote by Λ the set of action labels. If $\alpha \in \Lambda$, then we denote by $\underline{\alpha}$ its name (e.g., $\underline{a(2,3,true)} = a$).

The set of multi-actions $\mathbb{M}$ is generated by the following grammar:

$$\alpha := \alpha|\alpha \mid \tau \mid a(d_1,\ldots,d_n),$$

where $a(d_1,\ldots,d_n) \in \Lambda$ and d_1 to d_n are data expressions or Boolean expressions. Also for each multi-action α we define $\underline{\alpha}$:

$$\underline{\tau} = \tau$$

$$\underline{a(d_1,...,d_n)} = a$$

$$\underline{\alpha|\beta} = \underline{\alpha}|\underline{\beta}.$$

The set of multi-actions where each action label in the multi-action is in $\underline{\Lambda}$ is $\underline{\mathbb{M}}$.

We define a multi-set over A, (A,m), where $m : A \to \mathbb{N}$ is a function assigning a multiplicity to each element of A. We define $a \in (A,m)$ to be true if and only if $m(a) > 0$. As notation we use $\wr\,\wr$ where the elements are listed together with their multiplicity, e.g. $\wr a : 2, b : 3\wr$. Over multi-sets (A,m) and (A,m') we define a binary operator *addition*, denoted by $+$, that results in a multi-set (A,m''), where for all a in A it is the case that $m''(a) = m(a) + m'(a)$. Similarly, we define a binary operator *subtraction*, denoted by $-$, such that it results in a multi-set (A,m''), where for all $a \in A$ we have that $m''(a) = max(m(a) - m'(a), 0)$. Furthermore, we define inclusion, denoted by $\subseteq$, to hold if and only if for all $a \in A$ we have that $m(a) \leq m'(a)$. For multi-sets over labels we define $\underline{(\Lambda,m)} = (\underline{\Lambda},m')$, where for all $a \in \Lambda$ it holds that $m(a) = m'(\underline{a})$.

Given a multi-action α we inductively associate a *semantic multi-action* $[\![\alpha]\!]$ with it:

$$[\![\tau]\!] = \wr\,\wr$$

$$[\![a(d_1,...,d_n)]\!] = \wr a([\![d_1]\!],...,[\![d_n]\!]) : 1\wr$$

$$[\![\alpha|\beta]\!] = [\![\alpha]\!] + [\![\beta]\!]$$

The set of all semantic multi-actions is $\mathscr{M}$. The set $\mathscr{P}_{mCRL2}$ of process expressions of the fragment of mCRL2 that we need in the translation is generated by the following grammar:

$$P := \lambda.P \mid \delta \mid P+P \mid P\|P \mid \nabla_M(P) \mid X(d_1,...,d_n) \mid \sum_{d:D} P \mid \tau_I(P) \mid \Gamma_C(P),$$

where $\lambda \in \mathbb{M}$, M a set of multi-action names, $M \subseteq \underline{\mathbb{M}}$, X is a process name, I is a set of action names, $I \subseteq \underline{\Lambda}$, and C is set of renamings from a set of multi-action names to an action name, notation $a|...|b \to c$.

We introduce a function $\gamma_C(\alpha)$, where α is a semantic multi-action that applies communications in C to α, e.g. $\Gamma_{\{a|b\to c\}}\{\!|a:2,b:3|\!\} = \{\!|b:1,c:2|\!\}$. A communication can only be performed when the parameters of action labels match. For the exact semantics of $\gamma_C(\alpha)$ we refer the reader to [6].

We define a function $\theta_I((\Lambda,m))$, such that it results in a multi-set (Λ,m') such that

$$\forall_{a\in\Lambda} m'(a) = \begin{cases} 0 & \text{if } \underline{a} \in I \\ m(a) & \text{if } \underline{a} \notin I \end{cases}$$

The sum operator facilitates a non-deterministic choice over a data domain. For example, in the case the data domain D is the natural numbers, $\sum_{n:D} a(n).P$ can make an $a(0)$ step to process $P[n := 0]$, an $a(1)$ step to process $P[n := 1]$, etcetera.

$$\text{(PREF)}\ \frac{}{\alpha.P \xrightarrow{[\![\alpha]\!]} P} \qquad \text{(PAR)}\ \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\beta} Q'}{P\|Q \xrightarrow{\alpha+\beta} P'\|Q'}$$

$$\text{(REC)}\ \frac{P[d_1 := t_,...,d_n := t_n] \xrightarrow{\alpha} P' \quad X(d_1:D_1,...,d_n:D_n) \stackrel{\text{def}}{=} P}{X(t_1,...,t_n) \xrightarrow{\alpha} P'}$$

$$\text{(SUM-L)}\ \frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'} \qquad \text{(SUM-R)}\ \frac{Q \xrightarrow{\alpha} Q'}{P+Q \xrightarrow{\alpha} Q'}$$

$$\text{(SUM)}\ \frac{P[d := t_e] \xrightarrow{\alpha} P' \quad t_e \in \mathscr{D}}{\sum_{d:D} P \xrightarrow{\alpha} P'} \qquad \text{(HIDE)}\ \frac{P \xrightarrow{\alpha} P'}{\tau_I(P) \xrightarrow{\theta_I(\alpha)} \tau_I(P')}$$

$$\text{(PAR-L)}\ \frac{P \xrightarrow{\alpha} P'}{P\|Q \xrightarrow{\alpha} P'\|Q} \qquad \text{(PAR-R)}\ \frac{Q \xrightarrow{\alpha} Q'}{P\|Q \xrightarrow{\alpha} P\|Q'}$$

$$\text{(COMM)}\ \frac{P \xrightarrow{\alpha} P'}{\Gamma_C(P) \xrightarrow{\gamma_C(\alpha)} \Gamma_C(P')} \qquad \text{(ALLOW)}\ \frac{P \xrightarrow{\alpha} P'}{\nabla_M(P) \xrightarrow{\alpha} \nabla_M(P')}\ [\![\underline{\alpha}]\!] \in M$$

Table 2: Structural operational semantics of our fragment of mCRL2.

An LTS $(S,\mathscr{M},\to,P)$ can be associated to a process expression P. The set of states is the set of process expressions, $S = \mathscr{P}_{mCRL2}$. The set of transitions is generated by the proof system based on the structural operation semantics (see Table 2).

6.2 Translation of process expressions and valuations

Recall that a specification in the process algebra with global variables consists of the following: a data domain D, a set of variable names Var, a set of action labels Act, a set of process names PN and their defining equations, a communication function γ, and an initial state consisting of a process expression and an initial valuation V. We consider a restricted grammar for the translation.

Sequential components The set of *sequential process expressions* $\mathscr{P}_{Seq}$ is generated by the following grammar (with v ranging over Var, d ranging over D, X ranging over PN and λ ranging over TL):

$$Seq := Seq + Seq \mid (v = d) \to Seq \mid \lambda.Seq \mid \delta \mid \lambda.X.$$

By a *sequential recursive specification* E we mean a set of defining equations $X \stackrel{\text{def}}{=} t$, with t a sequential process expression, including precisely one such equation for every $X \in PN$.

Parallel-sequential processes Presupposing a sequential recursive specification E, the set of *parallel-sequential* process expressions $\mathscr{P}_{Par}$ over E is generated by the following grammar (with X ranging over PN and Seq ranging over sequential process expressions):

$$Par := Par \| Par \mid X \mid Seq.$$

We assume that the recursive specification E is sequential and that the process expression under consideration for translation is of the shape $\partial_B(P)$, where P is parallel-sequential process expression and $B \subseteq Act$. For the sake of readability, in our explanations below we restrict our attention to the case that there is one global variable g. In Section 7 we explain how to generalise the translation to any number of variables. Now that the input for the translation is clear, we show how it is translated to mCRL2.
The value of global variables is tracked by a dedicated process $Globs$, defined below.
$Globs(d : D) =$
$\quad checkG(d,true).Globs(d)$
$\quad + checkG(d,true)|checkG(d,true).Globs(d)$
$\quad + \Sigma_{new:D}.checkG(d,true)|assignG(g,new).Globs(new)$
$\quad + value(g,d).Globs(d)$;
The process can communicate the current value of the global variable with a `checkG` action, of which the first parameter is of type D and the second a constant of type $Bool$. It can perform a `checkG` action twice in a multi-action to facilitate informing two parallel processes in one step. Since our process algebra with global variables only allows handshaking communication there can never be more than two parallel processes that participate in a transition. It facilitates changing the value of the global variable with an `assignG` action, with one parameter of type D carrying the new value. It can emit the current value of a global variable with a `value` action, with a single parameter of type D.

We translate the recursive specification E to a recursive mCRL2 specification E', which includes defining equations for all the process names in PN and additionally a defining equation for $Globs$. Let $\wp(A)$ denote the powerset of A. We introduce a function $\chi : \mathscr{P}_{Seq} \times \wp(D) \to \mathscr{P}_{mCRL2}$, which we will define shortly. For every defining equation $X \stackrel{\text{def}}{=} t$ in E there is a defining equation $X \stackrel{\text{def}}{=} \chi(t,\emptyset)$ in E'. The function χ is defined below, where $\varepsilon \subseteq D$ is a set of constraints on the global variable that is eventually transformed into an appropriate $checkP$ action.

$$\begin{array}{lcl}
\chi(P_1+P_2,\varepsilon) & = & \chi(P_1,\varepsilon)+\chi(P_2,\varepsilon)\\
\chi((g=d)\to P_1,\varepsilon) & = & \chi(P_1,\varepsilon\cup\{d\})\\
\chi(a.P_1,\varepsilon) & = & (\sum_{d_1:D} a|checkP(d_1,\bigwedge_{d\in\varepsilon} d_1\approx d)).\chi(P_1,\emptyset)\\
\chi(assign(g,d').P_1,\varepsilon) & = & (\sum_{d_1:D} assignP(g,d')|checkP(d_1,\bigwedge_{d\in\varepsilon} d_1\approx d)).\chi(P_1,\emptyset)\\
\chi(X,\varepsilon) & = & X\\
\chi(\delta,\varepsilon) & = & \delta
\end{array}$$

We define a set of communications C_γ, such that $a|b\to c\in C_\gamma$ or $b|a\to c\in C_\gamma$ if and only if $\gamma(a,b)=c$ (we should include only one of $a|b\to c$ and $b|a\to c$ in C_γ to satisfy the requirement that the left-hand sides of communications in C_γ are disjoint). We define an extended set of communications that includes communications with *Globs*: $C_{G\gamma}=C_\gamma\cup\{assignP|assignG\to assign, checkP|checkG\to check\}$. Given a set of encapsulated actions B we define a set of allowed actions $A_B=(Act\setminus B)\cup\{value,assign\}$. We extend χ to parallel-sequential process expressions in the following way.

$$\chi(P_1||P_2,\emptyset) \quad = \quad \chi(P_1,\emptyset)||\chi(P_2,\emptyset)$$

We translate the process expression $\partial_B(P)$, with an initial valuation $V, V(g)=d$, to the mCRL2 process expression $\nabla_{A_B}(\tau_{\{check\}}(\Gamma_{C_{G\gamma}}(\chi(P,\emptyset)||Globs(d))))$, which we abbreviate to $\Psi(P,V)$.

6.3 Translation of formula

The selfloops labelled with *value* provide information on the values of global variables in every state, which we will exploit in the translation of HML^{check} formulas. Given a HML^{check} formula we eliminate each occurrence of the check operator of the shape $(v=e)$ by substituting it with $\langle value(v,e)\rangle true$. We denote this substitution function with θ, which we define inductively:

$$\begin{array}{lcl}
\theta(true) & = & true,\\
\theta(false) & = & false,\\
\theta(v=e) & = & \langle value(v,e)\rangle true,\\
\theta(\neg\phi) & = & \neg\theta(\phi),\\
\theta(\phi_1\wedge\phi_2) & = & \theta(\phi_1)\wedge\theta(\phi_2),\\
\theta(\phi_1\vee\phi_2) & = & \theta(\phi_1)\vee\theta(\phi_2),\\
\theta(\langle T\rangle\phi) & = & \langle T\rangle\theta(\phi),\\
\theta([T]\phi) & = & [T]\theta(\phi).
\end{array}$$

6.4 Correctness of translation

From here on, when we consider the translation of some state $\langle P,V\rangle$ to $\Psi(P,V)$ we assume that the context of the process expression, such as the data domain D, the set of actions and a recursive specification have been encoded in mCRL2 as described in the previous section.

We will prove that a HML^{check} formula ϕ holds in a state $\langle P,V\rangle$ if and only if $\theta(\phi)$ holds for $\Psi(P,V)$. To achieve this we use a stepping stone. In Definition 13 we define a relation between LTSs with and without a valuation function in the state, called *variable consistency*. We prove that the LTSs induced by $\langle P,V\rangle$ and $\Psi(P,V)$ are variable consistent, which we use in Theorem 23 to prove that any HML^{check} formula ϕ holds for $\langle P,V\rangle$ if and only if $\theta(\phi)$ holds for $\Psi(P,V)$

Definition 13. Let $\mathscr{L}_1=(S_1,TL_1,\to_1,s_1)$ be an LTS such that $S_1=\mathscr{P}\times\mathscr{V}$, and let $\mathscr{L}_2=(S_2,TL_2,\to_2,s_2)$ be an LTS such that $S_2=\mathscr{P}_{mCRL2}$. We say that $\mathscr{L}_2$ is *variable-consistent* with $\mathscr{L}_1$ if there exists a mapping $\ell:S_1\to S_2$ such that whenever some state s_1' is reachable from s_1 in $\mathscr{L}_1$, then $\ell(s_1')$ is reachable from $\ell(s_1)$ in $\mathscr{L}_2$ and

1. for all states $s'_1, s'_2 \in S_2$ reachable from s_2 and such that $s'_1 \xrightarrow{\lambda} s'_2$ we have that $\lambda \in TL \cup \{value(v,d) \mid v \in Var \wedge d \in D\}$,
2. for all $\langle P,V\rangle \in S_1, s' \in S_2, v \in Var, d \in D$ we have that $\ell(\langle P,V\rangle) \xrightarrow{value(v,d)} s'$ if and only if $V(v) = d$ and $\ell(\langle P,V\rangle) = s'$,
3. for all $\lambda \in TL_1$ and reachable states $s'_1, s'_2 \in S_1$ we have that $s'_1 \xrightarrow{\lambda} s'_2$ if and only if $\ell(s'_1) \xrightarrow{\lambda} \ell(s'_2)$.

For the first property of variable consistency we prove the following lemma.

Lemma 14. For all parallel-sequential process expressions P, process expression $P' \in \mathscr{P}_{mCRL2}$, valuations V, $\alpha \in \mathbb{M}$ and $B \subseteq Act$ we have that $\Psi(P,V) \xrightarrow{\alpha} P'$ implies $\alpha \in TL \cup \{value(g,d) \mid d \in D\}$.

Proof. This follows immediately from the allow operator in $\Psi(P,V)$, which does not allow multi-actions that are not in $TL \cup \{value(g,d) \mid d \in D\}$. □

Towards proving the second property of variable consistency we prove the following lemma.

Lemma 15. For all parallel sequential process expressions P, process expression $P' \in \mathscr{P}_{mCRL2}$, valuation V, $d \in D$, $B \subseteq Act$ we have that $\Psi(P,V) \xrightarrow{value(g,d)} P'$ if and only if $V(g) = d$ and $\Psi(P,V) = P'$.

Proof. The process expression $\Psi(P,V)$ contains a parallel component $Globs(d)$. The *Globs* process can make a $value(g,d)$ transition where $V(g) = d$. Moreover, all such $value(g,d)$ transitions are self-loops. Finally, the *Globs* process is the only sub process in $\Psi(P,V)$ that is able to produce a *value* transition. □

Towards proving the third property of variable consistency we first provide a number of auxiliary lemmas.

Lemma 16. For all sequential process expressions P, process expression $P' \in \mathscr{P}_{mCRL2}$, $\lambda \in Act \cup \{assignP(g,d) \mid d \in D\}$, $d_1 \in D$ we have that $\chi(P,\emptyset) \xrightarrow{\lambda|checkP(d_1,true)} P'$ implies that there exists P'' such that $\chi(P'',\emptyset) = P'$.

Proof. From the definition of χ it follows that if $\chi(P,\emptyset)$ can make a $\lambda|checkP(d_1,true)$ labelled transition then there exists some Q and P_1 such that $\chi(P,\emptyset) = Q + (\sum_{d_1:D} \lambda|checkP(d_1, \bigwedge_{d\in\varepsilon} d_1 \approx d)).\chi(P_1,\emptyset)$. Hence after making the $\lambda|checkP(d_1,true)$ labelled transition we end up in $\chi(P_1,\emptyset)$. □

Lemma 17. For all parallel-sequential process expressions P, $\alpha \in \mathbb{M}$ we have that $\Psi(P,V) \xrightarrow{\alpha} P'$ implies that there exists P'' and V' such that $\Psi(P'',V') = P'$.

Proof. By Lemma 14 we conclude that $\alpha \in TL \cup \{value(g,d) \mid d \in D\}$. In the case that α is a *value* transition it is a selfloop and ends in $\Psi(P,V)$. In any other case $\Psi(P,V)$ makes a step that includes a contribution from one or more of the parallel components of P. From the definition of χ it follows that any contribution of a parallel component is of the shape $\lambda|checkP(d_1,b)$, where $d_1 \in D$ and $b \in \{true, false\}$. The *checkP* must communicate with a *checkG*, otherwise the action will be blocked by the allow operator. Hence $b = true$, enabling us to use Lemma 16 to conclude that for every parallel component contributing to α there exists some process expression P_{seq} such that the parallel component ends in $\chi(P_{seq},\emptyset)$. The parallel components of $\Psi(P,V)$ that do not contribute to α remain in a shape such that there exists some process expression P_{seq} such that the parallel component is $\chi(P_{seq},\emptyset)$. The *Globs* process remains unchanged or its valuation is updated, in which case there exists some valuation V' that reflects the updated value. The allow, hide and communication operators remain unchanged. Hence, after any α step $\Psi(P,V)$ ends in a state $\Psi(P'',V')$. □

Lemma 18. For all sequential process expressions P, P', $DD \subseteq \mathscr{D}$, $a \in Act$ and $assign(g,d') \in TL$ we have that $\forall_{V \in \mathscr{V}}(\bigwedge_{d \in DD} V(g) = [\![d]\!]) \implies \langle P,V\rangle \xrightarrow{a} \langle P',V\rangle$ if and only if $\exists_{d_1 \in D} \chi(P,\emptyset) \xrightarrow{a|checkP(d_1,true)} \chi(P',\emptyset) \wedge \bigwedge_{d \in DD} d_1 = [\![d]\!]$ and we have that $\forall_{V \in \mathscr{V}}(\bigwedge_{d \in DD} V(g) = [\![d]\!]) \implies \langle P,V\rangle \xrightarrow{assign(g,d')} \langle P',V[g \mapsto d']\rangle$ if and only if $\exists_{d_1 \in D} \chi(P,\emptyset) \xrightarrow{assignP(g,d')|checkP(d_1,true)} \chi(P',\emptyset) \wedge \bigwedge_{d \in DD} d_1 = [\![d]\!]$.

Proof. This can be proven by induction on the structure of P, the induction hypothesis is that the bi-implication holds for every direct subprocess of P and for every defining equation of process names. The key insight is the second field of the *checkP* action is only true when the condition for the data value in the first field of *checkP*, constructed by χ, is true. □

Lemma 19. For any parallel-sequential process expression P, process expression P', $\lambda \in TL$ and valuations V, V' we have that $\langle \partial_B(P),V\rangle \xrightarrow{\lambda} \langle \partial_B(P'),V'\rangle$ implies P' is again a parallel-sequential process expression.

Proof. Any step made from $\partial_B(P)$ leaves the ∂_B operator and the parallel composition intact. One or more of the parallel components make a step. By the structure of parallel-sequential process expressions these parallel components are either a process name or a sequential process. Since we also assume that the defining equations of every process name is a sequential process expression the process name will make a step as such. By the structure of sequential process expressions they can make a step to a sequential process expression or a process name. Hence, after any step P is again a parallel composition with process names and sequential process expressions. □

Lemma 20. For all valuations V and V', $\lambda \in TL$, parallel-sequential process expressions P, $B \subseteq Act$ we have that $\langle \partial_B(P),V\rangle \xrightarrow{\lambda} \langle \partial_B(P'),V'\rangle$ if and only if $\Psi(P,V) \xrightarrow{\lambda} \Psi(P',V')$.

Proof. Both directions of the bi-implication can be proven with a case distinction on the type of transition using three cases: an action from Act stemming from one of the parallel components, a handshake stemming from two parallel components and an assignment. To prove the implication from left to right Lemma 18 can be used to prove that for each contribution by a parallel component of $\langle \partial_B(P),V\rangle$ the step can be matched with an appropriate step from a parallel component of $\Psi(P,V)$. Lemma 19 ensures that after taking a transition we end in the translation of a parallel-sequential process again. To prove the implication from right to left Lemma 16 and Lemma 18 can be used to prove that for each contribution by a parallel component of $\Psi(P,V)$ the step can be matched with an appropriate step from a parallel component of $\langle \partial_B(P),V\rangle$. □

Theorem 21. For all parallel-sequential process expressions P, valuations V, we have that the LTSs induced by $\langle P,V\rangle$ and $\Psi(P,V)$ are variable consistent.

Proof. For every state $\langle P',V'\rangle$ reachable from $\langle P,V\rangle$ we define $\ell(\langle P',V'\rangle) = \Psi(P',V')$. Lemma 14 proves condition 1, Lemma 15 proves condition 2 and Lemma 20 together with Lemma 17 proves condition 3. □

Corollary 22. For all parallel-sequential process expressions P, Q and valuations V_1, V_2 we have that $\langle P,V_1\rangle \underline{\leftrightarrow}_{sb} \langle Q,V_2\rangle$ if and only if $\Psi(P,V_1) \underline{\leftrightarrow} \Psi(Q,V_2)$.

Proof. This follows immediately from the definition of variable consistency. The difference between state-based bisimilarity and strong bisimilarity is only that state-based bisimilarity requires that the valuation in states is equal. By condition 2 of variable consistency the valuations V_1 and V_2 are equal if and only if $\Psi(P,V_1)$ and $\Psi(Q,V_2)$ have the same *value* labelled self-loops on states. □

Theorem 23. Let $(S,TL,\rightarrow,s)$ be an LTS where $S = \mathscr{P} \times \mathscr{V}$, let $(S',TL',\rightarrow',s')$ be an LTS where $S' = \mathscr{P}_{mCRL2}$ and let these two LTSs be variable consistent. A HML^{check} formula ϕ holds in some state $\langle P,V\rangle \in S$ if and only if $\theta(\phi)$ holds in $\ell(\langle P,V\rangle) \in S'$.

Proof. The proof is by induction on the structure of ϕ with the induction hypothesis that any subformula ϕ' of ϕ holds for $\langle P,V\rangle \in S$ if and only if $\theta(\phi')$ holds for $\ell(\langle P,V\rangle)$. In the case $\phi = (v = e)$ condition 2 of variable consistency is necessary to relate the valuation in a state and the *value* labelled selfloops. Condition 3 of variable consistency is needed for the case $\phi = \langle T\rangle\phi'$ and $\phi = [T]\phi'$ to show that transitions can be mimicked. Furthermore, in the case $\phi = [T]\phi'$ we also need condition 1 of variable consistency to show that $\ell(\langle P,V\rangle)$ does not have more λ labelled transitions. □

7 Discussion

For the encoding in mCRL2 and subsequent correctness proofs we have made the assumption that there is only one global variable, which is rather restrictive. To generalize the translation to handle any number of global variables we would need to adjust the following. The *Globs* process should be adjusted to track more global variables by making the parameter of the process a mapping from variable names to values. Upon performing an $assignG(v,d)$ action *Globs* should update the mapping such that v maps to d. To communicate the values of global variables in a $check(d_1,\ldots,d_n,true)$ action we need an ordering on the global variables: d_1 is given the value of variable one, d_2 is given the value of variable two, etcetera. The condition determining the last parameter of the *checkP* action should also be adjusted to use this ordering, e.g. when χ gathers a requirement (v,d) and variable v is the ith variable then the condition in *checkP* should include a conjunct $d_i \approx d$.

We intend to continue researching process algebras with global variables. One research direction is to extend mCRL2 with global variables. The simple process algebra presented in this paper only allows for very simple conditions on global variables: checking whether a variable has a specific value. If global variables could be integrated into mCRL2 we could use its powerful data language to specify complex conditions. We would also like to research scoped shared variables, including creation and scope extrusion.

8 Conclusion

In this paper we have presented a simple process calculus with global variables and studied various aspects of it. To start we examined appropriate notions of equivalence: stateless bisimulation for process expressions and state-based bisimulation for states. Then, for our first contribution we presented a logic extending HML with a check and a set operator and proved that HML^{check} is strong enough to differentiate states that are not state-based bisimilar and HML$^{check+set}$ is strong enough to differentiate process expressions that are not stateless bisimilar. Finally, for our second contribution we give a translation to mCRL2, using the multi-action concept, preserving HML^{check} formulas. Translating to mCRL2 allows us to reuse the already existing tools. The translation mostly preserves the syntactic structure and, in particular, the parallel composition (adding one extra parallel process).

When analysing whether a distributed system satisfies a liveness property, it is necessary to define through a completeness criterion which runs of the system should be considered in the analysis. Recently, *justness* was proposed as a suitable completeness criterion that takes into account the component structure of the system [5] and excludes unrealistic runs. Modelling shared variables as separate components hampers a straightforward definition of justness [4, 3]. Since global variables need not be modelled as separate components in the process algebra proposed in Section 2, it may facilitate a more elegant analysis of liveness properties under justness assumptions for distributed systems that rely on shared variables for the communication between components.

Acknowledgements

For the presentation of the semantics of mCRL2, and in particular for the semantics of multi-actions, we have benefited from work by Maurice Laveaux. We would also like to extend our gratitude to the anonymous reviewers. Their comments led to Corollary 22.

References

[1] J. C. M. Baeten, T. Basten & M. A. Reniers (2009): *Process Algebra: Equational Theories of Communicating Processes*. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, doi:10.1017/CBO9781139195003.

[2] J.C.M. Baeten (2005): *A brief history of process algebra*. *Theoretical Computer Science* 335(2-3), pp. 131–146, doi:10.1016/j.tcs.2004.07.036.

[3] Mark Bouwman, Bas Luttik & Tim A. C. Willemse (2020): *Off-the-shelf automated analysis of liveness properties for just paths*. *Acta Informatica* 57(3-5), pp. 551–590, doi:10.1007/s00236-020-00371-w.

[4] Victor Dyseryn, Rob J. van Glabbeek & Peter Höfner (2017): *Analysing Mutual Exclusion using Process Algebra with Signals*. In Kirstin Peters & Simone Tini, editors: *Proceedings Combined 24th International Workshop on Expressiveness in Concurrency and 14th Workshop on Structural Operational Semantics and 14th Workshop on Structural Operational Semantics, EXPRESS/SOS 2017, Berlin, Germany, 4th September 2017.*, *EPTCS* 255, pp. 18–34, doi:10.4204/EPTCS.255.2.

[5] Rob J. van Glabbeek & Peter Höfner (2015): *CCS: It's not fair! - Fair schedulers cannot be implemented in CCS-like languages even under progress and certain fairness assumptions*. *Acta Inf.* 52(2-3), pp. 175–205, doi:10.1007/s00236-015-0221-6.

[6] Jan Friso Groote & Mohammad Reza Mousavi (2014): *Modeling and analysis of communicating systems*. MIT press, doi:10.7551/mitpress/9946.001.0001.

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

[8] Robin Milner (1989): *Communication and concurrency*. PHI Series in computer science, Prentice Hall.

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

[10] Rocco De Nicola & Rosario Pugliese (1996): *A Process Algebra Based on LINDA*. In: *COORDINATION*, *Lecture Notes in Computer Science* 1061, Springer, pp. 160–178, doi:10.1007/3-540-61052-9_45.

[11] A. W. Roscoe (2010): *Understanding Concurrent Systems*. Texts in Computer Science, Springer, doi:10.1007/978-1-84882-258-0.

[12] Vijay A. Saraswat, Martin C. Rinard & Prakash Panangaden (1991): *Semantic Foundations of Concurrent Constraint Programming*. In: *POPL*, ACM Press, pp. 333–352, doi:10.1145/99583.99627.

[13] Muck van Weerdenburg (2008): *Process Algebra with Local Communication*. Electron. Notes Theor. Comput. Sci. 215, pp. 191–208, doi:10.1016/j.entcs.2008.06.028.

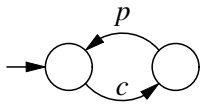
Reactive Temporal Logic

Rob van Glabbeek
Data61, CSIRO, Sydney, Australia
School of Computer Science and Engineering, University of New South Wales, Sydney, Australia
rvg@cs.stanford.edu

Whereas standard treatments of temporal logic are adequate for *closed systems*, having no run-time interactions with their environment, they fall short for *reactive systems*, interacting with their environments through synchronisation of actions. This paper introduces *reactive temporal logic*, a form of temporal logic adapted for the study of reactive systems. I illustrate its use by applying it to formulate definitions of a fair scheduler, and of a correct mutual exclusion protocol. Previous definitions of these concepts were conceptually much more involved or less precise, leading to debates on whether or not a given protocol satisfies the implicit requirements.

1 Introduction

Labelled transition systems are a common model of distributed systems. They consist of sets of states, also called *processes*, and transitions—each transition going from a source state to a target state. A given distributed system $\mathscr{D}$ corresponds to a state P in a transition system $\mathbb{T}$—the initial state of $\mathscr{D}$. The other states of $\mathscr{D}$ are the processes in $\mathbb{T}$ that are reachable from P by following the transitions. The transitions are labelled by *actions*, either visible ones or the invisible action τ. Whereas a τ-labelled transition represents a state-change that can be made spontaneously by the represented system, a-labelled transitions for $a \neq \tau$ merely represent potential activities of $\mathscr{D}$, for they require cooperation from the *environment* in which $\mathscr{D}$ will be running, sometimes identified with the *user* of system $\mathscr{D}$. A typical example is the acceptance of a coin by a vending machine. For this transition to occur, the vending machine should be in a state where it is enabled, i.e., the opening for inserting coins should not be closed off, but also the user of the system should partake by inserting the coin.



Consider a vending machine that alternatingly accepts a coin (c) and produces a pretzel (p). Its labelled transition system is depicted on the right. In standard temporal logic one can express that each action c is followed by p: whenever a coin is inserted, a pretzel will be produced. Aligned with intuition, this formula is valid for the depicted system. However, by symmetry one obtains the validity of a formula saying that each p is followed by a c: whenever a pretzel is produced, eventually a new coin will be inserted. But that clashes with intuition.

In this paper I enrich temporal logic judgements $P \models \varphi$, saying that system P satisfies formula φ, with a third argument B, telling which actions can be blocked by the environment (by failing to act as a synchronisation partner) and which cannot. When stipulating that the coin needs cooperation from a user, but producing the pretzel does not, the two temporal judgements can be distinguished, and only one of them holds. I also introduce a fourth argument CC—a completeness criterion—that incorporates progress, justness and fairness assumptions employed when making a temporal judgement. This yields statements of the form $P \models_B^{CC} \varphi$.

Then I use the so obtained formalism to formalise the correctness requirements of mutual exclusion protocols and of fair schedulers. Making these requirements precise helps in stating negative results on the possibilities to render such protocols in a given setting. In the case of fair schedulers, reactive

O. Dardha and J. Rot (Eds.): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics (EXPRESS/SOS 2020).
EPTCS 322, 2020, pp. 51–68, doi:10.4204/EPTCS.322.6

temporal logic leads to a much easier to understand formalisation than the one in the literature. In the case of mutual exclusion protocols it leads to more precise and less ambiguous requirements, that may help to settle debates on whether or not some formalisation of a mutual exclusion protocol is correct.

2 Kripke Structures and Linear-time Temporal Logic

Definition 1 Let AP be a set of *atomic predicates*. A *Kripke structure* over AP is tuple $(S, \rightarrow, \models)$ with S a set (of *states*), $\rightarrow \subseteq S \times S$, the *transition relation*, and $\models \subseteq S \times AP$. $s \models p$ says that predicate $p \in AP$ *holds* in state $s \in S$.

Here I generalise the standard definition [14] by dropping the condition of *totality*, requiring that for each state $s \in S$ there is a transition $(s, s') \in \rightarrow$. A *path* in a Kripke structure is a nonempty finite or infinite sequence $s_0, s_1, \ldots$ of states, such that $(s_i, s_{i+1}) \in \rightarrow$ for each adjacent pair of states s_i, s_{i+1} in that sequence. A *suffix* π' of a path π is any path obtained from π by removing an initial segment. Write $\pi \Rightarrow \pi'$ if π' is a suffix of π; this relation is reflexive and transitive.

A distributed system $\mathscr{D}$ can be modelled as a state s in a Kripke structure K. A run of $\mathscr{D}$ then corresponds with a path in K starting in s. Whereas each finite path in K starting from s models a *partial run* of $\mathscr{D}$, i.e., an initial segment of a (complete) run, typically not each path models a run. Therefore a Kripke structure constitutes a good model of distributed systems only in combination with a *completeness criterion* [9]: a selection of a set of paths as *complete paths*, modelling runs of the represented system.

The default completeness criterion, implicitly used in almost all work on temporal logic, classifies a path as complete iff it is infinite. In other words, only the infinite paths, and all of them, model (complete) runs of the represented system. This applies when adopting the condition of totality, so that each finite path is a prefix of an infinite path. Naturally, in this setting there is no reason to use the word "complete", as "infinite" will do. As I plan to discuss alternative completeness criteria in Section 4, I will here already refer to paths satisfying a completeness criterion as "complete" rather than "infinite". Moreover, when dropping totality, the default completeness criterion is adapted to declare a path complete iff it either is infinite or ends in a state without outgoing transitions [1].

Linear-time temporal logic (LTL) [23, 14] is a formalism explicitly designed to formulate properties such as the safety and liveness requirements of mutual exclusion protocols. Its syntax is

$$\varphi, \psi ::= p \mid \neg\varphi \mid \varphi \wedge \psi \mid \mathbf{X}\varphi \mid \mathbf{F}\varphi \mid \mathbf{G}\varphi \mid \psi\mathbf{U}\varphi$$

with $p \in AP$ an atomic predicate. The propositional connectives $\Rightarrow$ and $\vee$ can be added as syntactic sugar. It is interpreted on the paths in a Kripke structure. The relation $\models$ between paths and LTL formulae, with $\pi \models \varphi$ saying that the path π *satisfies* the formula φ, or that φ is *valid* on π, is inductively defined by

- $\pi \models p$, with $p \in AP$, iff $s \models p$, where s is the first state of π,
- $\pi \models \neg\varphi$ iff $\pi \not\models \varphi$,
- $\pi \models \varphi \wedge \psi$ iff $\pi \models \varphi$ and $\pi \models \psi$,
- $\pi \models \mathbf{X}\varphi$ iff $\pi' \models \varphi$, where π' is the suffix of π obtained by omitting the first state,
- $\pi \models \mathbf{F}\varphi$ iff $\pi' \models \varphi$ for some suffix π' of π,
- $\pi \models \mathbf{G}\varphi$ iff $\pi' \models \varphi$ for each suffix π' of π, and
- $\pi \models \psi\mathbf{U}\varphi$ iff $\pi' \models \varphi$ for some suffix π' of π, and $\pi'' \models \psi$ for each path $\pi'' \neq \pi'$ with $\pi \Rightarrow \pi'' \Rightarrow \pi'$.

In [18], Lamport argues against the use of the next-state operator **X**, as it is incompatible with abstraction from irrelevant details in system descriptions. Following this advice, I here restrict attention to LTL without the next-state modality, $\text{LTL}_{-\mathbf{X}}$.

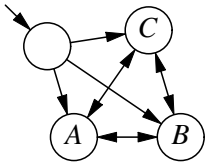
In the standard treatment of LTL [23, 14], judgements $\pi \models \varphi$ are pronounced only for infinite paths π. Here I apply the same definitions verbatim to finite paths as well. At this point I benefit from the exclusion of the next-state operator **X**. In its presence I would have to decide what is the meaning of a judgement $\pi \models \mathbf{X}\varphi$ when π is a path consisting of a single state.[1]

Having given meaning to judgements $\pi \models \varphi$, as a derived concept one defines when an $\text{LTL}_{-\mathbf{X}}$ formula φ holds for a state s in a Kripke structure, modelling a distributed system $\mathscr{D}$, notation $s \models \varphi$ or $\mathscr{D} \models \varphi$. This is the case iff φ holds for all runs of $\mathscr{D}$.

Definition 2 $s \models \varphi$ iff $\pi \models \varphi$ for all complete paths π starting in state s.

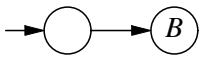
Note that this definition depends on the underlying completeness criterion, telling which paths model actual system runs. In situations where I consider different completeness criteria, I make this explicit by writing $s \models^{CC} \varphi$, with CC the name of the completeness criterion used. When leaving out the superscript CC I here refer to the default completeness criterion, defined above.

Example 1 Alice, Bart and Cameron stand behind a bar, continuously ordering and drinking beer. Assume they do not know each other and order individually. As there is only one barman, they are served sequentially. Also assume that none of them is served twice in a row, but as it takes no longer to drink a beer than to pour it, each if them is ready for next beer as soon as another person is served.

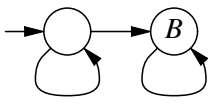


A Kripke structure of this distributed system $\mathscr{D}$ is drawn on the right. The initial state of $\mathscr{D}$ is indicated by a short arrow. The other three states are labelled with the atomic predicates A, B and C, indicating that Alice, Bart or Cameron, respectively, has just acquired a beer. When assuming the default completeness criterion, valid $\text{LTL}_{-\mathbf{X}}$ formulae are $\mathbf{F}(A \vee C)$, saying that eventually either Alice or Cameron will get a beer, or $\mathbf{G}(A \Rightarrow \mathbf{F}\neg A)$, saying that each time Alice got a beer is followed eventually by someone else getting one. However, it is not guaranteed that Bart will ever get a beer: $\mathscr{D} \not\models \mathbf{F}B$. A counterexample for this formula is the infinite run in which Alice and Cameron get a beer alternatingly.

Example 2 Bart is the only customer in a bar in London, with a single barman. He only wants one beer. A Kripke structure of this system $\mathscr{E}$ is drawn on the right. When assuming the default completeness criterion, this time Bart gets his beer: $\mathscr{E} \models \mathbf{F}B$.



Example 3 Bart is the only customer in a bar in London, with a single barman. He only wants one beer. At the same time, Alice and Cameron are in a bar in Tokyo. They drink a lot of beer. Bart is not in contact with Alice and Cameron, nor is there any connection between the two bars. Yet, one may choose to model the drinking in these two bars as a single distributed system. A Kripke structure of this system $\mathscr{F}$ is drawn on the right, collapsing the orders of Alice and Cameron, which can occur before or after Bart gets a beer, into self-loops. When assuming the default completeness criterion, Bart cannot count on a beer: $\mathscr{F} \not\models \mathbf{F}B$.



[1] One possibility would be to declare this judgement to be false, regardless of φ. However, this would invalidate the self-duality of the **X** modality, stating that $\neg\mathbf{X}\varphi$ holds for the same paths as $\mathbf{X}\neg\varphi$.

3 Labelled Transition Systems, Process Algebra and Petri Nets

The most common formalisms in which to present reactive distributed systems are pseudocode, process algebra and Petri nets. The semantics of these formalisms is often given by translation into labelled transition systems (LTSs), and these in turn can be translated into Kripke structures, on which temporal formulae from languages such as LTL are interpreted. These translations make the validity relation $\models$ for temporal formulae applicable to all these formalisms. A state in an LTS, for example, is defined to satisfy an $\mathrm{LTL}_{-\mathbf{X}}$ formula φ iff its translation into a state in a Kripke structure satisfies this formula.

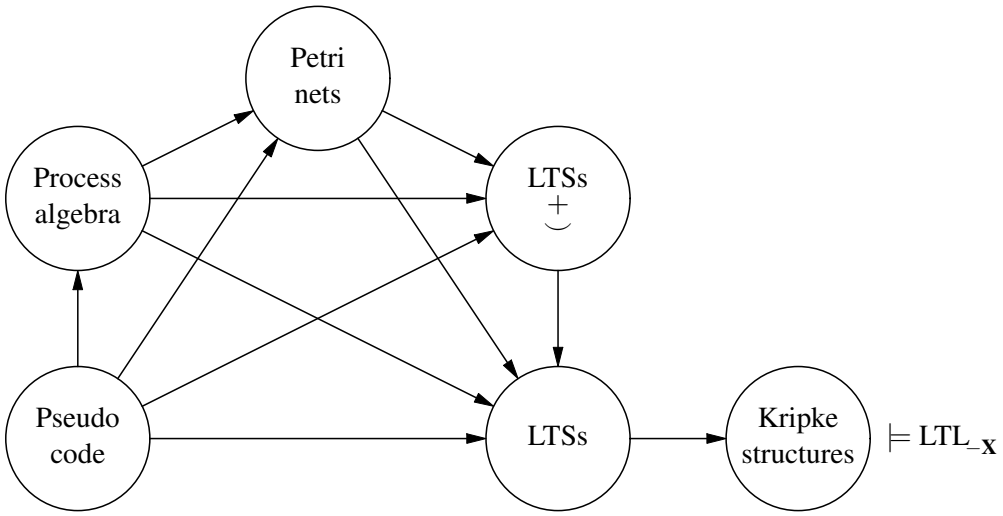


Figure 1: *Formalisms for modelling mutual exclusion protocols*

Figure 1 shows a commuting diagram of semantic translations found in the literature, from pseudocode, process algebra and Petri nets via LTSs to Kripke structures. Each step in the translation abstracts from certain features of the formalism at its source. Some useful requirements on distributed systems can be adequately formalised in process algebra or Petri nets, and informally described for pseudocode, whereas LTSs and Kripke structures have already abstracted from the relevant information. An example will be FS 1 on page 64. I also consider LTSs upgraded with a concurrency relation $\smile$ between transitions; these will be expressive enough to formalise some of these requirements.

3.1 Labelled Transition Systems

Definition 3 Let A be a set of *observable actions*, and let $Act := A \cup \{\tau\}$, with $\tau \notin A$ the *hidden action*. A *labelled transition system* (LTS) over Act is tuple $(\mathbb{P}, Tr, source, target, \ell)$ with $\mathbb{P}$ a set (of *states* or *processes*), Tr a set (of *transitions*), $source, target : Tr \to \mathbb{P}$ and $\ell : Tr \to Act$.

Write $s \xrightarrow{\alpha} s'$ if there exists a transition t with $source(t) = s \in \mathbb{P}$, $\ell(t) = \alpha \in Act$ and $target(t) = s' \in \mathbb{P}$. In this case t *goes* from s to s', and is an *outgoing transition* of s. A *path* in an LTS is a finite or infinite alternating sequence of states and transitions, starting with a state, such that each transition goes from the state before it to the state after it (if any). A *completeness criterion* on an LTS is a set of its paths.

As for Kripke structures, a distributed system $\mathscr{D}$ can be modelled as a state s in an LTS upgraded with a completeness criterion. A (complete) run of $\mathscr{D}$ is then modelled by a complete path starting in s. As for Kripke structures, the default completeness criterion deems a path complete iff it either is infinite or ends

in a *deadlock*, a state without outgoing transitions. An alternative completeness criterion could declare some infinite paths incomplete, saying that they do not model runs that can actually occur, and/or declare some finite paths that do not end in deadlock complete. A complete path π ending in a state models a run of the represented system that follows the path until its last state, and then stays in that state forever, without taking any of its outgoing transitions. A complete path that ends in a transition models a run in which the action represented by this last transition starts occurring but never finishes. It is often assumed that transitions are instantaneous, or at least of finite duration. This assumption is formalised through the adoption of a completeness criterion that holds all paths ending in a transition to be incomplete.

The most prominent translation from LTSs to Kripke structures is from De Nicola & Vaandrager [1]. Its purpose is merely to efficiently lift the validity relation $\models$ from Kripke structures to LTSs. It simply creates a new state halfway along any transition labelled by a visible action, and moves the transition label to that state.

Definition 4 Let $(\mathbb{P}, Tr, source, target, \ell)$ be an LTS over $Act = A \cup \{\tau\}$. The associated Kripke structure $(S, \rightarrow, \models)$ over A is given by

- $S := \mathbb{P} \cup \{t \in Tr \mid \ell(t) \neq \tau\}$,
- $\rightarrow := \{(source(t), t), (t, target(t)) \mid t \in Tr \wedge \ell(t) \neq \tau\} \cup \{(source(t), target(t)) \mid t \in Tr \wedge \ell(t) = \tau\}$
- and $\models := \{(t, \ell(t)) \mid t \in Tr \wedge \ell(t) \neq \tau\}$.

Ignoring paths ending within a τ-transition, which are never deemed complete anyway, this translation yields a bijective correspondence between the paths in an LTS and the path in its associated Kripke structure. Consequently, any completeness criterion on the LTS induces a completeness criterion on the Kripke structure. Hence it is now well-defined when $s \models^{CC} \varphi$, with s a state in an LTS, CC a completeness criterion on this LTS and φ an $\text{LTL}_{-\mathbf{X}}$ formula.

3.2 Petri Nets

Definition 5 A *(labelled) Petri net* over Act is a tuple $N = (S, T, F, M_0, \ell)$ where

- S and T are disjoint sets (of *places* and *transitions*),
- $F : (S \times T \cup T \times S) \rightarrow \mathbb{N}$ (the *flow relation* including *arc weights*) such that $\forall t \in T \exists s \in S.\, F(s,t) > 0$,
- $M_0 : S \rightarrow \mathbb{N}$ (the *initial marking*), and
- $\ell : T \rightarrow Act$ (the *labelling function*).

Petri nets are depicted by drawing the places as circles and the transitions as boxes, containing their label. For $x, y \in S \cup T$ there are $F(x,y)$ arrows (*arcs*) from x to y. When a Petri net represents a distributed system, a global state of this system is given as a *marking*, a multiset of places, depicted by placing $M(s)$ dots (*tokens*) in each place s. The initial state is M_0. The behaviour of a Petri net is defined by the possible moves between markings M and M', which take place when a finite multiset G of transitions *fires*. In that case, each occurrence of a transition t in G consumes $F(s,t)$ tokens from each place s. Naturally, this can happen only if M makes all these tokens available in the first place. Next, each t produces $F(t,s)$ tokens in each s. Definition 7 formalises this notion of behaviour.

A *multiset* over a set X is a function $A : X \rightarrow \mathbb{N}$, i.e. $A \in \mathbb{N}^X$. Object $x \in X$ is an *element of* A iff $A(x) > 0$. A multiset is *empty* iff it has no elements, and *finite* iff the set of its elements in finite. For multisets A and B over X I write $A \leq B$ iff $A(x) \leq B(x)$ for all $x \in X$; $A + B$ denotes the multiset over X with $(A+B)(x) := A(x) + B(x)$, $A - B$ is given by $(A-B)(x) := A(x) \dot{-} B(x) = \max(A(x) - B(x), 0)$, and for $k \in \mathbb{N}$ the multiset $k \cdot A$ is given by $(k \cdot A)(x) := k \cdot A(x)$. With $\{x, x, y\}$ I denote a multiset A with $A(x) = 2$ and $A(y) = 1$, rather than the set $\{x, y\}$ itself.

Definition 6 Let $N=(S,T,F,M_0,\ell)$ be a Petri net and $t\in T$. The multisets ${}^\bullet t,\ t^\bullet : S\to\mathbb{N}$ are given by ${}^\bullet t(s)=F(s,t)$ and $t^\bullet(s)=F(t,s)$ for all $s\in S$. The elements of ${}^\bullet t$ and $t^\bullet$ are called *pre-* and *postplaces* of t, respectively. These functions extend to finite multisets $G:T\to\mathbb{N}$ as usual, by ${}^\bullet G:=\sum_{t\in T}G(t)\cdot{}^\bullet t$ and $G^\bullet:=\sum_{t\in T}G(t)\cdot t^\bullet$.

Definition 7 Let $N=(S,T,F,M_0,\ell)$ be a Petri net, $G\in\mathbb{N}^T$, G non-empty and finite, and $M,M'\in\mathbb{N}^S$. G is a *step* from M to M', written $M\xrightarrow{G}_N M'$, iff ${}^\bullet G\le M$ (G is *enabled*) and $M'=(M-{}^\bullet G)+G^\bullet$.

Write $M_0\twoheadrightarrow_N M$ iff there are transitions $t_i\in T$ and markings $M_i\in\mathbb{N}^S$ for $i=1,\ldots,k$, such that $M_k=M$ and $M_{i-1}\xrightarrow{\{t_i\}}_N M_i$ for $i=1,\ldots,k$. Moreover, $M_0\twoheadrightarrow\xrightarrow{G}$ means that $M_0\twoheadrightarrow_N M\xrightarrow{G}_N M'$ for some M and M'.

Definition 8 ([10]) $N=(S,T,F,M_0,\ell)$ is a *structural conflict net* iff $\forall t,u.(M_0\twoheadrightarrow\xrightarrow{\{t,u\}})\Rightarrow {}^\bullet t\cap{}^\bullet u=\emptyset$.

Here I restrict myself to structural conflict nets, henceforth simply called *nets*, a class of Petri nets containing the *safe* Petri nets that are normally used to give semantics to process algebras.

Given a net $N=(S,T,F,M_0,\ell)$, its associated LTS $(\mathbb{P},Tr,source,target,\ell)$ is given by $\mathbb{P}:=\mathbb{N}^S$, $Tr:=\{(M,t)\in\mathbb{N}^S\times Tr\mid {}^\bullet t\le M\}$, $source(M,t):=M$, $target(M,t):=(M-{}^\bullet t)+t^\bullet$ and $\ell(M,t):=\ell(t)$. The net N maps to the state M_0 in this LTS. A *completeness criterion* on a net is a completeness criterion on its associated LTS. Now $N\models^{CC}\varphi$ is defined to hold iff $M_0\models^{CC}\varphi$ in the associated LTS.

3.3 CCS

CCS [19] is parametrised with sets $\mathscr{K}$ of *agent identifiers* and $\mathscr{A}$ of *names*; each $X\in\mathscr{K}$ comes with a defining equation $X\stackrel{def}{=}P$ with P being a CCS expression as defined below. $Act:=\mathscr{A}\dot\cup\bar{\mathscr{A}}\dot\cup\{\tau\}$ is the set of *actions*, where τ is a special *internal action* and $\bar{\mathscr{A}}:=\{\bar a\mid a\in\mathscr{A}\}$ is the set of *co-names*. Complementation is extended to $\bar{\mathscr{A}}$ by setting $\bar{\bar a}=a$. Below, a ranges over $\mathscr{A}\cup\bar{\mathscr{A}}$, α over Act, and X,Y over $\mathscr{K}$. A *relabelling* is a function $f:\mathscr{A}\to\mathscr{A}$; it extends to Act by $f(\bar a)=\overline{f(a)}$ and $f(\tau):=\tau$. The set $\mathrm{T_{CCS}}$ of CCS expressions or *processes* is the smallest set including:

$\sum_{i\in I}\alpha_i.P_i$	for I an index set, $\alpha_i\in Act$ and $P_i\in\mathrm{T_{CCS}}$	*guarded choice*
$P\mid Q$	for $P,Q\in\mathrm{T_{CCS}}$	*parallel composition*
$P\backslash L$	for $L\subseteq\mathscr{A}$ and $P\in\mathrm{T_{CCS}}$	*restriction*
$P[f]$	for f a relabelling and $P\in\mathrm{T_{CCS}}$	*relabelling*
X	for $X\in\mathscr{K}$	*agent identifier*

The process $\sum_{i\in\{1,2\}}\alpha_i.P_i$ is often written as $\alpha_1.P_1+\alpha_2.P_2$, and $\sum_{i\in\emptyset}\alpha_i.P_i$ as $\mathbf{0}$. The semantics of CCS is given by the transition relation $\to\subseteq\mathrm{T_{CCS}}\times Act\times\mathscr{P}(\mathscr{C})\times\mathrm{T_{CCS}}$, where transitions $P\xrightarrow{\alpha,C}Q$ are derived from the rules of Table 1. Ignoring the labels $C\in\mathscr{P}(\mathscr{C})$ for now, such a transition indicates that process P can perform the action $\alpha\in Act$ and transform into process Q. The process $\sum_{i\in I}\alpha_i.P_i$ performs one of the actions α_j for $j\in I$ and subsequently acts as P_j. The parallel composition $P\mid Q$ executes an action from P, an action from Q, or a synchronisation between complementary actions c and $\bar c$ performed by P and Q, resulting in an internal action τ. The restriction operator $P\backslash L$ inhibits execution of the actions from L and their complements. The relabelling $P[f]$ acts like process P with all labels α replaced by $f(\alpha)$. Finally, the rule for agent identifiers says that an agent X has the same transitions as the body P of its defining equation. The standard version of CCS [19] features a *choice* operator $\sum_{i\in I}P_i$; here I use the fragment of CCS that merely features guarded choice.

The second label of a transition indicates the set of (parallel) *components* involved in executing this transition. The set $\mathscr{C}$ of components is defined as $\{\mathrm{L},\mathrm{R}\}^*$, that is, the set of strings over the indicators Left and Right, with $\varepsilon\in\mathscr{C}$ denoting the empty string and $\mathrm{D}\cdot C:=\{\mathrm{D}\sigma\mid\sigma\in C\}$ for $\mathrm{D}\in\{\mathrm{L},\mathrm{R}\}$ and $C\subseteq\mathscr{C}$.

Table 1: Structural operational semantics of CCS

$$\sum_{i\in I}\alpha_i.P_i \xrightarrow{\alpha_j\,,\,\{\varepsilon\}} P_j \quad (j\in I)$$

$$\frac{P \xrightarrow{\alpha\,,\,C} P'}{P|Q \xrightarrow{\alpha\,,\,\mathrm{L}\cdot C} P'|Q} \qquad \frac{P \xrightarrow{a\,,\,C} P',\ Q \xrightarrow{\bar a\,,\,D} Q'}{P|Q \xrightarrow{\tau\,,\,\mathrm{L}\cdot C\cup \mathrm{R}\cdot D} P'|Q'} \qquad \frac{Q \xrightarrow{\alpha\,,\,D} Q'}{P|Q \xrightarrow{\alpha\,,\,\mathrm{R}\cdot D} P|Q'}$$

$$\frac{P \xrightarrow{\alpha\,,\,C} P'}{P\backslash L \xrightarrow{\alpha\,,\,C} P'\backslash L}\ (\alpha,\bar\alpha\notin L) \qquad \frac{P \xrightarrow{\alpha\,,\,C} P'}{P[f] \xrightarrow{f(\alpha)\,,\,C} P'[f]} \qquad \frac{P \xrightarrow{\alpha\,,\,C} P'}{X \xrightarrow{\alpha\,,\,C} P'}\ (X \stackrel{def}{=} P)$$

Example 4 The CCS process $P := (X|\bar a.\mathbf{0})|\bar a.b.\mathbf{0}$ with $X \stackrel{def}{=} a.X$ has as outgoing transitions $P \xrightarrow{a\,,\,\{\mathrm{LL}\}} P$, $P \xrightarrow{\tau\,,\,\{\mathrm{LL},\mathrm{LR}\}} (X|\mathbf{0})|\bar a.b.\mathbf{0}$, $P \xrightarrow{\bar a\,,\,\{\mathrm{LR}\}} (X|\mathbf{0})|\bar a.b.\mathbf{0}$, $P \xrightarrow{\tau\,,\,\{\mathrm{LL},\mathrm{R}\}} (X|\bar a.\mathbf{0})|b.\mathbf{0}$ and $P \xrightarrow{\bar a\,,\,\{\mathrm{R}\}} (X|\bar a.\mathbf{0})|b.\mathbf{0}$. These components stem from Victor Dyseryn [personal communication] and were introduced in [8]. They were not part of the standard semantics of CCS [19], which can be retrieved by ignoring them.

The LTS of CCS is $(\mathrm{T}, Tr, source, target, \ell)$, with $Tr = \{(P,\alpha,C,Q) \mid P \xrightarrow{\alpha\,,\,C} Q\}$, $\ell(P,\alpha,C,Q) = \alpha$, $source(P,\alpha,C,Q) = P$ and $target(P,\alpha,C,Q) = Q$. Employing this interpretation of CCS, one can pronounce judgements $P \models^{CC} \varphi$ for CCS processes P.

3.4 Labelled Transition Systems with Concurrency

Definition 9 A *labelled transition system with concurrency* (LTSC) is a tuple $(\mathbb{P}, Tr, source, target, \ell, \smile)$ consisting of a LTS $(\mathbb{P}, Tr, source, target, \ell)$ and a *concurrency relation* $\smile \subseteq Tr \times Tr$, such that:

$$t \not\smile t \text{ for all } t \in Tr, \tag{1}$$

$$\text{if } t \in Tr \text{ and } \pi \text{ is a path from } source(t) \text{ to } s \in \mathbb{P} \text{ such that } t \smile v \text{ for all transitions } v \text{ occurring in } \pi, \text{ then there is a } u \in Tr \text{ such that } source(u) = s,\ \ell(u) = \ell(t) \text{ and } t \not\smile u. \tag{2}$$

Informally, $t \smile v$ means that the transition v does not interfere with t, in the sense that it does not affect any resources that are needed by t, so that in a state where t and v are both possible, after doing v one can still do a future variant u of t.

LTSCs were introduced in [9], although there the model is more general on various counts. I do not need this generality in the present paper. In particular, I only need symmetric concurrency relations $\smile$; in [9] $\smile$ is not always symmetric, and denoted $\smile\!\!\!\!\bullet$.

The LTS associated with CCS can be turned into an LTSC by defining $(P,\alpha,C,P') \smile (Q,\beta,D,Q')$ iff $C \cap D = \emptyset$, that is, two transitions are concurrent iff they stem from disjoint sets of components [13, 8].

Example 5 Let the 5 transitions from Example 4 be t, u, v, w and x, respectively. Then $t \not\smile w$ because these transitions share the component LL. Yet $v \smile w$.

The LTS associated with a Petri net can be turned into an LTSC by defining $(M,t) \smile (M',u)$ iff ${}^\bullet t \cap {}^\bullet u = \emptyset$, i.e., the two LTS-transitions stem from net-transitions that have no preplaces in common.

Naturally, an LTSC can be turned into a LTS, and further into a Kripke structure, by forgetting $\smile$.

4 Progress, Justness and Fairness

In this section I define completeness criteria $CC \in \{SF(\mathscr{T}), WF(\mathscr{T}), J, Pr, \top \mid \mathscr{T} \in \mathscr{P}(\mathscr{P}(Tr))\}$ on LTSs $(\mathbb{P}, Tr, source, target, \ell)$, to be used in judgements $P \models^{CC} \varphi$, for $P \in \mathbb{P}$ and φ an $\mathrm{LTL}_{-\mathbf{X}}$ formula. These

criteria are called *strong fairness* (*SF*), *weak fairness* (*SF*), both parametrised with a set $\mathscr{T} \subseteq \mathscr{P}(Tr)$ of *tasks*, *justness* (*J*), *progress* (*Pr*) and the *trivial* completeness criterion ($\top$). Justness is merely defined on LTSCs. I confine myself to criteria that hold finite paths ending within a transition to be incomplete.

Reading Example 1, one could find it unfair that Bart might never get a beer. Strong and weak *fairness* are completeness criteria that postulate that Bart will get a beer, namely by ruling out as incomplete the infinite paths in which he does not. They can be formalised by introducing a set $\mathscr{T}$ of *tasks*, each being a set of transitions (in an LTS or Kripke structure).

Definition 10 ([13]) A task $T \in \mathscr{T}$ is *enabled* in a state s iff s has an outgoing transition from T. It is *perpetually enabled* on a path π iff it is enabled in every state of π. It is *relentlessly enabled* on π, if each suffix of π contains a state in which it is enabled.[2] It *occurs* in π if π contains a transition $t \in T$.

A path π is *weakly fair* if, for every suffix π' of π, each task that is perpetually enabled on π', occurs in π'. It is *strongly fair* if, for every suffix π' of π, each task that is relentlessly enabled on π', occurs in π'.

As completeness criteria, these notions take only the fair paths to be complete. In Example 1 it suffices to have a task "Bart gets a beer", consisting of the three transitions leading to the B state. Now in any path in which Bart never gets a beer this task is perpetually enabled, yet never taken. Hence weak fairness suffices to rule out such paths. We have $\mathscr{D} \models^{WF(\mathscr{T})} \mathbf{F}B$.

Local fairness [13] allows the tasks $\mathscr{T}$ to be declared on an ad hoc basis for the application at hand. On this basis one can call it unfair if Bart doesn't get a beer, without requiring that Cameron should get a beer as well. *Global fairness*, on the other hand, distils the tasks of an LTS in a systematic way out of the structure of a formalism, such as pseudocode, process algebra or Petri nets, that gave rise to the LTS. A classification of many ways to do this, and thus of many notions of strong and weak fairness, appears in [13]. In *fairness of directions* [5], for instance, each transition in an LTS is assumed to stem from a particular *direction*, or *instruction*, in the pseudocode that generated the LTS; now each direction represents a task, consisting of all transitions derived from that direction.

In [13] the assumption that a system will never stop when there are transitions to proceed is called *progress*. In Example 2 it takes a progress assumption to conclude that Bart will get his beer. Progress fits the default completeness criterion introduced before, i.e., $\models^{Pr}$ is the same as $\models$. Not (even) assuming progress can be formalised by the trivial completeness criterion $\top$ that declares all paths to be complete. Naturally, $\mathscr{E} \not\models^{\top} \mathbf{F}B$.

Completeness criterion D is called *stronger* than criterion C if it rules out more paths as incomplete. So $\top$ is the weakest of all criteria, and, for any given collection $\mathscr{T}$, strong fairness is stronger than weak fairness. When assuming that each transition occurs in at least one task—which can be ensured by incorporating a default task consisting of all transitions—progress is weaker than weak fairness.

Justness [13] is a strong form of progress, defined on LTSCs.

Definition 11 A path π is *just* if for each transition t with its source state $s := source(t)$ occurring on π, the suffix of π starting at s contains a transition u with $t \not\frown u$.

Example 6 The infinite path π that only ever takes transition t in Example 4/5 is unjust. Namely with transition v in the rôle of the t from Definition 11, π contains no transition y with $v \not\frown y$.

Informally, the only reason for an enabled transition not to occur, is that one of its resources is eventually used for some other transition. In Example 3 for instance, the orders of Alice and Cameron are clearly concurrent with the one of Bart, in the sense that they do not compete for shared resources. Taking t to be the transition in which Bart gets his beer, any path in which t does not occur is unjust. Thus $\mathscr{F} \models^{J} \mathbf{F}B$.

[2] This is the case if the task is enabled in infinitely many states of π, in a state that occurs infinitely often in π, or in the last state of a finite π.

For most choices of $\mathscr{T}$ found in the literature, weak fairness is a strictly stronger completeness criterion than justness. In Example 1, for instance, the path in which Bart does not get a beer is just. Namely, any transition u giving Alice or Cameron a beer competes for the same resource as the transition t giving Bart a beer, namely the attention of the barman. Thus $t \not\frown u$, and consequently $\mathscr{D} \not\models^J \mathbf{F}B$.

5 Reactive Temporal Logic

Standard treatments of temporal logic [23, 14] are adequate for *closed systems*, having no run-time interactions with their environment. However, they fall short for *reactive systems*, interacting with their environments through synchronisation of actions.

Example 7 Consider a vending machine that accepts a coin c and produces a pretzel p. We assume that accepting the coin requires cooperation from the user/environment, but producing the pretzel does not. A CCS specification is

$$VM = c.p.VM\ .$$

In standard LTL$_{-\mathbf{X}}$ (assuming progress) we have $VM \models \mathbf{G}(c \Rightarrow \mathbf{F}p)$. This formula says that whenever a coin is inserted, eventually a pretzel is produced. This formula is intuitively true indeed. But we also have $VM \models \mathbf{G}(p \Rightarrow \mathbf{F}c)$. This formula says that whenever a pretzel is produced, eventually a new coin will be inserted. This formula is intuitively false. This example shows that standard LTL$_{-\mathbf{X}}$ is not suitable to correctly describe the behaviour of this vending machine.

For this reason I here introduce *reactive* LTL$_{-\mathbf{X}}$. The syntax and semantics are unchanged, except that I use a validity relation $\models_B$ that is parametrised with a set $B \subseteq A$ of *blockable* actions. Here A is the set of all observable actions of the LTS on which LTL$_{-\mathbf{X}}$ is interpreted. The intuition is that actions $b \in B$ may be blocked by the environment, but actions $a \in A \backslash B$ may not. The relation $\models_B$ can be used to formalise the assumption that the actions in $A \backslash B$ are not under the control of the user of the modelled system, or that there is an agreement with the user not to block them. Either way, it is a disclaimer on the wrapping of our temporal judgement, that it is valid only when applying the involved distributed system in an environment that may block actions from B only. The hidden action τ may never be blocked.

The subscript B modifies the default completeness criterion, to call a path complete iff it is either infinite or ends in a state of which all outgoing transitions have a label from B. Note that the standard LTL$_{-\mathbf{X}}$ interpretation $\models$ is simply $\models_\emptyset$, obtained by taking the empty set of blocking actions.

In Example 7 one takes $B = \{c\}$. This choice of B says that the environment may block the action c, namely by not inserting a coin; however, the environment may not block p. As intuitively expected, we have $VM \models_B \mathbf{G}(c \Rightarrow \mathbf{F}p)$ but $VM \not\models_B \mathbf{G}(p \Rightarrow \mathbf{F}c)$.

Naturally, reactive LTL$_{-\mathbf{X}}$ can also be combined with a non-default completeness criterion, as discussed in Sections 2–4. When writing $P \models_B^{CC} \varphi$ the modifier B adapts the default completeness criterion by declaring certain finite paths complete, and the modifier $CC \neq \top$ adapts it by declaring some infinite paths incomplete. In the presence of the modifier B, Definition 11 and the first sentence of Definition 10 are adapted as follows:

Definition 12 A path π is *just* (or *B-just*) if for each transition $t \in Tr$ with $\ell(t) \notin B$ and its source state $s := source(t)$ occurring on π, the suffix of π starting at s contains a transition u with $t \not\frown u$.

Note that it doesn't matter whether $\ell(u) \in B$ or not.

Definition 13 A task $T \in \mathscr{T}$ is *enabled* in a state s iff s has an outgoing transition $t \in T$ with $\ell(t) \notin B$.

The above completes the formal definition of the validity of temporal judgements $P \models_B^{CC} \varphi$ with φ an $\mathrm{LTL}_{-\mathbf{X}}$ formula, $B \subseteq A$, and either

- $CC = Pr$ and P a state in an LTS, a CCS expression or a Petri net,
- $CC = J$ and P a state in an LTSC, a CCS expression or a Petri net,
- $CC = WF(\mathscr{T})$ or $SF(\mathscr{T})$ and P a state in an LTS $(\mathbb{P}, Tr, source, target, \ell)$ with $\mathscr{T} \in \mathscr{P}(\mathscr{P}(Tr))$, or P a CCS expression or Petri net with associated LTS $(\mathbb{P}, Tr, source, target, \ell)$ and $\mathscr{T} \in \mathscr{P}(\mathscr{P}(Tr))$.

Namely, in case P is a state in an LTS, it is also a state in the associated Kripke structure K. Moreover, B and CC combine into a single completeness criterion BC on that LTS, which translates as a completeness criterion BC on K. Now Definition 2 tells whether $P \models^{BC} \varphi$ holds.

In case $CC = J$ and P a state in an LTSC, B and J combine into a single completeness criterion BJ on that LTSC, which is also a completeness criterion on the associated LTS; now proceed as above.

In case P is a Petri net or CCS expression, first translate it into a state in an LTS or LTSC, using the translations at the end of Sections 3.2 or 3.3, respectively, and proceed as above.

Temporal judgements $P \models_B^{CC} \varphi$, as introduced above, are not limited to the case that φ is an LTL formula. In Section 10 I will show that allowing φ to be a CTL formula instead poses no additional complications, and I expect the same to hold for other temporal logics.

Judgements $P \models_B^{CC} \varphi$ get stronger (= less likely true) when the completeness criterion CC is weaker, and the set B of blockable actions larger.

Most concepts of reactive temporal logic introduced above stem from [12]. The main novelty contributed here is the annotated satisfaction relation $\models_B^{CC}$. In [12] we simply wrote $\models$, expecting CC and B to be determined once and for all in a given paper or application. Requirement specifications in which different values for B are combined, such as FS 1–2 in Section 8, were not foreseen there.

6 The Mutual Exclusion Problem and its History

The mutual exclusion problem was presented by Dijkstra in [3] and formulated as follows:

> "To begin, consider N computers, each engaged in a process which, for our aims, can be regarded as cyclic. In each of the cycles a so-called "critical section" occurs and the computers have to be programmed in such a way that at any moment only one of these N cyclic processes is in its critical section. In order to effectuate this mutual exclusion of critical-section execution the computers can communicate with each other via a common store. Writing a word into or nondestructively reading a word from this store are undividable operations; i.e., when two or more computers try to communicate (either for reading or for writing) simultaneously with the same common location, these communications will take place one after the other, but in an unknown order."

Dijkstra proceeds to formulate a number of requirements that a solution to this problem must satisfy, and then presents a solution that satisfies those requirements. The most central of these are:

- (*Safety*) "no two computers can be in their critical section simultaneously", and
- (*Dijkstra's Liveness*) If at least one computer intends to enter its critical section, then at least one "will be allowed to enter its critical section in due time".

Two other important requirements formulated by Dijkstra are

- (*Speed independence*) "(b) Nothing may be assumed about the relative speeds of the N computers",
- and (*Optionality*) "(c) If any of the computers is stopped well outside its critical section, this is not allowed to lead to potential blocking of the others."

A crucial assumption is that each computer, in each cycle, spends only a finite amount of time in its critical section. This is necessary for the correctness of any mutual exclusion protocol.

For the purpose of the last requirement one can partition each cycle into a *critical section*, a *noncritical section* (in which the process starts), an *entry protocol* between the noncritical and the critical section, during which a process prepares for entry in negotiation with the competing processes, and an *exit protocol*, that comes right after the critical section and before return to the noncritical section. Now "well outside its critical section" means in the noncritical section. Requirement (c) can equivalently be stated as admitting the possibility that a process chooses to remain forever in its noncritical section, without applying for entry in the critical section ever again.

Knuth [16] proposes a strengthening of Dijkstra's liveness requirement, namely

- (*Liveness*) If a computer intends to enter its critical section, then it will be allowed to enter in due time.

He also presents a solution that is shown to satisfy this requirement, as well as Dijkstra's requirements.[3] Henceforth I define a correct solution of the mutual exclusion problem as one that satisfies both safety and liveness, as formulated above, as well as optionality. I sometimes speak of "speed independent mutual exclusion" when also insisting on requirement (b) above.

The special case of the mutual exclusion problem for two processes ($N = 2$) was presented by Dijkstra in [2], three years prior to [3]. There Dijkstra presented a solution found by T.J. Dekker in 1959, and shows that it satisfies all requirements of [3]. Although not explicitly stated in [2], the arguments given therein imply straightforwardly that Dekker's solution also satisfy the liveness requirement above.

Peterson [22] presented a considerable simplification of Dekker's algorithm that satisfies the same correctness requirements. Many other mutual exclusion protocols appear in the literature, the most prominent being Lamport's bakery algorithm [17] and Szymański's mutual exclusion algorithm [24]. These guarantee some additional correctness criteria besides the ones discussed above.

7 Fair Schedulers

In [11] a *fair scheduler* is defined as

FS 1 "a reactive system with two input channels: one on which it can receive requests r_1 from its environment and one on which it can receive requests r_2. We allow the scheduler to be too busy shortly after receiving a request r_i to accept another request r_i on the same channel. However, the system will always return to a state where it remains ready to accept the next request r_i until r_i arrives. In case no request arrives it remains ready forever. The environment is under no obligation to issue requests, or to ever stop issuing requests. Hence for any numbers n_1 and $n_2 \in \mathbb{N} \cup \{\infty\}$ there is at least one run of the system in which exactly that many requests of type r_1 and r_2 are received.

FS 2 Every request r_i asks for a task t_i to be executed. The crucial property of the fair scheduler is that it will eventually grant any such request. Thus, we require that in any run of the system each occurrence of r_i will be followed by an occurrence of t_i."

FS 3 "We require that in any partial run of the scheduler there may not be more occurrences of t_i than of r_i, for $i = 1, 2$.

[3] It can be argued, however, that Knuth's mutual exclusion protocol is correct only when making certain assumptions on the hardware on which it will be running [7]; the same applies to all other mutual exclusion protocols mentioned in this section. This matter is not addressed in the present paper. However, the material presented in Section 9 paves the way for discussing it.

FS 4 The last requirement is that between each two occurrences of t_i and t_j for $i, j \in \{1,2\}$ an intermittent activity e is scheduled."

This fair scheduler serves two clients, but the concept generalises smoothly to N clients.

The intended applications of fair schedulers are for instance in operating systems, where multiple application processes compete for processing on a single core, or radio broadcasting stations, where the station manager needs to schedule multiple parties competing for airtime. In such cases each applicant must get a turn eventually. The event e signals the end of the time slot allocated to an application process on the single core, or to a broadcast on the radio station.

Fair schedulers occur (in suitable variations) in many distributed systems. Examples are *First in First out*[4], *Round Robin*, and *Fair Queueing* scheduling algorithms[5] as used in network routers [20, 21] and operating systems [15], or the *Completely Fair Scheduler*,[6] which is the default scheduler of the Linux kernel since version 2.6.23.

Each action r_i, t_i and e can be seen as a communication between the fair scheduler and one of its clients. In a reactive system such communications will take place only if both the fair scheduler and its client are ready for it. Requirement FS 1 of a fair scheduler quoted above effectively shifts the responsibility for executing r_i to the client. The actions t_i and e, on the other hand, are seen as the responsibility of the fair scheduler. We do not consider the possibility that the fair scheduler fails to execute t_i merely because the client does not collaborate. Hence [11] assumes that the client cannot prevent the actions t_i and e from occurring. It is furthermore assumed that executing the actions r_i, t_i and e takes a finite amount of time only.

A fair scheduler closely resembles a mutual exclusion protocol. However, its goal is not to achieve mutual exclusion. In most applications, mutual exclusion can be taken for granted, as it is physically impossible to allocate the single core to multiple applications at the same time, or the (single frequency) radio sender to multiple simultaneous broadcasts. Instead, its goal is to ensure that no applicant is passed over forever.

It is not hard to obtain a fair scheduler from a mutual exclusion protocol. For suppose we have a mutual exclusion protocol M, serving two processes P_i ($i = 1,2$). I instantiate the non-critical section of process P_i as patiently awaiting the request r_i. As soon as this request arrives, P_i leaves the noncritical section and starts the entry protocol to get access to the critical section. The liveness property for mutual exclusion guarantees that P_i will reach its critical section. Now the critical section consists of scheduling task t_i, followed by the intermittent activity e. Trivially, the composition of the two process P_i, in combination with protocol M, constitutes a fair scheduler, in that it meets the above four requirements.

One can not quite construct a mutual exclusion protocol from a fair scheduler, due to fact that in a mutual exclusion protocol leaving the critical section is controlled by the client process. For this purpose one would need to adapt the assumption that the client of a fair scheduler cannot block the intermittent activity e into the assumption that the client can postpone this action, but for a finite amount of time only. In this setting one can build a mutual exclusion protocol, serving two processes P_i ($i = 1,2$), from a fair scheduler F. Process i simply issues request r_i at F as soon as it has left the non-critical section, and when F communicates the action t_i, Process i enters its critical section. Upon leaving its critical section, which is assumed to happen after a finite amount of time, it participates in the synchronisation e with F. Trivially, this yields a correct mutual exclusion protocol.

[4] Also known as First Come First Served (FCFS)

[5] `http://en.wikipedia.org/wiki/Scheduling_(computing)`

[6] `http://en.wikipedia.org/wiki/Completely_Fair_Scheduler`

8 Formalising the Requirements for Fair Schedulers in Reactive LTL$_{-\mathbf{X}}$

The main reason fair schedulers were defined in [11] was to serve as an example of a realistic class of systems of which no representative can be correctly specified in CCS, or similar process algebras, or in Petri nets. Proving this impossibility result necessitated a precise formalisation of the four requirements quoted in Section 7. Through the provided translations of CCS and Petri nets into LTSs, a fair scheduler rendered in CCS or Petri nets can be seen as a state F in an LTS over the set $\{r_i, t_i, e \mid i = 1,2\}$ of visible actions; all other actions can be considered internal and renamed into τ.

Let a *partial trace* of a state s in an LTS be the sequence of visible actions encountered on a path starting in s [6]. Now the last two requirements (FS 3) and (FS 4) of a fair scheduler are simple properties that should be satisfied by all partial traces σ of state F:

(FS 3) σ contains no more occurrences of t_i than of r_i, for $i = 1,2$,

(FS 4) σ contains an occurrence of e between each two occurrences of t_i and t_j for $i, j \in \{1,2\}$.

FS 4 can be conveniently rendered in LTL$_{-\mathbf{X}}$:

(FS 4) $F \models \mathbf{G}\big(t_i \Rightarrow \big(t_i\mathbf{U}\big((\neg t_1 \wedge \neg t_2)\mathbf{W}e\big)\big)\big)$

for $i \in \{1,2\}$. Here the *weak until* modality $\psi\mathbf{W}\varphi$ is syntactic sugar for $\mathbf{G}\psi \vee (\psi\mathbf{U}\varphi)$. If I hadn't lost the $\mathbf{X}$ modality, I could write $\mathbf{X}$ for $t_i\mathbf{U}$ in the above formula; on Kripke structures distilled from LTSs the meaning is the same. The formula in FS 4 is of a kind where the meaning of $\models_B^{CC}$ is independent of B and CC. This follows from the fact that FS 4 merely formulates a property that should hold for all partial runs. Hence one need not worry about which B and CC to employ here.

Unfortunately, FS 3 cannot be formulated in LTL$_{-\mathbf{X}}$, due to the need to keep count of the difference in the number of r_i and t_i actions encountered on a path. However, one could strengthen FS 3 into

(FS 3′) σ contains an occurrence of r_i between each two occurrences of t_i, and prior to the first occurrence of t_i, for $i \in \{1,2\}$.

This would restrict the class of acceptable fair schedulers, but keep the most interesting examples. Consequently, the impossibility result from [11] applies to this modified class as well. FS 3 can be rendered in LTL$_{-\mathbf{X}}$ in the same style as FS 4:

(FS 3′) $F \models \big((\neg t_i)\mathbf{W}r_i\big) \wedge \mathbf{G}\big(t_i \Rightarrow \big(t_i\mathbf{U}\big((\neg t_i)\mathbf{W}r_i\big)\big)\big)$

for $i \in \{1,2\}$.

Requirement FS 2 involves a quantification over all complete runs of the system, and thus depends on the completeness criterion CC employed. It can be formalised as

(FS 2) $F \models_B^{CC} \mathbf{G}(r_i \Rightarrow \mathbf{F}t_i)$

for $i \in \{1,2\}$, where $B = \{r_1, r_2\}$. The set B should contain r_1 and r_2, as these actions are supposed to be under the control of the users of a fair scheduler. However, actions t_1, t_2 and e should not be in B, as they are under the control of the scheduler itself. In [11], the completeness criterion employed is justness, so the above formula with $CC := J$ captures the requirement on the fair schedulers that are shown in [11] not to exist in CCS or Petri nets. However, keeping CC a variable allows one to pose to the question under which completeness criterion a fair scheduler *can* be rendered in CCS. Naturally, it needs to be a stronger criterion than justness. In [11] it is shown that weak fairness suffices.

FS 2 is a good example of a requirement that can *not* be rendered correctly in standard LTL. Writing $F \models^{CC} \mathbf{G}(r_i \Rightarrow \mathbf{F}t_i)$ would rule out the complete runs of F that end because the user of F never supplies the input $r_j \in B$. The CCS process

$$F \stackrel{def}{=} r_1.r_2.t_1.e.t_2.e.F$$

for instance satisfies this formula, as well as FS 3 and 4; yet it does not satisfy requirement FS 2. Namely, the path consisting of the r_1-transition only is complete, since it ends in a state of which the only outgoing transition has the label $r_2 \in B$. Yet on this path r_1 is not followed by t_1.

Requirement FS 1 is by far the hardest to formalise. In [11] two formalisations are shown to be equivalent: one involving a coinductive definition of B-just paths that exploits the syntax of CCS, and the other requiring that requirements FS 2–4 are preserved under putting an input interface around process F. The latter demands that also $\widehat{F} := (I_1 \,|\, F[f] \,|\, I_2)\backslash\{c_1, c_2\}$ should satisfy FS 2–4; here f is a relabelling with $f(r_i) = c_i$, $f(t_i) = t_i$ and $f(e) = e$ for $i = 1, 2$, and $I_i \stackrel{def}{=} r_i.\bar{c}_i.I_i$ for $i \in \{1, 2\}$.

A formalisation of FS 1 on Petri nets also appears in [11]: each complete path π with only finitely many occurrences of r_i should contain a state (= marking) M, such that there is a transition v with $\ell(v) = r_i$ and ${}^\bullet v \leq M$, and for each transition u that occurs in π past M one has ${}^\bullet v \cap {}^\bullet u = \emptyset$.

When discussing proposals for fair schedulers by others, FS 1 is the requirement that is most often violated, and explaining why is not always easy.

In reactive LTL$_{-\mathbf{X}}$, this requirement is formalised as

(FS 1) $\quad F \models^{J}_{B\backslash\{r_i\}} \mathbf{GF}r_i$

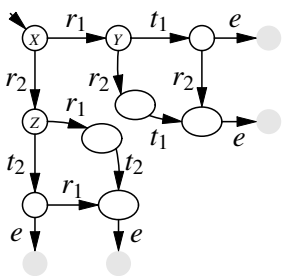
for $i \in \{1, 2\}$, or $F \models^{CC}_{B\backslash\{r_i\}} \mathbf{GF}r_i$ if one wants to discuss the completeness criterion CC as a parameter. The surprising element in this temporal judgement is the subscript $B \setminus \{r_i\} = \{r_{3-i}\}$, which contrasts with the assumption that requests are under the control of the environment. FS 1 says that, although we know that there is no guarantee that user i of F will ever issue request r_i, under the assumption that the user *does* want to make such a request, making the request should certainly succeed. This means that the protocol itself does not sit in the way of making this request.

The combination of requirements FS 1 and 2, which use different sets of blockable actions as a parameter, is enabled by reactive LTL$_{-\mathbf{X}}$ as presented here.

The following examples, taken from [11], show that all the above requirements are necessary for the result from [11] that fair schedulers cannot be rendered in CCS.

- The CCS process $F_1|F_2$ with $F_i \stackrel{def}{=} r_i.t_i.e.F_i$ satisfies FS 1, FS 2 and FS 3′. In FS 1 and 2 one needs to take $CC := J$, as progress is not a strong enough assumption here.
- The process $E_1|G|E_2$ with $E_i \stackrel{def}{=} r_i.E_i$ and $G \stackrel{def}{=} t_1.e.t_2.e.G$ satisfies FS 1, 2 and 4, again with $CC := J$.
- The process $E_1|E_2$ satisfies FS 1, 3′ and 4, again with $CC := J$ in FS 1.
- The process F_0 with $F_0 \stackrel{def}{=} r_1.t_1.e.F_0 + r_2.t_2.e.F_0$ satisfies FS 2–4. Here FS 2 merely needs $CC := Pr$, that is, the assumption of progress. Furthermore, it satisfies FS 1 with $CC := SF(\mathscr{T})$, as long as $\mathrm{R}_1, \mathrm{R}_2 \in \mathscr{T}$. Here R_i is the set of transitions with label r_i.

The process X given by $X \stackrel{def}{=} r_1.Y + r_2.Z$, $Y \stackrel{def}{=} r_2.t_1.e.Z + t_1.(r_2.e.Z + e.X)$ and $Z \stackrel{def}{=} r_1.t_2.e.Y + t_2.(r_1.e.Y + e.X)$, the *gatekeeper*, is depicted on the right. The grey shadows represent copies of the states at the opposite end of the diagram, so the transitions on the far right and bottom loop around. This process satisfies FS 3′ and 4, FS 2 with $CC := Pr$, and FS 1 with $CC := WF(\mathscr{T})$, thereby improving process F_0, and constituting the best CCS approximation of a fair scheduler seen so far. Yet, intuitively FS 1 is not ensured at all, meaning that weak fairness is too strong an assumption. Nothing really prevents all the choices between r_2 and any other action a to be made in favour of a.



9 Formalising Requirements for Mutual Exclusion in Reactive LTL$_{-\mathbf{X}}$

Define a process i participating in a mutual exclusion protocol to cycle through the stages *noncritical section*, *entry protocol*, *critical section*, and *exit protocol*, in that order, as explained in Section 6. Modelled as an LTS, its visible actions will be en_i, ln_i, ec_i and lc_i, of entering and leaving its (non)critical section. Put ln_i in B to make leaving the critical section a blockable action. The environment blocking it is my way of allowing the client process to stay in its noncritical section forever. This is the manner in which the requirement *Optionality* is captured in reactive temporal logic. On the other hand, ec_i should not be in B, for one does not consider the liveness property of a mutual exclusion protocol to be violated simply because the client process refuses to enter the critical section when allowed by the protocol. Likewise, en_i is not in B. Although exiting the critical section is in fact under control of the client process, it is assumed that it will not stay in the critical section forever. In the models of this paper this can be simply achieved by leaving lc_i outside B. Hence $B := \{ln_i \mid i = 1,\ldots,N\}$.

My first requirement on mutual exclusion protocols P simply says that the actions en_i, ln_i, ec_i and lc_i have to occur in the right order:

(ME 1) $P \models \big((\neg act_i)\mathbf{W}ln_i\big) \wedge \mathbf{G}\big(ln_i \Rightarrow \big(ln_i\mathbf{U}\big((\neg act_i)\mathbf{W}ec_i\big)\big)\big) \wedge \mathbf{G}\big(ec_i \Rightarrow \big(ec_i\mathbf{U}\big((\neg act_i)\mathbf{W}lc_i\big)\big)\big)$
$\wedge\, \mathbf{G}\big(lc_i \Rightarrow \big(lc_i\mathbf{U}\big((\neg act_i)\mathbf{W}en_i\big)\big)\big) \wedge \mathbf{G}\big(en_i \Rightarrow \big(en_i\mathbf{U}\big((\neg act_i)\mathbf{W}ln_i\big)\big)\big)$

for $i = 1,\ldots,N$. Here $act_i := (ln_i \vee ec_i \vee lc_i \vee en_i)$.

The second is a formalisation of *Safety*, saying that only one process can be in its critical section at the same time:

(ME 2) $P \models \mathbf{G}\,(ec_i \Rightarrow ((\neg ec_j)\mathbf{W}lc_i))$

for $i, j = 1,\ldots,N$ with $i \neq j$. Both ME 1 and ME 2 would be unaffected by changing $\models$ into $\models^{CC}$ or $\models^{CC}_B$.

Requirement *Liveness* of Section 6 can be formalised as

(ME 3) $P \models^{CC}_B \mathbf{G}(ln_i \Rightarrow \mathbf{F}ec_i)$

Here the choice of a completeness criterion is important. Finally, the following requirements are similar to *Liveness*, and state that from each section in the cycle of a Process i, the next section will in fact be reached. In regards to reaching the end of the noncritical section, this should be guaranteed only when assuming that the process wants to leave it critical section; hence ln_i is excepted from B.

(ME 4) $P \models^{CC}_B \mathbf{G}(ec_i \Rightarrow \mathbf{F}lc_i)$

(ME 5) $P \models^{CC}_B \mathbf{G}(lc_i \Rightarrow \mathbf{F}en_i)$

(ME 6) $P \models^{CC}_{B\setminus\{ln_i\}} \mathbf{F}ln_i \wedge \mathbf{G}(en_i \Rightarrow \mathbf{F}ln_i)$

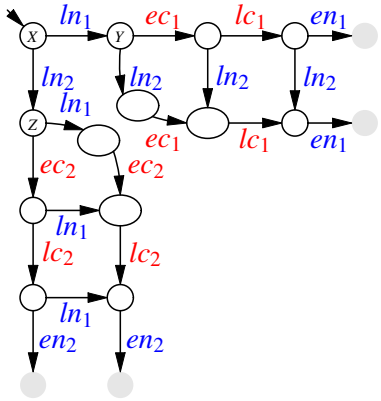
for $i = 1,\ldots,N$.

The requirement *Speed independence* is automatically satisfied for models of mutual exclusion protocols rendered in any of the formalisms discussed in this paper, as these formalisms lack the expressiveness to make anything dependent on speed.

The following examples show that none of the above requirements are redundant.

- The CCS process $F_1|F_2|\cdots|F_N$ with $F_i \stackrel{def}{=} ln_i.ec_i.lc_i.en_i.F_i$ satisfies all requirements, with $CC := J$, except for ME 2.
- The process $R_1|R_2|\cdots|R_N$ with $R_i \stackrel{def}{=} ln_i.\mathbf{0}$ satisfies all requirements except for ME 3.
- In case $N = 2$, the process $ln_1.ec_1.ln_2.ec_2.\mathbf{0} + ln_2.ec_2.ln_1.ec_1.\mathbf{0}$ satisfies all requirements except for ME 4. The case $N > 2$ is only notationally more cumbersome. In the same spirit one finds counterexamples failing only on ME 5, or on the second conjunct of ME 6.
- The process $\mathbf{0}$ satisfies all requirements except for the first conjunct of ME 6.
- In case $N = 1$, the process X with $X \stackrel{def}{=} lc_1.ec_1.lc_1.en_1.ln_1.X$ satisfies all requirements but ME 1.

The process X, a gatekeeper variant, given by $X \stackrel{def}{=} ln_1.Y + ln_2.Z$, $Y \stackrel{def}{=} ln_2.ec_1.lc_1.en_1.Z + ec_1.(ln_2.lc_1.en_1.Z + lc_1.(ln_2.en_1.Z + en_1.X))$ $Z \stackrel{def}{=} ln_1.ec_2.lc_2.en_2.Y + ec_2.(ln_1.lc_2.en_2.Y + lc_2.(ln_1.en_2.Y + en_2.X))$ is depicted on the right. It satisfies ME 1–5 with $CC := Pr$ and ME 6 with $CC := WF(\mathscr{T})$, where $\mathrm{LN}_1, \mathrm{LN}_2 \in \mathscr{T}$. It could be seen as a mediator that synchronises, on the actions ln_i, ec_i, lc_i and en_i, with the actual processes that need to exclusively enter their critical sections. Yet, it would not be commonly accepted as a valid mutual exclusion protocol, since nothing prevents it to never choose ln_2 when an alternative is available. This means that merely requiring weak fairness in ME 6 makes this requirement unacceptably weak. The problem with this protocol is that it ensures *Liveness* by making it hard for processes to leave their noncritical sections.



10 Reactive CTL

This section presents a reactive version of *Computation Tree Logic* (CTL) [4]. This shows that the ideas presented here are not specific to a linear-time logic. The syntax of CTL is

$$\varphi, \psi ::= p \mid \neg\varphi \mid \varphi \wedge \psi \mid \mathbf{EX}\varphi \mid \mathbf{AX}\varphi \mid \mathbf{EF}\varphi \mid \mathbf{AF}\varphi \mid \mathbf{EG}\varphi \mid \mathbf{AG}\varphi \mid \mathbf{E}\psi\mathbf{U}\varphi \mid \mathbf{A}\psi\mathbf{U}\varphi$$

with $p \in AP$ an atomic predicate. The relation $\models$ between states s in a Kripke structure, CTL formulae φ and completeness criteria CC is inductively defined by

- $s \models^{CC} p$, with $p \in AP$, iff $(s,p) \in \models$,
- $s \models^{CC} \neg\varphi$ iff $s \not\models^{CC} \varphi$,
- $s \models^{CC} \varphi \wedge \psi$ iff $s \models^{CC} \varphi$ and $s \models^{CC} \psi$,
- $s \models^{CC} \mathbf{EX}\varphi$ iff there is a state s' with $s \to s'$ and $s' \models^{CC} \varphi$,
- $s \models^{CC} \mathbf{AX}\varphi$ iff for each state s' with $s \to s'$ one has $s' \models^{CC} \varphi$,
- $s \models^{CC} \mathbf{EF}\varphi$ iff some complete path starting in s contains a state s' with $s' \models^{CC} \varphi$,
- $s \models^{CC} \mathbf{AF}\varphi$ iff each complete path starting in s contains a state s' with $s' \models^{CC} \varphi$,
- $s \models^{CC} \mathbf{EG}\varphi$ iff all states s' on some complete path starting in s satisfy $s' \models^{CC} \varphi$,
- $s \models^{CC} \mathbf{AG}\varphi$ iff all states s' on all complete paths starting in s satisfy $s' \models^{CC} \varphi$,
- $s \models^{CC} \mathbf{E}\psi\mathbf{U}\varphi$ iff some complete path π starting in s contains a state s' with $s' \models^{CC} \varphi$, and each state s'' on π prior to s' satisfies $s'' \models^{CC} \psi$,
- $s \models^{CC} \mathbf{A}\psi\mathbf{U}\varphi$ iff each complete path π starting in s contains a state s' with $s' \models^{CC} \varphi$, and each state s'' on π prior to s' satisfies $s'' \models^{CC} \psi$.

Exactly as for $\mathrm{LTL}_{-\mathbf{X}}$, this allows the formulation of CTL judgements $s \models^{CC}_{B} \varphi$.

11 Conclusion

I proposed a formalism for making temporal judgements $P \models^{CC}_{B} \varphi$, with P a process specified in any formalism that admits a translation into LTSs, φ a temporal formula from a logic like LTL or CTL, CC a completeness criterion, stating which paths in the LTS model complete system runs, and B the set of actions that may be blocked by the user or environment of a system. I applied this formalism to unambiguously express the requirements defining fair schedulers and mutual exclusion protocols.

References

[1] R. De Nicola & F.W. Vaandrager (1995): *Three Logics for Branching Bisimulation*. Journal of the ACM 42(2), pp. 458–487, doi:10.1145/201019.201032.

[2] E.W. Dijkstra (1962 or 1963): *Over de sequentialiteit van processbeschrijvingen*. Available at `http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD35.PDF`.

[3] E.W. Dijkstra (1965): *Solution of a problem in concurrent programming control*. Communications of the ACM 8(9), p. 569, doi:10.1145/365559.365617.

[4] E. Allen Emerson & Edmund M. Clarke (1982): *Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons*. Sci. Comput. Program. 2(3), pp. 241–266, doi:10.1016/0167-6423(83)90017-5.

[5] N. Francez (1986): *Fairness*. Springer, New York, doi:10.1007/978-1-4612-4886-6.

[6] R.J. van Glabbeek (1993): *The Linear Time – Branching Time Spectrum II; The semantics of sequential systems with silent moves*. In E. Best, editor: Proceedings *CONCUR'93*, 4^{th} International Conference on Concurrency Theory, Hildesheim, Germany, August 1993, LNCS 715, Springer, pp. 66–81, doi:10.1007/3-540-57208-2_6.

[7] R.J. van Glabbeek (2018): *Is Speed-Independent Mutual Exclusion Implementable?* In S. Schewe & L. Zhang, editors: Proceedings 29th International Conference on Concurrency Theory (CONCUR'18), Beijing, China, September 2018, Leibniz International Proceedings in Informatics (LIPIcs) 118, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, doi:10.4230/LIPIcs.CONCUR.2018.3.

[8] R.J. van Glabbeek (2019): *Ensuring liveness properties of distributed systems: Open problems*. Journal of Logical and Algebraic Methods in Programming 109:100480, doi:10.1016/j.jlamp.2019.100480. Available at `http://arxiv.org/abs/1912.05616`.

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

[10] R.J. van Glabbeek, U. Goltz & J.-W. Schicke (2011): *Abstract Processes of Place/Transition Systems*. Information Processing Letters 111(13), pp. 626–633, doi:10.1016/j.ipl.2011.03.013. Available at `http://arxiv.org/abs/1103.5916`.

[11] R.J. van Glabbeek & P. Höfner (2015): *CCS: It's not fair! - Fair schedulers cannot be implemented in CCS-like languages even under progress and certain fairness assumptions*. Acta Informatica 52(2-3), pp. 175–205, doi:10.1007/s00236-015-0221-6. Available at `http://arxiv.org/abs/1505.05964`.

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

[13] R.J. van Glabbeek & P. Höfner (2019): *Progress, Justness and Fairness*. ACM Computing Surveys 52(4):69, doi:10.1145/3329125. Available at `https://arxiv.org/abs/1810.07414`.

[14] M. Huth & M.D. Ryan (2004): *Logic in Computer Science — Modelling and Reasoning about Systems*, 2nd edition. Cambridge University Press, doi:10.1017/CBO9780511810275.

[15] L. Kleinrock (1964): *Analysis of A Time-Shared Processor*. Naval Research Logistics Quarterly 11(1), pp. 59–73, doi:10.1002/nav.3800110105.

[16] D.E. Knuth (1966): *Additional comments on a problem in concurrent programming control*. Communications of the ACM 9(5), pp. 321–322, doi:10.1145/355592.365595.

[17] L. Lamport (1974): *A New Solution of Dijkstra's Concurrent Programming Problem*. Communications of the ACM 17(8), pp. 453–455, doi:10.1145/361082.361093.

[18] L. Lamport (1983): *What good is temporal logic?* In R.E. Mason, editor: Information Processing 83, North-Holland, pp. 657–668.

[19] R. Milner (1990): *Operational and algebraic semantics of concurrent processes*. In J. van Leeuwen, editor: *Handbook of Theoretical Computer Science*, chapter 19, Elsevier Science Publishers B.V. (North-Holland), pp. 1201–1242. Alternatively see *Communication and Concurrency*, Prentice-Hall, Englewood Cliffs, 1989, of which an earlier version appeared as *A Calculus of Communicating Systems*, LNCS 92, Springer, 1980, doi:10.1007/3-540-10235-3.

[20] J. Nagle (1985): *On Packet Switches with Infinite Storage*. RFC 970, Network Working Group. Available at `http://tools.ietf.org/rfc/rfc970.txt`.

[21] J. Nagle (1987): *On Packet Switches with Infinite Storage*. *IEEE Trans. Communications* 35(4), pp. 435–438, doi:10.1109/TCOM.1987.1096782.

[22] G.L. Peterson (1981): *Myths About the Mutual Exclusion Problem*. *Information Processing Letters* 12(3), pp. 115–116, doi:10.1016/0020-0190(81)90106-X.

[23] Amir Pnueli (1977): *The Temporal Logic of Programs*. In: *Foundations of Computer Science* (FOCS '77), IEEE, pp. 46–57, doi:10.1109/SFCS.1977.32.

[24] B.K. Szymański (1988): *A simple solution to Lamport's concurrent programming problem with linear wait*. In J. Lenfant, editor: *Proceedings of the 2nd international conference on Supercomputing, ICS 1988, Saint Malo, France, July 4-8, 1988*, ACM, pp. 621–626, doi:10.1145/55364.55425.

Substructural Observed Communication Semantics

Ryan Kavanagh
Computer Science Department
Carnegie Mellon University
Pittsburgh, Pennsylvania, 15213-3891, USA
rkavanagh@cs.cmu.edu

Session-types specify communication protocols for communicating processes, and session-typed languages are often specified using substructural operational semantics given by multiset rewriting systems. We give an observed communication semantics [2] for a session-typed language with recursion, where a process's observation is given by its external communications. To do so, we introduce *fair executions* for multiset rewriting systems, and extract observed communications from fair process executions. This semantics induces an intuitively reasonable notion of observational equivalence that we conjecture coincides with semantic equivalences induced by denotational semantics [15], bisimulations [13], and barbed congruences [16, 27] for these languages.

1 Introduction

A proofs-as-processes correspondence between linear logic and the session-typed π-calculus is the basis of many programming languages for message-passing concurrency [4, 5, 26, 28]. Session types specify communication protocols, and all communication with session-typed processes must respect these protocols. If we take seriously the idea that we can only interact with processes through session-typed communication, then the only thing we can observe about them is their communications. Indeed, timing differences in communication are not meaningful due to the non-deterministic scheduling of process reductions, and "forwarding" or "linking" of channels renders process termination meaningless, even in the presence of recursion. It follows that processes should be observationally indistinguishable only if they always send the same output given the same input.

These ideas underlie Atkey's [2] novel *observed communication semantics* (OCS) for Wadler's Classical Processes [28]. Atkey's OCS uses a big-step evaluation semantics to observe communications on channels deemed "observable". Processes are then observationally equivalent whenever they have the same observed communications in all contexts.

Building on these ideas, **we give an OCS for session-typed languages that are specified using substructural operational semantics** (SSOS), a form of multiset rewriting. Our work differs from Atkey's on several key points. First, we assume that communication is asynchronous rather than synchronous. This assumption costs us nothing, for synchronous communication can be encoded in asynchronous systems [20], and it simplifies the semantics by eliminating the need for "configurations" and "visible" cuts. More importantly, **our OCS supports recursive and non-terminating processes**. To do so, we observe communications from process traces in (a conservative extension of) the usual SSOSs, instead of defining a separate big-step semantics.

To ensure that observed communications are well-defined in the presence of non-termination, we require that process executions be *fair*. Intuitively, fairness ensures that if a process can make progress, then it eventually does so. Fairness is also motivated by ongoing efforts to relate existing SSOSs to domain-theoretic semantics for this style of language [15]. There, processes denote continuous functions

O. Dardha and J. Rot (Eds.): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics (EXPRESS/SOS 2020).
EPTCS 322, 2020, pp. 69–87, doi:10.4204/EPTCS.322.7

between domains of session-typed communications, and fairness is built-in. To this end, **we introduce *fair executions* of multiset rewriting systems** (MRS) and give sufficient conditions for an MRS to have fair executions. We also introduce a new notion of trace equivalence, *union-equivalence*, that is key to defining our OCS.

We study fair executions of MRSs and their properties in section 2. In section 3, we give an SSOS for a session-typed language arising from a proofs-as-processes interpretation of intuitionistic linear logic. It supports recursive processes and types. Though it is limited, it represents the core of other SSOS-specified session-typed languages [3, 13, 15, 20, 25], and the techniques presented in this paper scale to their richer settings. In section 4, we give our observed communication semantics, where we use a coinductively defined judgment to extract observations from fair executions.

2 Fair Executions of Multiset Rewriting Systems

In this section, we introduce *fairness* and *fair executions* for multiset rewriting systems. We begin by revisiting (first-order) multiset rewriting systems, as presented by Cervesato et al. [9]. We present a notion of fairness for sequences of rewriting steps, and constructively show that under reasonable hypotheses, all fair sequences from the same multiset are permutations of each other. We introduce a new notion of trace equivalence, "union-equivalence", and give sufficient conditions for traces to be union-equivalent. Fairness and union-equivalence will be key ingredients for defining the observed communication semantics of section 4.

A **multiset** M is a pair (S,m) where S is a set (the *underlying set*) and $m : S \to \mathbb{N}$ is a function. It is finite if $\sum_{s\in S} m(s)$ is finite. We say s is an **element** of M, $s \in M$, if $m(s) > 0$. When considering several multisets, we assume without loss of generality that they have equal underlying sets. The **sum** M_1, M_2 of multisets $M_1 = (S,m_1)$ and $M_2 = (S,m_2)$ is the multiset $(S, \lambda s \in S.m_1(s) + m_2(s))$. Their **intersection** $M_1 \cap M_2$ is the multiset $(S, \lambda s \in S.\min(m_1(s), m_2(s)))$. Their **difference** $M_1 \setminus M_2$ is the multiset $(S, \lambda s \in S.\max(0, m_1(s) - m_2(s)))$. We say that M_1 is **included** in M_2, written $M_1 \subseteq M_2$, if $m_1(s) \leq m_2(s)$ for all $s \in S$.

Consider finite multisets M of first-order atomic formulas over some signature whose constants are drawn from some countably infinite set. We call closed formulas **judgments**. Judgments represent facts, some of which we may deem to be persistent. To this end, we partition formulas as **persistent** (indicated by bold face, **p**) and **ephemeral** (indicated by sans serif face, p). We write $M(\vec{x})$ to mean that the formulas in M draw their variables from $\vec{x}$. A **multiset rewrite rule** r is an ordered pair of multisets $F(\vec{x})$ and $G(\vec{x},\vec{n})$, where the multiset $\pi(\vec{x})$ of persistent formulas in $F(\vec{x})$ is included in $G(\vec{x},\vec{n})$. We interpret the variables $\vec{x}$ as being universally quantified and the variables $\vec{n}$ as being existentially quantified. This relation is made explicit using the syntax

$$r : \forall \vec{x}.F(\vec{x}) \to \exists \vec{n}.G(\vec{x},\vec{n}).$$

In practice, we often elide $\forall \vec{x}$ and do not repeat the persistent formulas $\pi(\vec{x}) \subseteq F(\vec{x})$ on the right side of the arrow. A **multiset rewriting system** (MRS) is a set $\mathscr{R}$ of multiset rewrite rules.

Multiset rewrite rules describe localized changes to multisets of judgments. Given a rule $r : \forall \vec{x}.F(\vec{x}) \to \exists \vec{n}.G(\vec{x},\vec{n})$ in $\mathscr{R}$ and some choice of constants $\vec{c}$ for $\vec{x}$, we say that the **instantiation** $r(\vec{c}) : F(\vec{c}) \to \exists \vec{n}.G(\vec{c},\vec{n})$ is **applicable** to a multiset M of judgments if there exists a multiset M' such that $M = F(\vec{c}), M'$. The rule r is applicable to M if $r(\vec{c})$ is applicable to M for some $\vec{c}$. In these cases, the **result** of applying $r(\vec{c})$ to M is the multiset $G(\vec{c},\vec{d}), M'$, where $\vec{d}$ is a choice of fresh constants. In particular, we assume that the constants $\vec{d}$ do not appear in M or in $\mathscr{R}$. We call $\theta = [\vec{c}/\vec{x}]$ the **matching substitution** and $\xi = [\vec{d}/\vec{n}]$

the **fresh-constant substitution**. The **instantiating substitution** for r relative to M is the composite substitution $\delta = (\theta, \xi)$. We capture this relation using the syntax

$$F(\vec{c}), M' \xrightarrow{(r;\delta)} G(\vec{c},\vec{n}), M'.$$

For conciseness, we often abuse notation and write $r(\theta)$, $F(\theta)$, and $G(\theta,\xi)$ for $r(\vec{c})$, $F(\vec{c})$, and $G(\vec{c},\vec{d})$. We call $F(\vec{c})$ the **active** multiset and M' the **stationary** multiset.

Given an MRS $\mathscr{R}$ and a multiset M_0, a **trace** from M_0 is a countable sequence of steps

$$M_0 \xrightarrow{(r_1;\delta_1)} M_1 \xrightarrow{(r_2;\delta_2)} M_2 \xrightarrow{(r_3;\delta_3)} \cdots \tag{1}$$

such that, where $\delta_i = (\theta_i, \xi_i)$,

1. for all i, ξ_i is one-to-one;
2. for all $i < j$, the constants in M_i and ξ_j are disjoint.

The notation $(M_0, (r_i;\delta_i)_{i\in I})$ abbreviates the trace (1), where I always ranges over $\mathbb{N}^+$ or $\mathbf{n} = \{1,\ldots,n\}$ for some $n \in \mathbb{N}$. An **execution** is a maximally long trace.

Example 1. *We model queues using an MRS. Let the judgment* $\mathsf{que}(q,\$)$ *mean that q is the empty queue, and let* $\mathsf{que}(q, v \to q')$ *mean that the queue q has value v at its head and that its tail is the queue q'. Then the multiset* $Q = \mathsf{que}(q, 0 \to q'), \mathsf{que}(q',\$)$ *describes a one-element queue containing* 0*. The following two rules capture enqueuing values on empty and non-empty queues, respectively, where the formula* $\mathsf{enq}(q,v)$ *is used to enqueue v onto the queue q:*

$$e_1 : \forall x,y.\, \mathsf{enq}(x,y), \mathsf{que}(x,\$) \to \exists z.\, \mathsf{que}(x, y \to z), \mathsf{que}(z,\$),$$
$$e_2 : \forall x,y,z,w.\, \mathsf{enq}(x,y), \mathsf{que}(x, z \to w) \to \mathsf{que}(x, z \to w), \mathsf{enq}(w,y).$$

The following sequence is an execution from $Q, \mathsf{enq}(q,1)$*, and it captures enqueuing 1 on the queue q:*

$$Q, \mathsf{enq}(q,1) \xrightarrow{(e_2;([q,1,0,q'/x,y,z,w],\emptyset))} Q, \mathsf{enq}(q',1) \xrightarrow{(e_1;([q'/x,y],[a/z]))} \mathsf{que}(q, 0 \to q'), \mathsf{que}(q', 1 \to a), \mathsf{que}(a,\$).$$

The constants in fresh-constant substitutions are not semantically meaningful, so we identify traces up to refreshing substitutions. A **refreshing substitution** for a trace $T = (M_0, (r_i; (\theta_i, \xi_i))_i)$ is a collection of fresh-constant substitutions $\eta = (\eta_i)_i$ such that $[\eta]T = (M_0, (r_i; (\theta_i, \eta_i))_i)$ is also a trace. Explicitly, we identify traces T and T' if there exists a refreshing substitution η such that $T' = [\eta]T$.

Given rules $r_i : \forall \vec{x}_i.F_i(\vec{x}_i) \to \exists \vec{n}_i.G_i(\vec{x}_i,\vec{n}_i)$ and matching substitutions θ_i for $i = 1,2$, we say that the instantiations $r_1(\theta_1)$ and $r_2(\theta_2)$ are **equivalent**, $r_1(\theta_1) \equiv r_2(\theta_2)$, if both $F_1(\theta_1) = F_2(\theta_2)$ and (up to renaming of bound variables) $\exists \vec{n}_1.G_1(\theta_1,\vec{n}_1) = \exists \vec{n}_2.G_2(\theta_2,\vec{n}_2)$; otherwise they are **distinct**. Application does not distinguish between equivalent instantiations: if $r_1(\theta_1) \equiv r_2(\theta_2)$ are applicable to M_0, then applying each to M_0 gives the same result up to refreshing substitution.

Given an MRS $\mathscr{R}$, we say that an execution $(M_0, (r_i;\delta_i)_{i\in I})$ is **fair** if for all $i \in I$, $r \in \mathscr{R}$, and θ, whenever $r(\theta)$ is applicable to M_i, there exists a $j > i$ such that $r_j(\theta_j) \equiv r(\theta)$. Given a fair trace T, we write $\phi_T(i,r,\theta)$ for the least such j. In the case of MRSs specifying SSOSs of session-typed languages, this notion of fairness implies *strong process fairness* [11, 12, 18], which guarantees that if a process can take a step infinitely often, then it does so infinitely often. In particular, it implies that if a process can take a step, then it eventually does so.

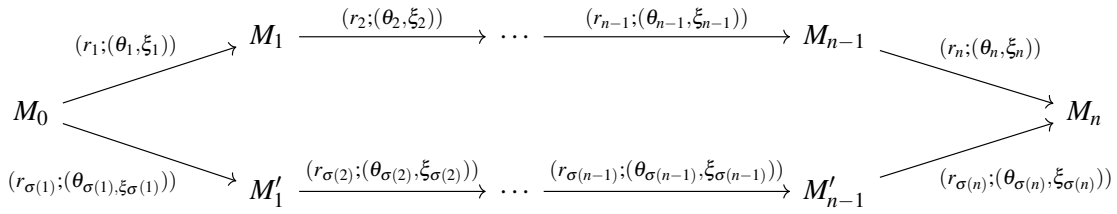
Example 2. *The execution of example 1 is fair.*

Proposition 1 (Fair Tail Property)**.** *If $(M_0, (t_i; \delta_i)_{i\in I})$ is fair, then so is $(M_n, (t_i; \delta_i)_{n<i,i\in I})$ for all $n \in I$.*

We consider various criteria that imply fairness. The first will be interference-freedom, which roughly means that at any given point, the order in which we apply applicable rules does not matter. It will hold whenever the rules do not "overlap". In general, given an MRS $\mathscr{R}$ and a property P, we say P holds **from** M_0 if for all traces $(M_0, (r_i; \delta_i)_{i\in I})$, P holds for M_0 and for M_i for all $i \in I$.

Write S_I for the group of bijections on I; its elements are called permutations. A permutation $\sigma \in S_I$ acts on a trace $T = (M_0, (t_i; \delta_i)_{i\in I})$ to produce a sequence $\sigma \cdot T = (M_0, (t_{\sigma(i)}; \delta_{\sigma(i)})_{i\in I})$. This sequence $\sigma \cdot T$ is a **permutation of** T whenever it is also a trace. We adopt group-theoretic notation for cyclic permutations and write $(x, \sigma(x), \sigma(\sigma(x)), \dots)$ for a cyclic permutation $\sigma : I \to I$; implicit is that all elements not in the orbit of x are fixed by σ. Cycles of length two are called transpositions.

Consider an MRS $\mathscr{R}$ and let $r_1(\theta_1), \dots, r_n(\theta_n)$ enumerate all distinct instantiations of rules in $\mathscr{R}$ applicable to M_0. We say that $\mathscr{R}$ **commutes** on M_0 or is **interference-free** on M_0 if for all corresponding pairwise-disjoint fresh-constant substitutions ξ_i, the following diagram commutes for all permutations $\sigma \in S_{\mathbf{n}}$, and both paths around it are traces:



We note that interference-freedom is only defined if the enumeration of distinct applicable instantiations is finite. The following proposition is an immediate consequence of the definition of commuting rules:

Proposition 2. *Let $\mathscr{R}$ commute on M_0, and let $r_i(\theta_i)$ with $1 \le i \le n$ be the distinct instantiations applicable on M_0. If $M_0 \xrightarrow{(r_1;(\theta_1,\xi_1))} M_1$, then $r_2(\theta_2), \dots, r_n(\theta_n)$ are applicable to and commute on M_1.*

Interference-freedom implies the existence of fair executions:

Proposition 3 (Fair Scheduler)**.** *Assume the axiom of countable choice. If $\mathscr{R}$ is interference-free from M_0, then there is a fair execution from M_0.*

Proof (Sketch). Let Q be a queue of rule instantiations applicable to M_0. Given M_n, dequeue a rule $r_{n+1}(\theta_{n+1})$ from Q and use the axiom of countable choice to choose a suitably disjoint fresh-constant substitution ξ_{n+1}. By interference-freedom, it is applicable to M_n, and let M_{n+1} be the result of doing so. Enqueue all newly-applicable rule instantiations. If Q is ever empty, then the trace is finite but maximally long. In all cases, the trace gives a fair execution: every distinct applicable rule instantiation is enqueued and then applied after some finite number of steps. □

Though interference-freedom simplifies fair scheduling, it is primarily of interest for reasoning about executions. For example, it is useful for showing confluence properties. It also lets us safely permute certain steps in a trace without affecting observations for session-typed processes (see section 4). This can simplify process equivalence proofs, because it lets us assume that related steps in an execution happen one after another.

Interference-freedom is a strong property, but it arises frequently in nature. This is because many systems can be captured using rules whose active multisets do not overlap, and rules whose active multisets are non-overlapping commute. In fact, even if their active multisets overlap, the rules do not disable each other so long as they preserve these overlaps.

To make this intuition explicit, consider multisets $M_i \subseteq M$ for $1 \leq i \leq n$. Their **overlap** in M is $\Omega_M(M_1,\ldots,M_n) = M_1,\ldots,M_n \setminus M$. Consider an MRS $\mathscr{R}$ and let $r_i(\theta_i) : F_i(\theta_i) \to \exists \vec{n}_i.G_i(\theta_i,\vec{n}_i)$, $1 \leq i \leq n$, enumerate all distinct instantiations of rules in $\mathscr{R}$ applicable to M. We say that $\mathscr{R}$ is **non-overlapping** on M if for all $1 \leq i \leq n$ and fresh-constant substitutions ξ_i, $F_i(\theta_i) \cap \Omega_M(F_1(\theta_1),\ldots,F_n(\theta_n)) \subseteq G_i(\theta_i,\xi_i)$.

Example 3. *The MRS given by example 1 is non-overlapping from any multiset of the form Q,E where Q is a queue rooted at q, and E contains at most one judgment of the form* $\mathsf{enq}(q,v)$.

Proposition 4 characterizes the application of non-overlapping rules, while proposition 5 characterizes the relationship between commuting and non-overlapping rules.

Proposition 4. *Let $\mathscr{R}$ be non-overlapping on M_0 and let $r_i(\theta_i) : F_i(\theta_i) \to \exists \vec{n}_i.G(\theta_i,\vec{n}_i)$ with $1 \leq i \leq n$ be the distinct instantiations applicable to M_0. If $M_0 \xrightarrow{(r_1;(\theta_1,\xi_1))} M_1$ and $r_1,\ldots,r_n$ are non-overlapping on M_0, then $r_2(\theta_2),\ldots,r_n(\theta_n)$ are applicable to and non-overlapping on M_1.*

In particular, set $O = \Omega_{M_0}(F_1,\ldots,F_n) \cap F_1$. There exist F_1' and G_1' be such that $F_1 = O,F_1'$ and $G_1 = O,G_1'$, and there exists an M such that $M_0 = O,F_1',M$ and $M_1 = O,G_1',M$. The instantiations $r_2(\theta_2),\ldots,r_n(\theta_n)$ are all applicable to $O,M \subseteq M_1$.

Proposition 5. *An MRS commutes on M_0 if it is non-overlapping on M_0; the converse is false.*

For the remainder of this section, assume that if $(M_0,(r_i;\delta_i)_i)$ is a fair trace, then its MRS is interference-free from M_0. Interference-freedom implies the ability to safely permute finitely many steps that do not depend on each other. However, it is not obvious that finite permutations, let alone infinite permutations, preserve fairness. To show that they do, we use the following lemma to reduce arguments about infinite permutations to arguments about finite permutations:

Lemma 1. *For all $n \in \mathbb{N}$ and permutations $\sigma : \mathbb{N} \to \mathbb{N}$, set $\chi_\sigma(n) = \sup_{k \leq n} \sigma^{-1}(k)$. Then there exist permutations $\tau,\rho : \mathbb{N} \to \mathbb{N}$ such that $\sigma = \rho \circ \tau$, $\tau(k) = k$ for all $k > \chi_\sigma(n)$, and $\rho(k) = k$ for all $k \leq n$.*

The following proposition shows that permutations of prefixes of traces preserve fairness. Its proof uses a factorization of permutations into cycles permuting adjacent steps, where each cycle preserves fairness.

Proposition 6. *Consider an MRS $\mathscr{R}$ that is interference-free from M_0 and let $T = (M_0,(r_i;(\theta_i,\xi_i))_{i\in I})$ be a trace, an execution, or a fair execution. Let $\sigma \in S_I$ be such that for some $n \in I$, $\sigma(i) = i$ for all $i > n$. Then $\sigma \cdot T$ is respectively a trace, an execution, or a fair execution.*

Corollary 1. *Fairness is invariant under permutation, that is, if $\mathscr{R}$ is interference-free from M_0, T is a fair trace from M_0, and $\Sigma = \sigma \cdot T$ is a permutation of T, then Σ is also fair.*

Proof. Let $T = (M_0,(t_i;\delta_i)_i)$ and $\delta_i = (\theta_i,\xi_i)$, and let Σ be the trace $M_0 = \Sigma_0 \xrightarrow{(t_{\sigma(1)};\delta_{\sigma(1)})} \Sigma_1 \xrightarrow{(t_{\sigma(2)};\delta_{\sigma(2)})} \cdots$. Consider some rule $r \in \mathscr{R}$ such that $\Sigma_i \xrightarrow{(r;(\theta,\xi))} \Sigma_i'$. We must show that there exists a j such that $\sigma(j) > \sigma(i)$, $t_{\sigma(j)}(\theta_{\sigma(j)}) \equiv r(\theta)$.

Let the factorization $\sigma = \rho \circ \tau$ be given by lemma 1 for $n = \sigma(i)$. By proposition 6, we get that $\tau \cdot T$ is fair. Moreover, by construction of τ, $\tau \cdot T$ and Σ agree on the first n steps and $n+1$ multisets. By fairness, there exists a $k > \sigma(i)$ such that the k-th step in $\tau \cdot T$ is $r(\theta)$. By construction of ρ, $\rho(k) > \sigma(i)$, so this step appears after Σ_i in Σ as desired. We conclude that Σ is fair. □

Corollary 1 established that permutations preserve fairness. Relatedly, all fair traces from a given multiset are permutations of each other. To do show this, we construct a potentially infinite sequence of permutations and use the following lemma to compose them:

Lemma 2. *Let $(\sigma_n)_{n\in I}$ be a family of bijections on I such that for all $m < n$,*

$$(\sigma_n \circ \cdots \circ \sigma_1)(m) = (\sigma_m \circ \cdots \circ \sigma_1)(m).$$

Let $\sigma : I \to I$ be given by $\sigma(m) = (\sigma_m \circ \cdots \circ \sigma_1)(m)$. Then σ is injective, but need not be surjective.

Lemma 3. *Let $\mathscr{R}$ be interference-free from M_0. Consider a fair execution $T = (M_0, (r_i; (\theta_i, \xi_i))_{i\in I})$ and a step $M_0 \xrightarrow{(t;(\tau,\rho))} M_1'$. Set $n = \phi_T(0, t, \tau)$ (so $t(\tau) \equiv r_n(\theta_n)$). Then $(1, \ldots, n) \cdot T$ is a permutation of T with $(t; (\tau, \xi_n))$ as its first step, and it is a fair execution.*

Proposition 7. *If $\mathscr{R}$ is interference-free from M_0, then all fair executions from M_0 are permutations of each other.*

Proof (Sketch). Consider traces $R = (R_0, (r_i; (\theta_i, \xi_i))_{i\in I})$ and $T = (T_0, (t_j; (\tau_j, \zeta_j))_{j\in J})$ where $R_0 = M_0 = T_0$. We construct a sequence of permutations $\sigma_0, \sigma_1, \ldots$, where $\Phi_0 = R$ and the step $\Phi_{n+1} = \sigma_{n+1} \cdot \Phi_n$ is given by lemma 3 such that Φ_{n+1} agrees with T on the first $n+1$ steps. We then assemble these permutations σ_n into an injection σ using lemma 2; fairness ensures that it is a surjection. We have $T = \sigma \cdot R$ by construction. □

Let the support of a multiset $M = (S, m)$ be the set $\mathrm{supp}(M) = \{s \in S \mid m(s) > 0\}$. We say that two traces $T = (M_0; (r_i, \delta_i)_I)$ and T' are **union-equivalent** if T' can be refreshed to a trace $(N_0; (s_j, \rho_j)_j)$ such that the unions of the supports of the multisets in the traces are equal, i.e., such that

$$\bigcup_{i\geq 0} \mathrm{supp}(M_i) = \bigcup_{j\geq 0} \mathrm{supp}(N_j)$$

Lemma 4. *Consider an MRS and assume T is a permutation of S. Then T and S are union-equivalent.*

Proof. Consider a trace $(M_0, (r_i; \delta_i)_i)$. For all n, each judgment in M_n appears either in M_0 or in the result of some rule r_i with $i \leq n$. Traces T and S start from the same multiset and have the same rules. It follows that they are union-equivalent. □

Corollary 2 will be key in section 4 to showing that processes have unique observations.

Corollary 2. *If $\mathscr{R}$ is interference-free from M, then all fair executions from M are union-equivalent.*

3 Session-Typed Processes

Session types specify communication protocols between communicating processes. In this section, we present a session-typed language arising from a proofs-as-programs interpretation of intuitionistic linear logic [4] extended to support recursive processes and recursive types.

We let A, B, C range over session types and a, b, c range over channel names. A process P provides a distinguished service A_0 over some channel c_0, and may use zero or more services A_i on channels c_i. In this sense, a process P is a server for the service A_0, and a client of the services A_i. The channels $c_1 : A_1, \ldots, c_n : A_n$ form a linear context Δ. We write $\Delta \vdash P :: c_0 : A_0$ to capture these data. We also allow P to depend on process variables p_i of type $\{b : B \leftarrow \Delta\}$. Values of type $\{b : B \leftarrow \Delta\}$ are processes Q such that $\Delta \vdash Q :: b : B$. We write Π for structural contexts of process variables $p_i : \{a_i : A_i \leftarrow \Delta_i\}$. These data are captured by the judgment $\Pi\,;\, \Delta \vdash P :: c_0 : A_0$, and we say that P is closed if Π is empty.

At any given point in a computation, communication flows in a single direction on a channel $c : A$. The direction of communication is determined by the *polarity* of the type A, where session types are

partitioned as positive or negative [20]. Consider a process judgment $\Pi\,;\Delta \vdash P :: c_0 : A_0$. Communication on positively-typed channels flows from left-to-right in this judgment: if A_0 is positive, then P can only send output on c_0, while if A_i is positive for $1 \leq i \leq n$, then P can only receive input on c_i. Symmetrically, communication on negatively-typed channels flows from right-to-left in the judgment. Bidirectional communication arises from the fact that the type of a channel evolves over the course of a computation, sometimes becoming positive, sometimes becoming negative.

Most session types have a polar dual, where the direction of communication is reversed. With one exception, we only consider positive session types here. Negative session types pose no difficulty and can be added by dualizing the constructions. To illustrate this dualization, we also consider the (negative) external choice type $\&\{l : A_l\}_{l\in L}$, the polar dual of the (positive) internal choice type $\oplus\{l : A_l\}_{l\in L}$.

The operational behaviour of closed processes is given by a substructural operational semantics (SSOS) in the form of a multiset rewriting system. The judgment $\mathsf{proc}(c,P)$ means that the closed process P provides a channel c. The judgment $\mathsf{msg}(c,m)$ means the channel c is carrying a message m. Process communication is asynchronous: processes send messages without synchronizing with recipients. To ensure that messages on a given channel are received in order, the $\mathsf{msg}(c,m)$ judgment encodes a queue-like structure similar to the queues of example 1, and we ensure that each channel name c is associated with at most one $\mathsf{msg}(c,m)$ judgment. For example, the multiset $\mathsf{msg}(c_0,m_0;c_0 \leftarrow c_1), \mathsf{msg}(c_1,m_0;c_1 \leftarrow c_2), \ldots$ captures the queue of messages $m_0, m_1, \ldots$ on c_0. There is no global ordering on sent messages: messages sent on different channels can be received out of order. We extend the usual SSOS with a new persistent judgment, $\mathbf{type}(c : A)$, which means that channel c has type A.

The **initial configuration** of $\cdot\,; c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0$ is the multiset

$$\mathsf{proc}(c_0,P), \mathbf{type}(c_0 : A_0), \ldots, \mathbf{type}(c_n : A_n).$$

A **process trace** is a trace from the initial configuration of a process, and a multiset in it is a **configuration**. A **fair execution** of $\cdot\,;\Delta \vdash P :: c : A$ is a fair execution from its initial configuration.

We give the typing rules and the substructural operational semantics in section 3.1. In section 3.2, we study properties of process traces and fair executions. In particular, we show that each step in these traces preserves various invariants, that the MRS of section 3.1 is non-interfering from initial process configurations, and that every process has a fair execution.

3.1 Statics and Dynamics

The process $a \to b$ forwards all messages from the channel a to the channel b; it assumes that both channel have the same positive type. It is formed by ($\textsc{Fwd}^+$) and its operational behaviour is given by (2).

$$\frac{}{\Pi\,; a : A \vdash a \to b :: b : A}\ (\textsc{Fwd}^+)$$
$$\mathsf{msg}(a,m), \mathsf{proc}(b, a \to b) \to \mathsf{msg}(b,m) \tag{2}$$

Process composition $a : A \leftarrow P;\ Q$ spawns processes P and Q that communicate over a shared private channel a of type A. It captures Milner's "parallel composition plus hiding" operation [17, pp. 20f.]. To ensure that the shared channel is truly private, we generate a globally fresh channel b for P and Q to communicate over.

$$\frac{\Pi\,;\Delta_1 \vdash P :: a : A \quad \Pi\,; a : A, \Delta_2 \vdash Q :: c : C}{\Pi\,;\Delta_1,\Delta_2 \vdash a : A \leftarrow P;\ Q :: c : C}\ (\textsc{Cut})$$
$$\mathsf{proc}(c, a : A \leftarrow P;\ Q) \to \exists b.\mathsf{proc}(b,[b/a]P), \mathsf{proc}(c,[b/a]Q), \mathbf{type}(b : A) \tag{3}$$

The process close a closes a channel a of type $\mathbf{1}$ by sending the "close message" $*$ over a. Dually, the process wait a; P blocks until it receives the close message on the channel a, and then continues as P.

$$\frac{}{\Pi\,;\,\cdot \vdash \mathsf{close}\ a :: a : \mathbf{1}}\ (\mathbf{1}R) \qquad \frac{\Pi\,;\,\Delta \vdash P :: c : C}{\Pi\,;\,\Delta, a : \mathbf{1} \vdash \mathsf{wait}\ a; P :: c : C}\ (\mathbf{1}L)$$

$$\mathsf{proc}(a, \mathsf{close}\ a) \to \mathsf{msg}(a, *) \tag{4}$$

$$\mathsf{msg}(a, *), \mathsf{proc}(c, \mathsf{wait}\ a; P) \to \mathsf{proc}(c, P) \tag{5}$$

Processes can communicate channels over channels of type $B \otimes A$, where the transmitted channel has type B and subsequent communication has type A. The process send $a\ b$; P sends a channel b over channel a and then continues as P. To ensure a queue-like structure for messages on a, we generate a fresh channel name d for the "continuation channel" that will carry subsequent communications. The process $b \leftarrow$ recv a; P blocks until it receives a channel over a, binds it to the name b, and continues as P. Operationally, we rename a in P to the continuation channel d carrying the remainder of the communications.

$$\frac{\Pi\,;\,\Delta \vdash P :: a : A}{\Pi\,;\,\Delta, b : B \vdash \mathsf{send}\ a\ b;\ P :: a : B \otimes A}\ (\otimes R^*) \qquad \frac{\Pi\,;\,\Delta, a : A, b : B \vdash P :: c : C}{\Pi\,;\,\Delta, a : B \otimes A \vdash b \leftarrow \mathsf{recv}\ a;\ P :: c : C}\ (\otimes L)$$

$$\mathsf{proc}(a, \mathsf{send}\ a\ b;\ P), \mathbf{type}(a : B \otimes A) \to \exists d.\mathsf{proc}(d, [d/a]P), \mathsf{msg}(a, \mathsf{send}\ a\ b;\ a \leftarrow d), \mathbf{type}(d : A) \tag{6}$$

$$\mathsf{msg}(a, \mathsf{send}\ a\ e;\ a \leftarrow d), \mathsf{proc}(c, b \leftarrow \mathsf{recv}\ a;\ P) \to \mathsf{proc}(c, [e, d/b, a]Q) \tag{7}$$

The internal choice type $\oplus\{l : A_l\}_{l \in L}$ offers a choice of services A_l. The process $a.k; P$ sends a label k on a to signal its choice to provide the service A_k on a. The process case $a\ \{l \Rightarrow P_l\}_{l \in L}$ blocks until it receives a label $k \in L$ on a, and then continues as P_k.

$$\frac{\Pi\,;\,\Delta \vdash P :: a : A_k \quad (k \in L)}{\Pi\,;\,\Delta \vdash a.k; P :: a : \oplus\{l : A_l\}_{l \in L}}\ (\oplus R_k) \qquad \frac{\Pi\,;\,\Delta, a : A_l \vdash P_l :: c : C \quad (\forall l \in L)}{\Pi\,;\,\Delta, a : \oplus\{l : A_l\}_{l \in L} \vdash \mathsf{case}\ a\ \{l \Rightarrow P_l\}_{l \in L} :: c : C}\ (\oplus L)$$

$$\mathsf{proc}(a, a.k; P), \mathbf{type}(a : \oplus\{l : A_l\}_{l \in L}) \to \exists d.\mathsf{msg}(a, a.k; a \leftarrow d), \mathsf{proc}(d, [d/a]P), \mathbf{type}(d : A_k) \tag{8}$$

$$\mathsf{msg}(a, a.k; a \leftarrow d), \mathsf{proc}(c, \mathsf{case}\ a\ \{l \Rightarrow P_l\}_{l \in L}) \to \mathsf{proc}(c, [d/a]P_k) \tag{9}$$

To illustrate the duality between positive and negative types, we consider the (negative) external choice type. It is the polar dual of the (positive) internal choice type. The external choice type $\&\{l : A_l\}_{l \in L}$ provides a choice of services A_l. The process case $a\ \{l \Rightarrow P_l\}_{l \in L}$ blocks until it receives a label $k \in L$ on a, and then continues as P_k. The process $a.k; P$ sends a label k on a to signal its choice to use the service A_k on a. Observe that, where a provider of an internal choice type *sends* a label in (8), a provider of the external choice type *receives* a label in (10). Analogously, a client of an internal choice type receives *receives* a label in (9), and a client of an external choice type *sends* a label in (11).

$$\frac{\Psi\,;\,\Delta \vdash P_l :: a : A_l \quad (\forall l \in L)}{\Psi\,;\,\Delta \vdash \mathsf{case}\ a\ \{l \Rightarrow P_l\}_{l \in L} :: a : \&\{l : A_l\}_{l \in L}}\ (\&R) \qquad \frac{\Psi\,;\,\Delta, a : A_k \vdash P :: c : C \quad (k \in L)}{\Psi\,;\,\Delta, a : \&\{l : A_l\}_{l \in L} \vdash a.k; P :: c : C}\ (\&L_k)$$

$$\mathsf{msg}(a, a.k; a \leftarrow d), \mathsf{proc}(a, \mathsf{case}\ a\ \{l \Rightarrow P_l\}_{l \in L}) \to \mathsf{proc}(d, [d/a]P_k) \tag{10}$$

$$\mathsf{proc}(c, a.k; P), \mathbf{type}(a : \oplus\{l : A_l\}_{l \in L}) \to \exists d.\mathsf{msg}(a, a.k; a \leftarrow d), \mathsf{proc}(c, [d/a]P), \mathbf{type}(d : A_k) \tag{11}$$

A communication of type $\rho\alpha.A$ is an unfold message followed by a communication of type $[\rho\alpha.A/\alpha]A$. The process send a unfold; P sends an unfold message and continues as P. The process unfold $\leftarrow$ recv a; P

blocks until it receives the unfold message on a and continues as P.

$$\frac{\Pi\,;\,\Delta \vdash P :: a : [\rho\alpha.A/\alpha]A}{\Pi\,;\,\Delta \vdash \mathsf{send}\ a\ \mathsf{unfold}; P :: a : \rho\alpha.A}\ (\rho^+R) \qquad \frac{\Pi\,;\,\Delta, a : [\rho\alpha.A/\alpha]A \vdash P :: c : C}{\Pi\,;\,\Delta, a : \rho\alpha.A \vdash \mathsf{unfold} \leftarrow \mathsf{recv}\ a; P :: c : C}\ (\rho^+L)$$

$$\begin{aligned} &\mathsf{proc}(a, \mathsf{send}\ a\ \mathsf{unfold}; P), \mathbf{type}(a : \rho\alpha.A) \to \\ &\qquad \exists d.\mathsf{msg}(a, \mathsf{send}\ a\ \mathsf{unfold}; a \leftarrow d), \mathsf{proc}(d, [d/a]P), \mathbf{type}(d : [\rho\alpha.A/\alpha]A) \end{aligned} \tag{12}$$

$$\mathsf{msg}(a, \mathsf{send}\ a\ \mathsf{unfold}; a \leftarrow d), \mathsf{proc}(c, \mathsf{unfold} \leftarrow \mathsf{recv}\ a; P) \to \mathsf{proc}(c, [d/a]P) \tag{13}$$

Finally, recursive processes are formed in the standard way. The SSOS is only defined on closed processes, so there are no rules for process variables. Recursive processes step by unfolding.

$$\frac{}{\Pi, p : \{c : C \leftarrow \Delta\}\,;\,\Delta \vdash p :: c : C}\ (\textsc{Var}) \qquad \frac{\Pi, p : \{c : C \leftarrow \Delta\}\,;\,\Delta \vdash P :: c : C}{\Pi\,;\,\Delta \vdash \mathsf{fix}\ p.P :: c : C}\ (\textsc{Rec})$$

$$\mathsf{proc}(c, \mathsf{fix}\ p.P) \to \mathsf{proc}(c, [\mathsf{fix}\ p.P/p]P) \tag{14}$$

Example 4. *The protocol* $\mathsf{conat} = \rho\alpha.(\mathsf{z} : \mathbf{1}) \oplus (\mathsf{s} : \alpha)$ *encodes conatural numbers. Indeed, a communication is either an infinite sequence of successor labels* s*, or some finite number of* s *labels followed by the zero label* z *and termination. The following process receives a conatural number* i *and outputs its increment on* o*:*

$$\cdot\,;\, i : \mathsf{conat} \vdash \mathsf{send}\ o\ \mathsf{unfold}; s.o; o \to i :: o : \mathsf{conat}.$$

It works by outputting a successor label on o*, and then forwarding the conatural number* i *to* o*. It has the following fair execution, where we elide* $\mathbf{type}(c : A)$ *judgments and annotations on the arrows:*

$$\begin{aligned} &\mathsf{proc}(o, \mathsf{send}\ o\ \mathsf{unfold}; s.o; o \to i) \longrightarrow \mathsf{msg}(c, \mathsf{send}\ o\ \mathsf{unfold}; o \leftarrow o_1), \mathsf{proc}(o_1, s.o_1; o_1 \to i) \longrightarrow \\ &\qquad \mathsf{msg}(o, \mathsf{send}\ o\ \mathsf{unfold}; o \leftarrow o_1), \mathsf{msg}(o_1, s.o_1; s \leftarrow o_2), \mathsf{proc}(o_2, o_2 \leftarrow i). \end{aligned}$$

The following recursive process outputs the infinite conatural number $s(s(s(\cdots)))$ *on* o*:*

$$\cdot\,;\,\cdot \vdash \mathsf{fix}\ \omega.\mathsf{send}\ o\ \mathsf{unfold}; s.o; \omega :: o : \mathsf{conat}.$$

It has an infinite fair execution where for $n \geq 1$*, the rules* r_{3n-2}*,* r_{3n-1}*, and* r_{3n} *are respectively instantiations of* (14)*,* (12)*, and* (8)*.*

3.2 Properties of Process Traces

Let $\mathscr{P}$ be MRS given by the above rules. We prove various invariants maintained by process traces.

Let $\mathrm{fc}(P)$ be the set of free channel names in P. The following result follows by an induction on n and a case analysis on the rule used in the last step:

Proposition 8. *Let* $T = (M_0, (r_i; \delta_i)_i)$ *be a process trace. For all* n*, if* $\mathsf{proc}(c_0, P) \in M_n$*, then*

1. $c_0 \in \mathrm{fc}(P)$*;*
2. *for all* $c_i \in \mathrm{fc}(P)$*, there exists an* A_i *such that* $\mathbf{type}(c_i : A_i) \in M_n$*; and*
3. *where* $\mathrm{fc}(P) = \{c_0, \ldots, c_m\}$*, we have* $\cdot\,; c_1 : A_1, \ldots, c_m : A_m \vdash P :: c_0 : A_0$*.*

If $\mathsf{msg}(c, m) \in M_n$*, then*

- *if* $m = \mathsf{msg}(c, *)$*, then* $\mathbf{type}(c : \mathbf{1}) \in M_n$*;*

- *if* $m = c.l_j; c \leftarrow d$, *then either* $\mathbf{type}(c : \oplus\{l_i : A_i\}_{i\in I}) \in M_n$ *or* $\mathbf{type}(c : \&\{l_i : A_i\}_{i\in I}) \in M_n$ *for some* A_i $(i \in I)$, *and* $\mathbf{type}(d : A_j) \in M_n$ *for some* $j \in I$.
- *if* $m = \mathsf{send}\ c\ a;\ c \leftarrow b$, *then* $\mathbf{type}(c : A \otimes B), \mathbf{type}(a : A), \mathbf{type}(b : B) \in M_n$ *for some* A *and* B;
- *if* $m = \mathsf{send}\ c\ \mathsf{unfold}; c \leftarrow d$, *then* $\mathbf{type}(c : \rho\alpha.A), \mathbf{type}(d : [\rho\alpha.A/\alpha]A) \in M_n$ *for some* $\rho\alpha.A$.

The MRS $\mathscr{P}$ differs from the usual MRSs given for this style session-typed languages [13, 20, 25] in the addition of $\mathbf{type}(c : A)$ judgments. Corollary 3 shows that their addition does not change the operational behaviour of the semantics. Let $|M|$, $|\mathscr{P}|$, $|T|$, etc., be the result of erasing all $\mathbf{type}(c : A)$ judgments.

Corollary 3. *Consider a process* $\cdot\,; \Delta \vdash P :: c : A$ *with initial state* M_0. *If* T *is a trace from* M_0 *under* $\mathscr{P}$, *then* $|T|$ *is a trace from* $|M_0|$ *under* $|\mathscr{P}|$. *If* T *is a trace from* $|M_0|$ *under* $|\mathscr{P}|$, *then there exists a trace* T' *from* M_0 *under* $\mathscr{P}$ *such that* $|T'| = T$.

Proposition 8 showed that there were enough $\mathbf{type}(c : A)$ judgments in a trace. Proposition 9 shows that there are not too many:

Proposition 9. *Let* $(M_0, (r_i; \delta_i)_i)$ *be a process trace. For all channels* c *and all* $i, j \geq 0$, *if* $\mathbf{type}(c : A_i)$ *appears in* M_i *and* $\mathbf{type}(c : A_j)$ *appears in* M_j, *then* $A_i = A_j$.

We show an analogous uniqueness result for $\mathsf{msg}(c, m)$ judgments. It implies that each channel name in an execution carries at most one message. To prove it, we begin by partitioning a process's free channels into "input" and "output" channels and show that at all times, a channel is an output channel of at most one process. Given a process P, let $\mathrm{oc}(P)$ be the subset of $\mathrm{fc}(P)$ recursively defined by:

$$\begin{aligned}
\mathrm{oc}(a \to b) &= \{b\} & \mathrm{oc}(a \leftarrow P;\ Q) &= (\mathrm{oc}(P) \cup \mathrm{oc}(Q)) \setminus \{a\} \\
\mathrm{oc}(\mathsf{close}\ a) &= \{a\} & \mathrm{oc}(\mathsf{wait}\ a; P) &= \mathrm{oc}(P) \\
\mathrm{oc}(a.k; P) &= \{a\} \cup \mathrm{oc}(P) & \mathrm{oc}(\mathsf{case}\ a\ (l \Rightarrow P_l)_{l\in L}) &= \left(\bigcup_{l\in L} \mathrm{oc}(P_l)\right) \setminus \{a\} \\
\mathrm{oc}(\mathsf{send}\ a\ b;\ P) &= \{a\} \cup \mathrm{oc}(P) & \mathrm{oc}(b \leftarrow \mathsf{recv}\ a;\ P) &= \mathrm{oc}(P) \setminus \{a, b\} \\
\mathrm{oc}(\mathsf{send}\ a\ \mathsf{unfold}; P) &= \{a\} \cup \mathrm{oc}(P) & \mathrm{oc}(\mathsf{unfold} \leftarrow \mathsf{recv}\ a; P) &= \mathrm{oc}(P) \setminus \{a\} \\
\mathrm{oc}(p) &= \emptyset & \mathrm{oc}(\mathsf{fix}\ p.P) &= \mathrm{oc}(P)
\end{aligned}$$

Intuitively, $c \in \mathrm{oc}(P)$ if the next time P communicates on c, P sends a message on c. Given a configuration $\mathscr{C}$, let $\mathrm{oc}(\mathscr{C})$ be the union of the sets $\mathrm{oc}(P)$ for $\mathsf{proc}(c, P)$ in $\mathscr{C}$. Analogously, let $\mathrm{ic}(P)$ and $\mathrm{ic}(\mathscr{C})$ be the set of input channels of P and of $\mathscr{C}$.

Lemma 5. *If* $F(\vec{k}) \xrightarrow{(r;(\vec{k},\vec{a}))} G(\vec{k},\vec{a})$ *by a rule* r *of section 3.1, then*

- *if* $\mathsf{msg}(c, m) \in F(\vec{k})$, *then* $c \in \mathrm{ic}(F(\vec{k}))$;
- *if* $\mathsf{msg}(c, m) \in G(\vec{k},\vec{a})$, *then* $c \in \mathrm{oc}(F(\vec{k}))$;
- *if* $\mathsf{msg}(c, m; c \leftarrow d) \in G(\vec{k},\vec{a})$, *then* $d \in \vec{a}$ *and* $d \in \mathrm{fc}(G(\vec{k},\vec{a}))$; *and*
- $\mathrm{oc}(G(\vec{k},\vec{a})) \subseteq \mathrm{oc}(F,\vec{k}) \cup \vec{a}$ *and* $\mathrm{ic}(G(\vec{k},\vec{a})) \subseteq \mathrm{ic}(F,\vec{k}) \cup \vec{a}$.

Proof. Immediate by a case analysis on the rules. □

An induction with lemma 5 implies the desired disjointness result:

Lemma 6. *Let $(M_0,(r_i;\delta_i)_i)$ be a process trace. For all n, if $\mathsf{proc}(c,P)$ and $\mathsf{proc}(d,Q)$ appear in M_n, then $\mathrm{oc}(P)\cap\mathrm{oc}(Q)=\emptyset$ and $\mathrm{ic}(P)\cap\mathrm{ic}(Q)=\emptyset$.*

The following lemma shows that processes do not send messages on channels c already associated with a $\mathsf{msg}(c,m)$ judgment:

Lemma 7. *Let $(M_0,(r_i;\delta_i)_i)$ be a process trace. For all $n\leq k$, if $\mathsf{msg}(c,m)\in M_n$ and $\mathsf{proc}(d,P)\in M_k$, then $c\notin\mathrm{oc}(P)$.*

The desired result then follows by induction and the above results:

Corollary 4. *Let $(M_0,(r_i;\delta_i)_i)$ be a process trace. For all channels c and all $i,j\geq 0$, if $\mathsf{msg}(c,m_i)$ appears in M_i and $\mathsf{msg}(c,m_j)$ appears in M_j, then $m_i=m_j$.*

We now turn our attention to showing that all well-typed, closed processes have fair executions. This fact will follow easily from the following proposition:

Proposition 10. *The MRS $\mathscr{P}$ is non-overlapping from the initial configuration of $\cdot\,;\Delta\vdash P::c:A$ for all $\cdot\,;\Delta\vdash P::c:A$.*

Proof. Consider a trace $(M_0,(r_i;(\theta_i,\xi_i)))$ from the initial configuration of $\cdot\,;\Delta\vdash P::c:A$ and some arbitrary n. It is sufficient to show that if $s_1(\phi_1)$ and $s_2(\phi_2)$ are distinct instantiations applicable to M_n, then $F_1(\phi_1)$ and $F_2(\phi_2)$ are disjoint multisets: $F_1(\phi_1)\cap F_2(\phi_2)=\emptyset$. Indeed, if this is the case and $s_1(\phi_1),\ldots,s_k(\phi_k)$ are the distinct rule instantiations applications to M_n, then $F_1(\theta_1),\ldots,F_k(\phi_k)\subseteq M_n$, so $\Omega_{M_n}(F_1(\phi_1),\ldots,F_k(\phi_k))=\emptyset$.

We proceed by case analysis on the possible judgments in $F_1(\phi_1)\cap F_2(\phi_2)$.

Case $\mathsf{msg}(c,m)$**.** Then $c\in\mathrm{ic}(F_1(\phi_1))$ and $c\in\mathrm{ic}(F_2(\phi_2))$ by lemma 5. This is a contradiction by lemma 6.

Case $\mathsf{proc}(c,P)$**.** Then $s_1=s_2$ by a case analysis on the rules. We show that $\phi_1=\phi_2$. If s_1 is one of (2) to (6), (8), (11), (12) and (14), then we have $\phi_1=\phi_2$, because all constants matched by ϕ_1 and ϕ_2 appear in $\mathsf{proc}(c,P)$. If s_1 is one of (7), (9), (10) and (13), then $F_i(\phi_i)$ contain a judgment $\mathsf{msg}(d,m_i)$ where there is a constant $e_i\in m_i$ that appears in ϕ_i, but not in $\mathsf{proc}(c,P)$ (explicitly, e_i is the name of the continuation channel). By corollary 4, $m_1=m_2$, so $e_1=e_2$. All other channel names in ϕ_i appear in $\mathsf{proc}(c,P)$, so $\phi_1=\phi_2$. So $s_1(\phi_1)$ and $s_2(\phi_2)$ are not distinct rule instantiations, a contradiction.

Case type$(c:A)$**.** By case analysis on the rules, $s_1=s_2$ and there exist judgments $\mathsf{proc}(d_i,P_i)\in F_i(\phi_i)$. Suppose to the contrary that $P_1\neq P_2$. By case analysis on the rules, s_1 is one of (6), (8), (11) and (12). This implies that $c\in\mathrm{oc}(P_1)\cap\mathrm{oc}(P_2)$, a contradiction of lemma 6. So $P_1=P_2$. Because all constants in ϕ_1 and ϕ_2 appear in P_1, we conclude that $\phi_1=\phi_2$. So $s_1(\phi_1)$ and $s_2(\phi_2)$ are not distinct rule instantiations, a contradiction. □

Corollary 5. *Every process $\cdot\,;\Delta\vdash P::c:A$ has a fair execution. Its fair executions are all permutations of each other and they are all union-equivalent.*

Proof. By proposition 10, $\mathscr{P}$ is non-overlapping from the initial configuration M_0 of $\cdot\,;\Delta\vdash P::c:A$. It is then interference-free from M_0 by proposition 5, so a fair execution exists by proposition 3. All of its fair executions are permutations of each other by proposition 7. They are union-equivalent by corollary 2. □

4 Observed Communications

Consider a closed process $\cdot ; c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0$. In this section, we will define the observation of P to be a tuple $(c_i : v_i)_{0 \le i \le n}$, where v_i is the communication of type A_i observed on channel c_i in a fair execution of P. We extract communications from fair executions using a coinductively defined judgment. We colour-code the modes of judgments, where inputs to a judgment are in blue and outputs are in red.

We begin by defining session-typed communications. Let a communication v be a (potentially infinite) tree generated by the following grammar, where k and l_i range over labels. We explain these communications v below when we associate them with session types. For convenience, we also give a grammar generating the session types A of section 3.1. Session types are always finite expressions, and we treat $\rho\alpha.A$ as a binding operator.

$$\begin{aligned} v, v' &:= \bot_A \mid * \mid (k, v) \mid (v, v') \mid (\mathsf{unfold}, v) \\ A, A_i, B &:= \alpha \mid \mathbf{1} \mid A \otimes B \mid \oplus(l_1 : A_1, \ldots, l_n : A_n) \mid \&(l_1 : A_1, \ldots, l_n : A_n) \mid \rho\alpha.A. \end{aligned}$$

As in section 3.1, we abbreviate $\oplus(l_1 : A_1, \ldots, l_n : A_n)$ and $\&(l_1 : A_1, \ldots, l_n : A_n)$ by $\oplus\{l : A_l\}_{l \in L}$ and $\&\{l : A_l\}_{l \in L}$, respectively, where L is the finite set of labels.

Next, we associate communications with session types. The judgment $v \mathrel{\varepsilon} A$ means that the syntactic communication v has type A. It is coinductively defined by the following rules, where A is assumed to have no unbound occurrences of α. The rule forming $(k, v_k) \mathrel{\varepsilon} \oplus\{l : A_l\}_{l \in L}$ has the side condition $k \in L$.

$$\frac{}{\bot_{\mathbf{1}} \mathrel{\varepsilon} \mathbf{1}} \qquad \frac{}{* \mathrel{\varepsilon} \mathbf{1}} \qquad \frac{}{\bot_{A \otimes B} \mathrel{\varepsilon} A \otimes B} \qquad \frac{v \mathrel{\varepsilon} A \quad v' \mathrel{\varepsilon} B}{(v, v') \mathrel{\varepsilon} A \otimes B} \qquad \frac{}{\bot_{\rho\alpha.A} \mathrel{\varepsilon} \rho\alpha.A} \qquad \frac{v \mathrel{\varepsilon} [\rho\alpha.A/\alpha]A}{(\mathsf{unfold}, v) \mathrel{\varepsilon} \rho\alpha.A}$$

$$\frac{}{\bot_{\oplus\{l:A_l\}_{l \in L}} \mathrel{\varepsilon} \oplus\{l : A_l\}_{l \in L}} \qquad \frac{v_k \mathrel{\varepsilon} A_k}{(k, v_k) \mathrel{\varepsilon} \oplus\{l : A_l\}_{l \in L}} \qquad \frac{}{\bot_{\&\{l:A_l\}_{l \in L}} \mathrel{\varepsilon} \&\{l : A_l\}_{l \in L}} \qquad \frac{v_k \mathrel{\varepsilon} A_k}{(k, v_k) \mathrel{\varepsilon} \&\{l : A_l\}_{l \in L}}$$

Every closed session type A has an empty communication $\bot_A$ representing the absence of communication of that type. The communication $*$ represents the close message. A communication of type $\oplus\{l : A_l\}_{l \in L}$ or $\oplus\{l : A_l\}_{l \in L}$ is a label $k \in L$ followed by a communication v_k of type A_k, whence the communication (k, v_k). Though by itself the communication (k, v_k) does not capture the direction in which the label k travelled, this poses no problem to our development: we never consider communications without an associated session type, and the polarity of the type specifies the direction in which k travels. We cannot directly observe channels, but we can observe communications over channels. Consequently, we observe a communication of type $A \otimes B$ as a pair (v, v') of communications v of type A and v' of type B. A communication of type $\rho\alpha.A$ is an unfold message followed by a communication of type $[\rho\alpha.A/\alpha]A$.

Given a trace $T = (M_0, (r_i; (\theta_i, \xi_i))_i)$, we write $\mathscr{T}$ for the set-theoretic union of the M_i, that is, $x \in \mathscr{T}$ if and only if $x \in \mathrm{supp}(M_i)$ for some i. Write $T \vdash c : A$ if $\mathbf{type}(c : A) \in \mathscr{T}$. This judgment is defined on all channel names c that appear in T by proposition 8 and it is a function by proposition 9.

Assuming the channel c appears in T, the judgment $T \rightsquigarrow v \mathrel{\varepsilon} A \mathbin{/} c$ means that we observed a communication v of type A on the channel c during T. We will show below that whenever $T \rightsquigarrow v \mathrel{\varepsilon} A \mathbin{/} c$, we also have $T \vdash c : A$ and $v \mathrel{\varepsilon} A$. Fixing T, the judgment $T \rightsquigarrow v \mathrel{\varepsilon} A \mathbin{/} c$ is coinductively defined by the following rules, i.e., $T \rightsquigarrow v \mathrel{\varepsilon} A \mathbin{/} c$ is the largest set of triples (v, c, A) closed under the following rules.

We observe no communications on a channel c if and only if $\mathsf{msg}(c, m)$ does not appear in the trace for any m. Subject to the side condition that for all m, $\mathsf{msg}(c, m) \notin \mathscr{T}$, we have the rule

$$\frac{T \vdash c : A}{T \rightsquigarrow \bot_A \mathrel{\varepsilon} A \mathbin{/} c} \ (\text{O-}\bot)$$

We observe a close message on c if and only if the close message was sent on c:

$$\frac{\mathsf{msg}(c,*) \in \mathscr{T}}{T \rightsquigarrow * \;\varepsilon\; \mathbf{1} \;/\; c} \;(\text{O-}\mathbf{1})$$

We observe label transmission as labelling communications on the continuation channel. We rely on the judgment $T \vdash c : \oplus\{l : A_l\}_{l \in L}$ or $T \vdash c : \&\{l : A_l\}_{l \in L}$ to determine the type of c:

$$\frac{\mathsf{msg}(c, c.l; c \leftarrow d) \in \mathscr{T} \quad T \rightsquigarrow v \;\varepsilon\; A_l \;/\; d \quad T \vdash c : \oplus\{l : A_l\}_{l \in L}}{T \rightsquigarrow (l, v) \;\varepsilon\; \oplus\{l : A_l\}_{l \in L} \;/\; c} \;(\text{O-}\oplus)$$

$$\frac{\mathsf{msg}(c, c.l; c \leftarrow d) \in \mathscr{T} \quad T \rightsquigarrow v \;\varepsilon\; A_l \;/\; d \quad T \vdash c : \&\{l : A_l\}_{l \in L}}{T \rightsquigarrow (l, v) \;\varepsilon\; \&\{l : A_l\}_{l \in L} \;/\; c} \;(\text{O-}\&)$$

As described above, we observe channel transmission as pairing of communications:

$$\frac{\mathsf{msg}(c, \mathsf{send}\ c\ a;\ c \leftarrow d) \in \mathscr{T} \quad T \rightsquigarrow u \;\varepsilon\; A \;/\; a \quad T \rightsquigarrow v \;\varepsilon\; B \;/\; d}{T \rightsquigarrow (u, v) \;\varepsilon\; A \otimes B \;/\; c} \;(\text{O-}\otimes)$$

Finally, we observe the unfold message as an unfold message:

$$\frac{\mathsf{msg}(c, \mathsf{send}\ c\ \mathsf{unfold}; c \leftarrow d) \in \mathscr{T} \quad T \rightsquigarrow v \;\varepsilon\; [\rho\alpha.A/\alpha]A \;/\; d}{T \rightsquigarrow (\mathsf{unfold}, v) \;\varepsilon\; \rho\alpha.A \;/\; c} \;(\text{O-}\rho)$$

The following three propositions imply that for any T, $T \rightsquigarrow v \;\varepsilon\; A \;/\; c$ is a total function from channel names c in T to session-typed communications $v \;\varepsilon\; A$.

Proposition 11. *If $T \rightsquigarrow v \;\varepsilon\; A \;/\; c$, then $v \;\varepsilon\; A$.*

Proof. Immediate by rule coinduction. □

Proposition 12. *If T is a process trace, then for all c, if $T \vdash c : A$, then $T \rightsquigarrow v \;\varepsilon\; A \;/\; c$ for some v.*

Proof (Sketch). Let S be the set of all triples (v, A, c) for session-typed communications $v \;\varepsilon\; A$ and channel names c. Let $\Phi : \wp(S) \to \wp(S)$ be the rule functional defining $T \rightsquigarrow v \;\varepsilon\; A \;/\; c$. Then the judgment $T \rightsquigarrow v \;\varepsilon\; A \;/\; c$ is given by the greatest fixed point $\mathrm{gfp}(\Phi)$ of Φ on the complete lattice $\wp(S)$, where $T \rightsquigarrow v \;\varepsilon\; A \;/\; c$ if and only if $(v, A, c) \in \mathrm{gfp}(\Phi)$. The functional Φ is cocontinuous by [23, Theorem 2.9.4], so $\mathrm{gfp}(\Phi) = \bigcap_{n \geq 0} \Phi^n(S)$ by [23, Theorem 2.8.5]. It is sufficient to show that if $T \vdash c : A$, then there exists a v such that $(c, v, A) \in \Phi^n(S)$ for all n. This v can be constructed using a coinductive argument and a case analysis on $\mathsf{msg}(c, m) \in \mathscr{T}$. □

Proposition 13. *If T is a trace from the initial configuration of a process, then for all c, if $T \rightsquigarrow v \;\varepsilon\; A \;/\; c$ and $T \rightsquigarrow w \;\varepsilon\; B \;/\; c$, then $v = w$ and $A = B$.*

Proof (Sketch). Let $R = \{(T \rightsquigarrow v \;\varepsilon\; A \;/\; c, T \rightsquigarrow w \;\varepsilon\; B \;/\; c) \mid \exists v, w, c, A, B. T \rightsquigarrow v \;\varepsilon\; A \;/\; c \wedge T \rightsquigarrow w \;\varepsilon\; B \;/\; c\}$. We claim that R is a bisimulation. Indeed, let $(T \rightsquigarrow v \;\varepsilon\; A \;/\; c, T \rightsquigarrow w \;\varepsilon\; B \;/\; c) \in R$ be arbitrary. By corollary 4, at most one rule is applicable to form a judgment of the form $T \rightsquigarrow u \;\varepsilon\; C \;/\; c$ (with c fixed), so $T \rightsquigarrow v \;\varepsilon\; A \;/\; c$ and $T \rightsquigarrow w \;\varepsilon\; B \;/\; c$ were both formed by the same rule. A case analysis shows on this rule shows that R satisfies the definition of a bisimulation.

Consider arbitrary $T \rightsquigarrow v \;\varepsilon\; A \;/\; c$ and $T \rightsquigarrow w \;\varepsilon\; B \;/\; c$. They are related by R, so they are bisimilar. By [14, Theorem 2.7.2], bisimilar elements of the terminal coalgebra are equal, so $v = w$ and $A = B$. □

Corollary 6 gives the converse of proposition 12:

Corollary 6. *If T is a process trace, then for all c, if $T \rightsquigarrow v \;\varepsilon\; A \;/\; c$, then $T \vdash c : A$.*

Proof. We show by case analysis on the rules that if $T \rightsquigarrow v \;\varepsilon\; A \;/\; c$, then $T \vdash c : B$ for some B. The case (O-$\bot$) is obvious, while for each other case, if $T \rightsquigarrow v \;\varepsilon\; A \;/\; c$, then $\mathsf{msg}(c,m) \in \mathscr{T}$ for some m. For each of these cases, proposition 8 implies $\mathbf{type}(c : B) \in \mathscr{T}$ for some B, i.e., $T \vdash c : B$.

Assume $T \rightsquigarrow v \;\varepsilon\; A \;/\; c$. By the claim, $T \vdash c : B$ for some B. By proposition 12, there exists a w such that $T \rightsquigarrow w \;\varepsilon\; B \;/\; c$. By proposition 13, $A = B$, so $T \vdash c : A$. □

Theorem 1. *Let T be a fair execution of $\cdot\,; c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0$. For all $0 \leq i \leq n$, there exist unique v_i such that $v_i \;\varepsilon\; A_i$ and $T \rightsquigarrow v_i \;\varepsilon\; A_i \;/\; c_i$.*

Proof. By definition of fair execution, we have $\mathbf{type}(c_i : A_i) \in \mathscr{T}$ for all $0 \leq i \leq n$, i.e., $T \vdash c_i : A_i$ for all $0 \leq i \leq n$. By proposition 12, for all $0 \leq i \leq n$, there exists a v_i such that $T \rightsquigarrow v_i \;\varepsilon\; A_i \;/\; c_i$, and $v_i \;\varepsilon\; A_i$ by proposition 11. Each v_i is unique by proposition 13. □

The following theorem captures the confluence property typically enjoyed by SILL-style languages:

Theorem 2. *Let T and T' be a fair executions of $\cdot\,; c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0$. For all $0 \leq i \leq n$, if $T \rightsquigarrow v_i \;\varepsilon\; A_i \;/\; c_i$ and $T' \rightsquigarrow w_i \;\varepsilon\; A_i \;/\; c_i$, then $v_i = w_i$.*

Proof. Assume $T \rightsquigarrow v_i \;\varepsilon\; A_i \;/\; c_i$ and $T' \rightsquigarrow w_i \;\varepsilon\; A_i \;/\; c_i$. By corollary 5, traces T and T' are union-equivalent, i.e., $\mathscr{T} = \mathscr{T}'$. It immediately follows that $T' \rightsquigarrow w_i \;\varepsilon\; A_i \;/\; c_i$ if and only if $T \rightsquigarrow w_i \;\varepsilon\; A_i \;/\; c_i$. So $v_i = w_i$ by proposition 13. □

We use theorems 1 and 2 to define the **operational observation** $(\!|\cdot\,; c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0|\!)$ of $\cdot\,; c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0$. It is the tuple of observed communications

$$(\!|\cdot\,; c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0|\!) = (c_0 : v_0, \ldots, c_n : v_n)$$

where $T \rightsquigarrow v_i \;\varepsilon\; A_i \;/\; c_i$ for $0 \leq i \leq n$ for some fair execution T of $\cdot\,; c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0$. Such a T exists by corollary 5, and $(\!|\cdot\,; c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0|\!)$ does not depend on the choice of T by theorem 2. The v_i such that $T \rightsquigarrow v_i \;\varepsilon\; A_i \;/\; c_i$ exist by proposition 9, and they are unique by proposition 13.

Uniqueness of operational observations and theorem 2 crucially depend on fairness. Indeed, without fairness a process can have infinitely many observations. To see this, let Ω and B respectively be given by

$$\cdot\,; \cdot \vdash \mathsf{fix}\ \omega.\omega :: a : \mathbf{1}$$
$$\cdot\,; a : \mathbf{1} \vdash \mathsf{fix}\ p.\mathsf{send}\ b\ \mathsf{unfold}; b.l; p :: b : \rho\beta.\oplus\{l : \beta\}$$

Rule (3) is the first step of any execution of their composition $\cdot\,; \cdot \vdash a : \mathbf{1} \leftarrow \Omega;\ B :: b : \rho\beta.\oplus\{l : \beta\}$. It spawns Ω and B as separate processes. Without fairness, an execution could then consist exclusively of applications of rule (14) to Ω. This would give the observation $(b : \bot)$. Alternatively, B could take finitely many steps, leading to observations where b is a tree of correspondingly finite height. Fairness ensures that B and Ω both take infinitely many steps, leading to the unique observation $(b : (\mathsf{unfold}, (l, (\mathsf{unfold}, \ldots))))$.

Operational observation does not take into account the order in which a process sends on channels. For example, the following processes have the same operational observation $(a : (l, \bot_{\mathbf{1}}), b : (r, \bot_{\mathbf{1}}))$, even though they send on a and on b in different orders:

$$\cdot\,; a : \&\{l : \mathbf{1}\} \vdash a.l; b.r; a \to b :: b : \oplus\{r : \mathbf{1}\}$$
$$\cdot\,; a : \&\{l : \mathbf{1}\} \vdash b.r; a.l; a \to b :: b : \oplus\{r : \mathbf{1}\}.$$

The order in which channels are used does not matter for several reasons. First, messages are only ordered on a per-channel basis, and messages sent on different channels can arrive out of order. Second, each channel has a unique pair of endpoints, and the (CUT) rule organizes processes in a tree-like structure. This means that two processes communicating with a process R cannot at the same time also directly communicate with each other to compare the order in which R sent them messages. In other words, the ordering cannot be distinguished by other processes.

Our notion of operational observation scales to support language extensions. Indeed, for each new session type one first defines its corresponding session-typed communications. Then, one specifies how to observe message judgments $\mathsf{msg}(c,m)$ in a trace as communications. Informally, it seems desirable to ensure that if two message judgments $\mathsf{msg}(c,m)$ can be distinguished by a receiving process, then they are observed as different session-typed communications.

A **typed context** $\cdot\,;\Delta \vdash C[\cdot]^{\Delta'}_{a:A} :: b:B$ is a context derived using the process typing rules of section 3.1, plus exactly one instance of the axiom

$$\frac{}{\cdot\,;\Delta' \vdash [\cdot]^{\Delta'}_{a:A} :: a:A}\ (\textsc{Hole})$$

Given a context $\cdot\,;\Delta \vdash C[\cdot]^{\Delta'}_{a:A} :: b:B$ and a process $\cdot\,;\Delta' \vdash P :: a:A$, we let $\cdot\,;\Delta \vdash C[P] :: b:B$ be the result of "plugging" P into the hole, that is, of replacing the axiom (HOLE) by the derivation $\Delta' \vdash P :: a:A$ in the derivation $\Delta \vdash C[\cdot]^{\Delta'}_{a:A} :: b:B$.

We say that processes $\cdot\,;\Delta \vdash P :: c:C$ and $\cdot\,;\Delta \vdash Q :: c:C$ are **observationally congruent**, $P \approx Q$, if $(\!|\cdot\,;\Delta' \vdash C[P] :: b:B|\!) = (\!|\cdot\,;\Delta' \vdash C[Q] :: b:B|\!)$ for all typed contexts $\cdot\,;\Delta' \vdash C[\cdot]^{\Delta}_{c:C} :: b:B$. Intuitively, this means that no context C can differentiate processes P and Q.

To illustrate observational congruence, we show that process composition is associative:

Proposition 14. *We have* $c_1:C_1 \leftarrow P_1;\ (c_2:C_2 \leftarrow P_2;\ P_3) \approx c_2:C_2 \leftarrow (c_1:C_1 \leftarrow P_1;\ P_2);\ P_3$ *for all* $\cdot\,;\Delta_1 \vdash P_1 :: c_1:C_1$, *all* $\cdot\,;c_1:C_1,\Delta_2 \vdash P_2 :: c_2:C_2$, *and all* $\cdot\,;c_2:C_2,\Delta_3 \vdash P_3 :: c_3:C_3$.

Proof (Sketch). Let $L = c_1:C_1 \leftarrow P_1;\ (c_2:C_2 \leftarrow P_2;\ P_3)$ and $R = c_2:C_2 \leftarrow (c_1:C_1 \leftarrow P_1;\ P_2);\ P_3$. Consider some arbitrary observation context $C[\cdot]$ and a fair execution T of $C[L]$. It is sufficient to show that T agrees on message judgments with a fair execution $C[R]$. Union-equivalence of process traces is invariant under permutation, so we can assume without loss of generality that whenever $\mathsf{proc}(c_3,L)$ appears in some M_n of T, then the next two steps are applications (3) to decompose L:

$$\begin{aligned}\mathsf{proc}(c_3,L) \longrightarrow\ & \mathsf{proc}(c_1',[c_1'/c_1]P_1), \mathsf{proc}(c_3,[c_1'/c_1](c_2:C_2 \leftarrow P_2;\ P_3)) \longrightarrow \\ & \mathsf{proc}(c_1',[c_1'/c_1]P_1), \mathsf{proc}(c_2',[c_1',c_2'/c_1,c_2]P_2), \mathsf{proc}(c_3',[c_2'/c_2]P_3)\end{aligned}$$

(For conciseness, we elide the $\mathbf{type}(c:A)$ judgments.) There exists a fair execution T' of $C[R]$ that agrees with T on all steps, except for those involving R, where we make the same assumption:

$$\begin{aligned}\mathsf{proc}(c_3,R) \longrightarrow\ & \mathsf{proc}(c_2',[c_2'/c_2](c_1:C_1 \leftarrow P_1;\ P_2)), \mathsf{proc}(c_3,[c_2'/c_2]P_3) \longrightarrow \\ & \mathsf{proc}(c_1',[c_1'/c_1]P_1), \mathsf{proc}(c_2',[c_1',c_2'/c_1,c_2]P_2), \mathsf{proc}(c_3',[c_2'/c_2]P_3)\end{aligned}$$

So traces T and T' agree on all message judgments, whence $(\!|C[L]|\!) = (\!|C[R]|\!)$. □

5 Related Work

Multiset rewriting systems with existential quantification were first introduced by Cervesato et al. [8]. They were used to study security protocols and were identified as the first-order Horn fragment of linear

logic. Since, MRSs have modelled other security protocols, and strand spaces [7, 9]. Cervesato and Scedrov [10] studied the relationship between MRSs and linear logic. These works do not explore fairness.

Weak and strong fairness were first introduced by Apt and Olderog [1] and Park [18] in the context of do-od languages, and were subsequently adapted to process calculi, e.g., by Costa and Stirling [11] for Milner's CCS. Our novel notion of fairness for multiset rewriting systems in section 2 implies strong process fairness (so also weak process fairness) for the session-typed processes of section 3. We conjecture that this notion of fairness is stronger than required for many applications. In future work, we intend to explore other formulations of fairness for MRSs and their impact on applications.

Substructural operational semantics [24] based on multiset rewriting are widely used to specify the operational behaviour of session-typed languages arising from proofs-as-processes interpretations of linear logic and adjoint logic. Examples include functional languages with session-typed concurrency [25], languages with run-time monitoring [13], message-passing interpretations of adjoint logic [22], and session-typed languages with sharing [3]. The fragment of section 3.1 illustrates some of the key ideas of this approach, and extends to these richer settings.

Some of these languages are already equipped with observational equivalences. For example, Pérez et al. [19] introduced *typed context bisimilarity*, a labelled bisimilarity for session-typed processes. It does not support recursive processes or recursive session types. Toninho [27] explored barbed congruence for session-typed processes and shows that it coincides with logical equivalence. Kokke, Montesi, and Peressotti [16] showed that the usual notions of bisimilarity and barbed congruence carry over from the π-calculus. They also gave a denotational semantics using Brzozowski derivatives to "hypersequent classical processes" that built on Atkey's denotational semantics for CP, and showed that all three notions of equivalence agreed on well-typed programs. In future work, we intend to show that our observational congruence agrees with barbed congruence. Gommerstadt, Jia, and Pfenning [13] define a bisimulation-style observational equivalence on multisets in process traces. It deems two configurations equivalent if whenever both configurations send an externally visible message, then the messages are equivalent. It is easy to adapt this bisimulation to also require that one configuration sends an externally visible message if and only if the other does. We conjecture that this modified observational equivalence coincides with the one defined in section 4.

Session-typed languages enjoy other notions of process equivalence. Several session-typed languages are equipped with denotational semantics, and denotational semantics induce a compositional notion of program equivalence. For example, Castellan and Yoshida [6] gave a game semantics to a session-typed π-calculus with recursion, where session types denote event structures that encode games, and processes denote maps that encode strategies. Kavanagh [15] gave a domain-theoretic semantics to a full-featured functional language with session-typed message passing concurrency, where session types denote domains of communications and processes are continuous functions between these.

Atkey's observed communication semantics [2] for Wadler's CP [28] was motivated by two problems. Because CP uses a synchronous communication semantics, processes need partners to communicate with and get stuck if they try to communicate on a free channel. On the one hand, if processes have partners, then their communication are hidden by the (CUT) rule and cannot be observed, while on the other hand, if we leave the channels free, then we need to introduce reduction rules ("commuting conversions") for stuck processes, and these rules do not correspond to operationally justified communication steps. Atkey's elegant solution to this tension was to give communication partners to processes with free channels via closing "configurations", and then observing communications on these channels. Our task in section 4 is made easier by the fact that we use an asynchronous communication semantics. In our setting, a process can send messages on free channels, and we can observe these without having to provide it with communication partners via configurations. Atkey's observational equivalence and ours suffer from the

same weakness: to reason about observational equivalence, we must quantify over all observation contexts. Atkey addresses this by relating his semantics to a denotational semantics for CP and showing that they induce the same notion of equivalence. We are actively working on relating our OCS to Kavanagh's domain semantics [15]. Indeed, our OCS is largely motivated by efforts to relate denotational semantics of session-typed languages to their existing substructural operational semantics. We believe that our results on fair executions and their permutations should also simplify reasoning about observational equivalence.

6 Conclusion and Acknowledgements

We studied fair executions of multiset rewriting systems, and gave various conditions for an MRS to have fair executions. We used these results to define an observed communication semantics for session-typed languages that are defined by substructural operational semantics: the observation of a process is its communications on its free channels. Processes are then observationally equivalent if they cannot be distinguished through communication. We believe this work lays the foundation for future work on the semantics of session-typed processes, and in particular, we hope that it will be useful for exploring other notions of process equivalence.

The author thanks Stephen Brookes, Iliano Cervesato, Frank Pfenning, and the anonymous reviewers for their helpful comments.

References

[1] Krzysztof R. Apt & Ernst-Rüdiger Olderog (1982): *Proof Rules Dealing With Fairness*. Extended Abstract. In Dexter Kozen, editor: *Logics of Programs*, Logics of Programs Workshop, *Lecture Notes in Computer Science* 131, Springer-Verlag Berlin Heidelberg, pages 1–8, doi: 10.1007/BFb0025770.

[2] Robert Atkey (2017): *Observed Communication Semantics for Classical Processes*. In Hongseok Yang, editor: *Programming Languages and Systems*, 26th European Symposium on Programming (ESOP 2017), *Lecture Notes in Computer Science* 10201, Springer Berlin Heidelberg, Berlin, pages 56–82, doi: 10.1007/978-3-662-54434-1.

[3] Stephanie Balzer & Frank Pfenning (2017): *Manifest Sharing With Session Types*. *Proceedings of the ACM on Programming Languages* 1(ICFP):37, 29 pages, doi: 10.1145/3110281.

[4] Luís Caires & Frank Pfenning (2010): *Session Types as Intuitionistic Linear Propositions*. In Paul Gastin & François Laroussinie, editors: *CONCUR 2010 — Concurrency Theory*, 21st International Conference, CONCUR 2010, *Lecture Notes in Computer Science* 6269, Springer-Verlag Berlin Heidelberg, pages 222–236, doi: 10.1007/978-3-642-15375-4_16.

[5] Luís Caires, Frank Pfenning & Bernardo Toninho (2016): *Linear Logic Propositions As Session Types*. *Mathematical Structures in Computer Science*. *Behavioural Types Part 2* 26(3), pages 367–423, doi: 10.1017/s0960129514000218. Luís Caires & Frank Pfenning (2010): *Session Types as Intuitionistic Linear Propositions*. In Paul Gastin & François Laroussinie, editors: *CONCUR 2010 — Concurrency Theory*, 21st International Conference, CONCUR 2010, *Lecture Notes in Computer Science* 6269, Springer-Verlag Berlin Heidelberg, pages 222–236, doi: 10.1007/978-3-642-15375-4_16.

[6] Simon Castellan & Nobuko Yoshida (2019): *Two Sides of the Same Coin: Session Types and Game Semantics: A Synchronous Side and an Asynchronous Side*. *Proceedings of the ACM on Programming Languages* 3(POPL):27, 29 pages, doi: 10.1145/3290340.

[7] I. Cervesato, N. Durgin, M. Kanovich & A. Scedrov (2000): *Interpreting Strands in Linear Logic*. In *2000 Workshop on Formal Methods and Computer Security*.

[8] I. Cervesato, N. A. Durgin, P. D. Lincoln, J. C. Mitchell & A. Scedrov (1999): *A Meta-Notation for Protocol Analysis*. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, 12th IEEE Computer Security Foundations Workshop (CSFW'99), IEEE Computer Society, Los Alamitos, California, pages 55–69, doi: 10.1109/CSFW.1999.779762.

[9] Iliano Cervesato, Nancy A. Durgin, Patrick D. Lincoln, John C. Mitchell & Andre Scedrov (2005): *A Comparison Between Strand Spaces and Multiset Rewriting for Security Protocol Analysis*. *Journal of Computer Security* 13(2), pages 265–316, doi: 10.3233/JCS-2005-13203.

[10] Iliano Cervesato & Andre Scedrov (2009): *Relating State-Based and Process-Based Concurrency Through Linear Logic (full-Version)*. *Information and Computation*. *13th Workshop on Logic, Language, Information and Computation (WoLLIC 2006)* 207(10), pages 1044–1077, doi: 10.1016/j.ic.2008.11.006.

[11] Gerardo Costa & Colin Stirling (1987): *Weak and Strong Fairness in CCS*. *Information and Computation* 73(3), pages 207–244, doi: 10.1016/0890-5401(87)90013-7.

[12] Nissim Francez (1986): *Fairness*, xiii+295 pages. *Texts and Monographs in Computer Science*, Springer-Verlag New York Inc. doi: 10.1007/978-1-4612-4886-6.

[13] Hannah Gommerstadt, Limin Jia & Frank Pfenning (2018): *Session-Typed Concurrent Contracts*. In Amal Ahmed, editor: *Programming Languages and Systems*, 27th European Symposium on Programming (ESOP 2018), *Lecture Notes in Computer Science* 10801, Springer, Cham, pages 771–798, doi: 10.1007/978-3-319-89884-1.

[14] Bart Jacobs & Jan Rutten (2012): *An Introduction to (Co)algebra and (Co)induction*. In Davide Sangiorgi & Jan Rutten, editors: *Advanced Topics in Bisimulation and Coinduction*, pages 38–99. *Cambridge Tracts in Theoretical Computer Science* 52, Cambridge University Press, Cambridge, United Kingdom, doi: 10.1017/CBO9780511792588.003.

[15] Ryan Kavanagh (2020): *A Domain Semantics for Higher-Order Recursive Processes*, arXiv: 2002.01960v3 [cs.PL].

[16] Wen Kokke, Fabrizio Montesi & Marco Peressotti (2019): *Better Late Than Never: A Fully-Abstract Semantics for Classical Processes*. *Proceedings of the ACM on Programming Languages* 4(POPL):24, 29 pages, doi: 10.1145/3290337.

[17] Robin Milner (1980): *A Calculus of Communicating Systems*, vi+171 pages. *Lecture Notes in Computer Science* 92, Springer-Verlag Berlin Heidelberg. doi: 10.1007/978-3-540-38311-6.

[18] David Park (1982): *A Predicate Transformer for Weak Fair Iteration*. *RIMS Kôkyûroku* 454, pages 211–228, ISSN: 1880-2818, HDL: 2433/103001. Also appears in [21].

[19] Jorge A. Pérez, Luís Caires, Frank Pfenning & Bernardo Toninho (2014): *Linear Logical Relations and Observational Equivalences for Session-Based Concurrency*. *Information and Computation* 239, pages 254–302, doi: 10.1016/j.ic.2014.08.001.

[20] Frank Pfenning & Dennis Griffith (2015): *Polarized Substructural Session Types*. In Andrew Pitts, editor: *Foundations of Software Science and Computation Structures*, 18th International Conference, FOSSACS 2015, *Lecture Notes in Computer Science* 9034, Springer-Verlag GmbH Berlin Heidelberg, Berlin Heidelberg, pages 3–32, doi: 10.1007/978-3-662-46678-0_1.

[21] (1981): *Proceedings of the Sixth IBM Symposium on Mathematical Foundations of Computer Science: Logic Aspects of Programs*, 6th IBM Symposium on Mathematical Foundations of Computer Science, Corporate & Scientific Programs, IBM Japan, Tokyo, Japan.

[22] Klaas Pruiksma & Frank Pfenning (2019): *A Message-Passing Interpretation of Adjoint Logic*. In Francisco Martins & Dominic Orchard, editors: *Proceedings: Programming Language Approaches to Concurrency- and Communication-cEntric Software*, Programming Language Approaches to Concurrency- and Communication-cEntric Software (PLACES), *Electronic Proceedings in Theoretical Computer Science* 291, European Joint Conferences on Theory and Practice of Software, pages 60–79, doi: 10.4204/EPTCS.291.6, arXiv: 1904.01290v1 [cs.PL].

[23] Davide Sangiorgi (2012): *Introduction to Bisimulation and Coinduction*, xii+247 pages. Cambridge University Press, Cambridge, United Kingdom, doi: 10.1017/CBO9780511777110.

[24] Robert J. Simmons (2012): *Substructural Logical Specifications*, PhD thesis, xvi+300 pages. Computer Science Department, Carnegie Mellon University, Pittsburgh, Pennsylvania.

[25] Bernardo Toninho, Luis Caires & Frank Pfenning (2013): *Higher-Order Processes, Functions, and Sessions: A Monadic Integration*. In Matthias Felleisen & Philippa Gardner, editors: *Programming Languages and Systems*, 22nd European Symposium on Programming, ESOP 2013, *Lecture Notes in Computer Science* 7792, Springer Berlin Heidelberg, Berlin, Heidelberg, pages 350–369, doi: 10.1007/978-3-642-37036-6_20.

[26] Bernardo Toninho, Luís Caires & Frank Pfenning (2011): *Dependent Session Types Via Intuitionistic Linear Type Theory*. In *PPDP'11*, 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming (PPDP'11), Association for Computing Machinery, Inc., New York, New York, pages 161–172, doi: 10.1145/2003476.2003499.

[27] Bernardo Parente Coutinho Fernandes Toninho (2015): *A Logical Foundation for Session-based Concurrent Computation*, PhD thesis, xviii+178 pages. Universidade Nova de Lisboa.

[28] Philip Wadler (2014): *Propositions As Sessions*. *Journal of Functional Programming* 24(2-3), pages 384–418, doi: 10.1017/s095679681400001x.

Correctly Implementing Synchronous Message Passing in the Pi-Calculus By Concurrent Haskell's MVars

Manfred Schmidt-Schauß
Goethe-University, Frankfurt am Main, Germany
schauss@ki.cs.uni-frankfurt.de

David Sabel
LMU, Munich, Germany
david.sabel@ifi.lmu.de *

Comparison of concurrent programming languages and correctness of program transformations in concurrency are the focus of this research. As criterion we use contextual semantics adapted to concurrency, where may- as well as should-convergence are observed. We investigate the relation between the synchronous pi-calculus and a core language of Concurrent Haskell (CH). The contextual semantics is on the one hand forgiving with respect to the details of the operational semantics, and on the other hand implies strong requirements for the interplay between the processes after translation. Our result is that CH embraces the synchronous pi-calculus. Our main task is to find and prove correctness of encodings of pi-calculus channels by CH's concurrency primitives, which are MVars. They behave like (blocking) 1-place buffers modelling the shared-memory. The first developed translation uses an extra private MVar for every communication. We also automatically generate and check potentially correct translations that reuse the MVars where one MVar contains the message and two additional MVars for synchronization are used to model the synchronized communication of a single channel in the pi-calculus. Our automated experimental results lead to the conjecture that one additional MVar is insufficient.

1 Introduction

Our goals are the comparison of programming languages, correctness of transformations, compilation and optimization of programs, in particular of concurrent programs. We already used the contextual semantics of concurrent (functional) programming languages to effectively verify correctness of transformations [16, 23, 24], also under the premise not to worsen the runtime [30]. We propose to test may- and should-convergence in the contextual semantics, since, in particular, it rules out transformations that transform an always successful process into a process that may run into an error, for example a deadlock. There are also other notions of program equivalence in the literature, like bisimulation based program equivalences [27], however, these tend to take also implementation specific behavior of the operational semantics into account, whereas contextual equivalence abstracts from the details of the executions.

In [28, 31] we developed notions of correctness of translations w.r.t. contextual semantics, and in [32] we applied them in the context of concurrency, but for quite similar source and target languages. In this paper we translate a synchronous message passing model into a shared memory model, namely a synchronous π-calculus into a core-language of Concurrent Haskell, called *CH*.

The contextual semantics of concurrent programming languages is a generalization of the extensionality principle of functions. The test for a program P is whether $C[P]$ – i.e. P plugged into a program context – successfully terminates (converges) or not, which usually means that the standard reduction sequence ends with a value. For a concurrent program P, we use two observations: *may-convergence* ($P\downarrow$)

*This research is supported by the Deutsche Forschungsgemeinschaft (DFG) under grant SA2908/3-1

O. Dardha and J. Rot (Eds.): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics (EXPRESS/SOS 2020).
EPTCS 322, 2020, pp. 88–105, doi:10.4204/EPTCS.322.8

– at least one execution path terminates successfully, and *should-convergence*[1] ($P\Downarrow$) – every intermediate state of a reduction sequence may-converges. For two processes P and Q, $P \leq_c Q$ holds iff for all contexts $C[\cdot]$: $(C[P]\downarrow \implies C[Q]\downarrow)$, and P and Q are contextually equivalent, $P \sim_c Q$, iff $P \leq_c Q$ and $Q \leq_c P$. Showing equal expressivity of two (concurrent) calculi by a translation τ requires that may- and should-convergence make sense in each calculus. Important properties are convergence-equivalence (may- and should-convergencies are preserved and reflected by the translation) and adequacy (see Definition 4.4), which holds if $\tau(P) \leq_{c,CH} \tau(Q)$ implies $P \leq_{c,\pi} Q$, for all π-calculus processes P, Q. Full-abstraction, i.e. $\forall P, Q : \tau(P) \leq_c \tau(Q)$ iff $P \leq_c Q$, only holds if the two calculi are more or less the same.

Source and Target Calculi. The well-known π-calculus [15, 14, 27] is a minimal model for *mobile and concurrent processes*. Dataflow is expressed by passing messages between them via named channels, where messages are channel names. Processes and links between processes can be dynamically created and removed which makes processes mobile. The interest in the π-calculus is not only due to the fact that it is used and extended for various applications, like reasoning about cryptographic protocols [1], applications in molecular biology [21], and distributed computing [13, 7]. The π-calculus also permits the study of intrinsic principles and semantics of concurrency and the inherent nondeterministic behavior of mobile and communicating processes. We investigate a variant of the π-calculus which is the synchronous π-calculus with replication, but without sums, matching operators, or recursion. To observe termination of a process, the calculus has a constant `Stop` which signals successful termination.

The calculus *CH*, a core language of Concurrent Haskell, is a process calculus where threads evaluate expressions from a lambda calculus extended by data constructors, case-expressions, recursive let-expressions, and Haskell's seq-operator. Also monadic operations (sequencing and creating threads) are available. The shared memory is modelled by MVars (mutable variables) which are one-place buffers that can be either filled or empty. The operation `takeMVar` tries to empty a filled MVar and blocks if the MVar is already empty. The operation `putMVar` tries to fill an empty MVar and blocks if the MVar is already filled. The calculus *CH* is a variant (or a subcalculus) of the calculus *CHF* [23, 24] which extends Concurrent Haskell with futures. A technical advantage of this approach is that we can reuse studies and results on the contextual semantics of *CHF* also for *CH*.

Details and Variations of the Translation. One main issue for a correct translation from π-processes to *CH*-programs is to encode the synchronous communication of the π-calculus. The problem is that the MVars in *CH* have an asynchronous behavior (communication has to be implemented in two steps: the sender puts the message into an MVar, which is later taken by the receiver). To implement synchronous communication, the weaker synchronisation property of MVars has to be exploited, where we must be aware of the potential interferences of the executions of other translated communications on the same channel. The task of finding such translations is reminiscent of the channel-encoding used in [20], but, however, there an asynchronous channel is implemented while we look for synchronous communication.

We provide a translation τ_0 which uses a private MVar per channel and per communication to ensure that no other process can interfere with the interaction. A similar idea was used in [12, 3] for keeping channel names private in a different scenario (see [10, 9] for recent treatments of these encodings). We prove that the translation τ_0 is correct. Since we are also interested in simpler translations, we looked for correct translations with a fixed and static number of MVars per channel in the π-calculus. Since this

[1] An alternative observation is must-convergence (all execution paths terminate). The advantages of equivalence notions based on may- and should-convergence are invariance under fairness restrictions, preservation of deadlock-freedom, and equivalence of busy-wait and wait-until behavior (see e.g. [32]).

task is too complex and error-prone for hand-crafting, we automated it by implementing a tool to rule out incorrect translations[2]. Thereby we fix the MVars used for every channel: a single MVar for exchanging the channel-name and perhaps several additional MVars of unit type to perform checks whether the message was sent or received (we call them check-MVars, they behave like binary semaphores that are additionally blocking for signal-operations on an unlocked semaphore). The outcomes of our automated search are: a further correct translation that uses two check-MVars, where one is used as a mutex between all senders or receivers on one channel, and further correct translations using three additional MVars where the filling and emptying operations for each MVar need not come from the same sender or receiver. The experiments lead to the conjecture that there is no translation using only one check-MVar.

Results. Our novel result is convergence-equivalence and adequacy of the open translation τ (Theorems 4.5 and 4.8), translating the π-calculus into *CH*. The comparison of the π-calculus with a concurrent programming language (here *CH*) using contextual semantics for may- and should-convergence in both calculi exhibits that the π-calculus is embeddable in *CH* where we can prove that the semantical properties of interest are kept. The adaptation of the adequacy and full abstraction notions (Definition 4.4) for open processes is a helpful technical extension of our work in [28, 31].

We further define a general formalism for the representation of translations with global names and analyze different classes of such translations using an automated tool. In particular, we show correctness of two particular translations in Theorems 5.9 and 5.12. The discovered correct translations look quite simple and their correctness seems to be quite intuitive. However, our experience is that searching for correct translations is quite hard, since there are apparently correct (and simple) translations which were wrong. Our automated tool helped us to rule out wrong translations and to find potentially correct ones.

Discussion of Related Work on Characterizing Encodings. There are five criteria for valid translations resp. encodings proposed and discussed in [11, 9], which mainly restrict the translations w.r.t. language syntax and reduction semantics of the source and target language, called: compositionality, name invariance, operational correspondence, divergence reflection and success sensitiveness. Compositionality and name invariance restrict the syntactic form of the translated processes; operational correspondence means that the transitive closure of the reduction relation is transported by the translation, modulo the syntactic equivalence; and divergence reflection and success sensitiveness are conditions on the semantics.

In our approach, we define semantical congruence (and precongruence) relations on the source and target language. Thus the first two conditions are not part of our notion of contextual equivalence, however, may be used as restrictions in showing non-encodability. We also omit the third condition and only use stronger variants of the fourth and fifth condition. Convergence equivalence as a tool for finding out may-and should-convergence is our replacement of Gorla's divergence reflection and success sensitiveness. We do not define an infinite reduction sequence as an error, which has as nice consequence that synchronization could be implemented by busy-wait techniques.

Further Related Work. Encodings of synchronous communication by asynchronous communication using a private name mechanism are given in [12, 3] for (variants of the) π-calculus. Our idea of the translation τ_0 similarly uses a private MVar to encode the channel based communication, but our setting is different, since our target language is Concurrent Haskell. Encodings between π-calculi with synchronous and with asynchronous communication were, for instance, already considered in [12, 3, 19, 18]

[2]The tool and some output generated by the tool are available via https://gitlab.com/davidsabel/refute-pi.

$$P,Q \in \Pi_{\mathtt{Stop}} ::= \nu x.P \mid \bar{x}y.P \mid x(y).P \mid !P \mid P \mid Q \mid 0 \mid \mathtt{Stop}$$
$$C \in \Pi_{\mathtt{Stop},C} ::= [\cdot] \mid \bar{x}(y).C \mid x(y).C \mid C \mid P \mid P \mid C \mid !C \mid \nu x.C$$
$$\mathbb{D} \in PCtxt_{\pi} ::= [\cdot] \mid \mathbb{D} \mid P \mid P \mid \mathbb{D} \mid \nu x.\mathbb{D}.$$

Figure 1: Syntax of processes $\Pi_{\mathtt{Stop}}$, process contexts $\Pi_{\mathtt{Stop},C}$ and reduction contexts $PCtxt_{\pi}$ where x, y are names.

Interaction rule: (ia) $x(y).P \mid \bar{x}z.Q \xrightarrow{ia} P[z/y] \mid Q$

Closure: If $P \equiv \mathbb{D}[P'], P' \xrightarrow{ia} Q', \mathbb{D}[Q'] \equiv Q,$ and $\mathbb{D} \in PCtxt_{\pi}$ then $P \xrightarrow{sr} Q$

Figure 2: Reduction rule and standard reduction in $\Pi_{\mathtt{Stop}}$

$$\begin{aligned}
P &\equiv Q, \text{ if } P =_{\alpha} Q \\
P \mid (Q \mid R) &\equiv (P \mid Q) \mid R \\
\nu x.(P \mid Q) &\equiv P \mid \nu x.Q, \text{if } x \notin FN(P) \\
P \mid 0 &\equiv P \\
\nu x.0 &\equiv 0 \\
\nu x.\mathtt{Stop} &\equiv \mathtt{Stop} \\
\nu x.\nu y.P &\equiv \nu y.\nu x.P \\
P \mid Q &\equiv Q \mid P \\
!P &\equiv P \mid !P
\end{aligned}$$

Figure 3: Structural congruence in $\Pi_{\mathtt{Stop}}$

where encodability results are obtained for the π-calculus without sums [12, 3], while in [18, 19] the expressive power of synchronous and asynchronous communication in the π-calculus with *mixed sums* was compared and non-encodability is a main result. Translations of the π-calculus into programming calculi and logical systems are given in [2], where a translation into a graph-rewriting calculus is given and soundness and completeness w.r.t. the operational behavior is proved. The article [33] shows a translation and a proof that the π-calculus is exactly operationally represented. There are several works on session types which are related to the π-calculus, e.g., [17] studies encodings from a session calculus into PCF extended by concurrency and effects and also an embedding in the other direction, mapping PCF extended by effects into a session calculus. The result is a (strong) operational correspondence between the two calculi. In [4] an embedding of a session π-calculus into ReactiveML is given and operational correspondence between the two languages is shown. Encodings of CML-events in Concurrent Haskell using MVars are published in [22, 5]. This approach is more high-level than ours (since it considers events, while we focus the plain π-calculus). In [5] correctness of a distributed protocol for selective-communication w.r.t. an excerpt of CML is shown and a correct implementation of the protocol in the π-calculus is given. The protocol is implemented in Concurrent-Haskell, but no correctness of this part is shown, since [5] focuses to show that CML-events are implementable in languages with first-order message-passing which is different from our focus (translating the π-calculus into *CH*).

Outline. We introduce the source and target calculi in Sections 2 and 3, the translation using private names in Section 4, and in Section 5 we treat translations with global names. We conclude and discuss future work in Section 6. Due to space constraints most proofs are in the technical report [29].

2 The π-Calculus with Stop

We explain the synchronous π-calculus [15, 14, 27] without sums, with replication, extended with a constant `Stop` [25], that signals successful termination. The π-calculus with `Stop` and the π-calculus without `Stop` but with so-called barbed convergences [26] are equivalent w.r.t. contextual semantics [29]. Thus, adding the constant `Stop` is not essential, but our arguments are easier to explain with `Stop`.

Definition 2.1 (Calculus $\Pi_{\mathtt{Stop}}$)**.** *Let $\mathcal{N}$ be a countable set of (channel) names. The syntax of* processes *is shown in Fig. 1. Name restriction $\nu x.P$ restricts the scope of name x to process P, $P \mid Q$ is the parallel composition of P and Q, the process $\bar{x}y.P$ waits on channel x to output y over channel x and then becoming P, the process $x(y).P$ waits on channel x to receive input, after receiving the input z, the process turns*

into $P[z/y]$ (where $P[z/y]$ is the substitution of all free occurrences of name y by name z in process P), the process $!P$ is the replication of process P, i.e. it behaves like an infinite parallel combination of process P with itself, the process 0 is the silent process, and `Stop` *is a process constant that signals success. We sometimes write $x(y)$ instead of $x(y).0$ as well as $\bar{x}y$ instead of $\bar{x}y.0$.*

Free names $FN(P)$, bound names $BN(P)$, and α-equivalence $=_\alpha$ in $\Pi_{\mathtt{Stop}}$ are as usual in the π-calculus. A process P is closed *if $FN(P)=\emptyset$. Let $\Pi^c_{\mathtt{Stop}}$ be the closed processes in $\Pi_{\mathtt{Stop}}$. Structural congruence $\equiv$ is the least congruence satisfying the laws shown in Fig. 3.* Process contexts $\Pi_{\mathtt{Stop},C}$ *and* reduction contexts $PCtxt_\pi$ *are defined in Fig. 1. Let $C[P]$ be the substitution of the hole $[\cdot]$ in C by P. The reduction rule $\xrightarrow{ia}$ performs* interaction *and standard reduction $\xrightarrow{sr}$ is its closure w.r.t. reduction contexts and $\equiv$ (see Fig. 2). Let $\xrightarrow{sr,n}$ denote n $\xrightarrow{sr}$-reductions and $\xrightarrow{sr,*}$ denotes the reflexive-transitive closure of $\xrightarrow{sr}$. A process $P\in\Pi_{\mathtt{Stop}}$ is* successful, *if $P\equiv\mathbb{D}[\mathtt{Stop}]$ for some $\mathbb{D}\in PCtxt_\pi$.*

Remark 2.2. *We do not include "new" laws for structural congruences on the constant* `Stop`*, like* `Stop | Stop` *equals* `Stop`*, since this would require to re-develop a lot of theory known from the π-calculus without* `Stop`*. In our view,* `Stop` *is a mechanism for a notion of success that can be easily replaced by other similar notions (e.g. observing an open input or output as in barbed testing). However, it is easy to prove those equations* on the semantic level. *(i.e. w.r.t. $\sim_c$ as defined below in Definition 2.5).*

As an example for a reduction sequence, consider sending name y over channel x and then sending name u over channel y: $(x(z).\bar{z}u.0 \mid \bar{x}y.y(x).0) \xrightarrow{ia} (\bar{z}u.0[y/z] \mid y(x).0) \equiv (y(x).0 \mid \bar{y}u.0) \xrightarrow{ia} (0 \mid 0) \equiv 0$.

For the semantics of processes, we observe whether standard reductions successfully terminate or not. Since reduction is nondeterministic, we test whether there exists a successful reduction sequence (may-convergence), and we test whether all reduction possibilities are successful (should-convergence).

Definition 2.3. *Let P be a $\Pi_{\mathtt{Stop}}$-process. We say process P is* may-convergent *and write $P{\downarrow}$, if and only if there is a successful process P' with $P\xrightarrow{sr,*}P'$. We say P is* should-convergent *and write $P{\Downarrow}$ if and only if for all P': $P\xrightarrow{sr,*}P'$ implies $P'{\downarrow}$. If P is not may-convergent, then P we say is* must-divergent *(written $P{\Uparrow}$). If P is not should-convergent, then we say it is* may-divergent *(written $P{\uparrow}$).*

Example 2.4. *The process $P:=\nu x,y.(x(z).0 \mid \bar{x}y.\mathtt{Stop})$ is may-convergent ($P{\downarrow}$) and should-convergent ($P{\Downarrow}$), since $P\xrightarrow{sr}0 \mid \mathtt{Stop}$ is the only $\xrightarrow{sr}$-sequence for P. The process $P':=\nu x,y.(x(z).0 \mid \bar{x}y.0)$ is may- and must-divergent (i.e. $P'{\uparrow}$ and $P'{\Uparrow}$), since $P'\xrightarrow{sr}0$ is the only $\xrightarrow{sr}$-sequence for P'.*
For $P'':=\nu x,y.(\bar{x}y.0 \mid x(z).\mathtt{Stop} \mid x(z).0)$, we have $P''\xrightarrow{sr}\nu x,y.(\mathtt{Stop} \mid x(z).0)$ and $P''\xrightarrow{sr}\nu x,y.x(z).\mathtt{Stop}$. The first result is successful, and the second result is not successful. Hence, for P'' we have $P''{\downarrow}$ and $P''{\uparrow}$.

Should-convergence implies may-convergence, and must-divergence implies may-divergence.

Definition 2.5. *For $P,Q\in\Pi_{\mathtt{Stop}}$ and observation $\xi\in\{\downarrow,\Downarrow,\uparrow,\Uparrow\}$, we define $P\leq_\xi Q$ iff $P\xi \implies Q\xi$. The ξ-contextual preorders $\leq_{c,\xi}$ and then ξ-contextual equivalences $\sim_{c,\xi}$ are defined as*

$$P\leq_{c,\xi}Q \text{ iff } \forall C\in\Pi_{\mathtt{Stop},C}: C[P]\leq_\xi C[Q] \quad \text{and} \quad P\sim_{c,\xi}Q \text{ iff } P\leq_{c,\xi}Q\wedge Q\leq_{c,\xi}P$$

Contextual equivalence *of $\Pi_{\mathtt{Stop}}$-processes is defined as $P\sim_c Q$ iff $P\sim_{c,\downarrow}Q\wedge P\sim_{c,\Downarrow}Q$.*

Example 2.6. *For $Q:=\nu x,y.(\bar{x}y.0 \mid x(z).\mathtt{Stop} \mid x(z).0)$, we have $\mathtt{Stop}\sim_{c,\downarrow}Q$ (which can be proved using the methods in [25]), but $\mathtt{Stop}\not\sim_c Q$, since $\mathtt{Stop}{\Downarrow}$ and $Q{\uparrow}$ and thus $\mathtt{Stop}\not\sim_{c,\Downarrow}Q$. Note that $\leq_{c,\Downarrow}=\leq_c$ holds in $\Pi_{\mathtt{Stop}}$, since there is a context C such that for all processes P: $C[P]{\Downarrow}\iff P{\downarrow}$ (see [25]). For instance, the equivalence $0\sim_{c,\Downarrow}Q$ does not hold, since $!0{\Uparrow}$ and $!Q{\Downarrow}$ and thus the context $C=![\cdot]$ distinguishes 0 and Q w.r.t. should-convergence.*

Contextual preorder and equivalence are (pre)-congruences. Contextual preorder remains unchanged if observation is restricted to closing contexts:

Lemma 2.7. *Let $\xi\in\{\downarrow,\uparrow,\Downarrow,\Uparrow\}$, P,Q be $\Pi_{\mathtt{Stop}}$-processes. Then $P\leq_{c,\xi}Q$ if, and only if $\forall C\in\Pi_{\mathtt{Stop},C}$ such that $C[P]$ and $C[Q]$ are closed: $C[P]\leq_\xi C[Q]$.*

$$P \in Proc_{CH} ::= (P_1 \mid P_2) \mid \Leftarrow e \mid \nu x.P \mid x\,\mathbf{m}\,e \mid x\,\mathbf{m}- \mid x = e$$
$$e \in Expr_{CH} ::= x \mid \lambda x.e \mid (e_1\ e_2) \mid c\ e_1 \ldots e_{\mathrm{ar}(c)} \mid \mathtt{letrec}\ x_1{=}e_1, \ldots, x_n{=}e_n\ \mathtt{in}\ e \mid m \mid \mathtt{seq}\ e_1\ e_2$$
$$\mid \mathtt{case}_T\ e\ \mathtt{of}\ (c_{T,1}\ x_1 \ldots x_{\mathrm{ar}(c_{T,1})}\ \mathtt{->}\ e_1) \ldots (c_{T,|T|}\ x_1 \ldots x_{\mathrm{ar}(c_{T,|T|})}\ \mathtt{->}\ e_{|T|})$$
$$m \in MExpr_{CH} ::= \mathtt{return}\,e \mid e \gg\!= e' \mid \mathtt{forkIO}\,e \mid \mathtt{takeMVar}\,e \mid \mathtt{newMVar}\,e \mid \mathtt{putMVar}\,e\,e'$$
$$\mathrm{t} \in Typ_{CH} ::= \mathtt{IO}\ \mathrm{t} \mid (T\ \mathrm{t}_1 \ldots \mathrm{t}_n) \mid \mathtt{MVar}\ \mathrm{t} \mid \mathrm{t}_1 \to \mathrm{t}_2$$
$$\mathbb{D} \in PCtxt_{CH} ::= [\cdot] \mid \mathbb{D} \mid P \mid P \mid \mathbb{D} \mid \nu x.\mathbb{D} \qquad \mathbb{E} \in ECtxt_{CH} ::= [\cdot] \mid (\mathbb{E}\,e) \mid (\mathtt{seq}\,\mathbb{E}\,e) \mid (\mathtt{case}\,\mathbb{E}\,\mathtt{of}\,alts)$$
$$\mathbb{M} \in MCtxt_{CH} ::= [\cdot] \mid \mathbb{M} \gg\!= e \qquad \mathbb{F} \in FCtxt_{CH} ::= \mathbb{E} \mid (\mathtt{takeMVar}\,\mathbb{E}) \mid (\mathtt{putMVar}\,\mathbb{E}\,e)$$

Figure 4: Syntax of processes, expressions, types, and context classes of *CH*

Functional Evaluation:

(cpce) $\Leftarrow \mathbb{M}[\mathbb{F}[x]] \mid x = e \xrightarrow{sr} \Leftarrow \mathbb{M}[\mathbb{F}[e]] \mid x = e$

(mkbinds) $\Leftarrow \mathbb{M}[\mathbb{F}[\mathtt{letrec}\,x_1{=}e_1, \ldots, x_n{=}e_n\,\mathtt{in}\,e]] \xrightarrow{sr} \nu x_1 \ldots x_n.(\Leftarrow \mathbb{M}[\mathbb{F}[e]] \mid x_1{=}e_1 \mid \ldots \mid x_n{=}e_n)$

(beta) $\Leftarrow \mathbb{M}[\mathbb{F}[((\lambda x.e_1)\ e_2)]] \xrightarrow{sr} \Leftarrow \mathbb{M}[\mathbb{F}[e_1[e_2/x]]]$

(case) $\Leftarrow \mathbb{M}[\mathbb{F}[\mathtt{case}_T\ (c\ e_1 \ldots e_n)\ \mathtt{of}\ \ldots (c\ y_1 \ldots y_n\ \mathtt{->}\ e) \ldots]] \xrightarrow{sr} \Leftarrow \mathbb{M}[\mathbb{F}[e[e_1/y_1, \ldots, e_n/y_n]]]$

(seq) $\Leftarrow \mathbb{M}[\mathbb{F}[(\mathtt{seq}\ v\ e)]] \xrightarrow{sr} \Leftarrow \mathbb{M}[\mathbb{F}[e]]$ where v is a functional value

Monadic Computations:

(lunit) $\Leftarrow \mathbb{M}[\mathtt{return}\ e_1 \gg\!= e_2] \xrightarrow{sr} \Leftarrow \mathbb{M}[e_2\ e_1]$

(tmvar) $\Leftarrow \mathbb{M}[\mathtt{takeMVar}\ x] \mid x\,\mathbf{m}\,e \xrightarrow{sr} \Leftarrow \mathbb{M}[\mathtt{return}\ e] \mid x\,\mathbf{m}-$

(pmvar) $\Leftarrow \mathbb{M}[\mathtt{putMVar}\ x\ e] \mid x\,\mathbf{m}- \xrightarrow{sr} \Leftarrow \mathbb{M}[\mathtt{return}\ ()] \mid x\,\mathbf{m}\,e$

(nmvar) $\Leftarrow \mathbb{M}[\mathtt{newMVar}\ e] \xrightarrow{sr} \nu x.(\Leftarrow \mathbb{M}[\mathtt{return}\ x] \mid x\,\mathbf{m}\,e)$

(fork) $\Leftarrow \mathbb{M}[\mathtt{forkIO}\ e] \xrightarrow{sr} \Leftarrow \mathbb{M}[\mathtt{return}\ ()] \mid \Leftarrow e$

Closure w.r.t. $\mathbb{D}$-contexts and $\equiv$: If $P_1 \equiv \mathbb{D}[P_1']$, $P_2 \equiv \mathbb{D}[P_2']$, and $P_1' \xrightarrow{sr} P_2'$ then $P_1 \xrightarrow{sr} P_2$.

Capture avoidance: We assume capture avoiding reduction for all reductions.

Figure 5: Standard reduction rules of *CH* (call-by-name-version)

3 The Process Calculus *CH*

The calculus *CH* (a variant of the language *CHF*, [23, 24]) models a core language of Concurrent Haskell [20]. We assume a partitioned set of *data constructors* c where each family represents a type T. The data constructors of type T are $c_{T,1}, \ldots, c_{T,|T|}$ where each $c_{T,i}$ has an arity $\mathrm{ar}(c_{T,i}) \geq 0$. We assume that there is a type `()` with data constructor `()`, a type `Bool` with constructors `True`, `False`, a type `List` with constructors `Nil` and `:` (written infix), and a type `Pair` with a constructor $(,)$ written (a,b).

Processes $P \in Proc_{CH}$ in *CH* have expressions $e \in Expr_{CH}$ as subterms. See Fig. 4 where u, w, x, y, z are variables from an infinite set *Var*. Processes are formed by parallel composition " $\mid$ ". The ν-binder restricts the scope of a variable. A concurrent thread $\Leftarrow e$ evaluates e. In a process there is (at most one) unique distinguished thread, called *main thread*, written as $\overset{\mathsf{main}}{\Longleftarrow} e$. MVars are mutable variables which are empty or filled. A thread blocks if it wants to fill a filled MVar $x\,\mathbf{m}\,e$ or empty an empty MVar $x\,\mathbf{m}-$. Here x is called the *name of the MVar*. Bindings $x = e$ model the global heap, of shared expressions, where x is called a *binding variable*. If x is a name of an MVar or a binding variable, then x is called an *introduced variable*. In $Q \mid \nu x.P$ the scope of x is P. A process is *well-formed*, if all introduced variables are pairwise distinct and there exists at most one main thread $\overset{\mathsf{main}}{\Longleftarrow} e$.

Expressions $Expr_{CH}$ consist of functional and monadic expressions $MExpr_{CH}$. Functional expres-

sions are variables, *abstractions* $\lambda x.e$, *applications* $(e_1\ e_2)$, `seq`-*expressions* $(\texttt{seq}\ e_1\ e_2)$, *constructor applications* $(c\ e_1\ \ldots\ e_{\mathrm{ar}(c)})$, `letrec`-*expressions* $(\texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e)$, and $\texttt{case}_T$-*expressions* for every type T. We abbreviate `case`-expressions as $\texttt{case}_T\ e\ \texttt{of}\ alts$ where *alts* are the `case`-*alternatives* such that there is exactly one alternative $(c_{T,i}\ x_1 \ldots x_{\mathrm{ar}(c_{T,i})}\ \texttt{->}\ e_i)$ for every constructor $c_{T,i}$ of type T, where $x_1, \ldots, x_{\mathrm{ar}(c_{T,i})}$ (occurring in the *pattern* $c_{T,i}\ x_1 \ldots x_{\mathrm{ar}(c_{T,i})}$) are pairwise distinct variables that become bound with scope e_i. We often omit the type index T in $\texttt{case}_T$. In $\texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e$ the variables $x_1, \ldots, x_n$ are pairwise distinct and the bindings $x_i = e_i$ are recursive, i.e. the scope of x_i is $e_1, \ldots, e_n$ and e. *Monadic operators* `newMVar`, `takeMVar`, and `putMVar` are used to create, to empty and to fill MVars, the "bind"-operator `>>=` implements sequential composition of IO-operations, the `forkIO`-operator performs thread creation, and `return` lifts expressions into the monad.

Monadic values are $\texttt{newMVar}\,e$, $\texttt{takeMVar}\,e$, $\texttt{putMVar}\,e_1\,e_2$, $\texttt{return}\,e$, $e_1\ \texttt{>>=}\ e_2$, or $\texttt{forkIO}\,e$. *Functional values* are abstractions and constructor applications. A *value* is a functional or a monadic value.

Abstractions, `letrec`-expressions, `case`-alternatives, and $\nu x.P$ introduce variable binders. This induces bound and free variables (dentoted by $FV(\cdot)$), α-renaming, and α-equivalence $=_\alpha$. If $FV(P) = \emptyset$, then we call process P *closed*. We assume the *distinct variable convention*: free variables are distinct from bound variables, and bound variables are pairwise distinct. We assume that α-renaming is applied to obey this convention. Structural congruence $\equiv$ of *CH*-processes is the least congruence satisfying the laws $P_1 \mid P_2 \equiv P_2 \mid P_1$, $(P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3)$, $\nu x_1.\nu x_2.P \equiv \nu x_2.\nu x_1.P$, $P_1 \equiv P_2$ if $P_1 =_\alpha P_2$, and $(\nu x.P_1) \mid P_2 \equiv \nu x.(P_1 \mid P_2)$ if $x \notin FV(P_2)$.

We assume expressions and processes to be well-typed w.r.t. a monomorphic type system: the typing rules are standard (they can be found in [29]). The syntax of types is in Fig. 4 where $(\texttt{IO}\ \mathfrak{t})$ is the type of a monadic action with return type $\mathfrak{t}$, $(\texttt{MVar}\ \mathfrak{t})$ is the type of an `MVar` with content type $\mathfrak{t}$, and $\mathfrak{t}_1 \to \mathfrak{t}_2$ is a function type. We treat constructors like overloaded constants to use them in a polymorphic way.

We introduce a call-by-name small-step reduction for *CH*. This operational semantics can be shown to be equivalent to a call-by-need semantics (see [23] for the calculus *CHF*). However, the equivalence of the reduction strategies is not important for this paper. That is why we do not include it.

In *CH*, a context is a process or an expression with a (typed) hole $[\cdot]$. We introduce several classes of contexts in Fig. 4. They are used by the reduction rules.

Definition 3.1. *The standard reduction* $\xrightarrow{sr}$ *is defined by the rules and the closure in Fig. 5. It is only permitted for well-formed processes which are not successful.*

Functional evaluation includes β-reduction (beta), copying shared bindings into needed positions (cpce), evaluating `case`- and `seq`-expressions (case) and (seq), and moving `letrec`-bindings into the global bindings (mkbinds). For monadic computations, rule (lunit) implements the monadic evaluation. Rules (nmvar), (tmvar), and (pmvar) handle the MVar creation and access. A `takeMVar`-operation can only be performed on a filled MVar, and a `putMVar`-operation needs an empty MVar. Rule (fork) spawns a new thread. A concurrent thread of the form $\Leftarrow \texttt{return}\ e$ is terminated (where e is of type `()`).

Example 3.2. *The process* $\overset{\mathsf{main}}{\Longleftarrow} \texttt{newMVar}\,() \texttt{>>=} (\lambda y.\texttt{forkIO}\,(\texttt{takeMVar}\,y)) \texttt{>>=} \lambda\_.\texttt{putMVar}\,y\,()$ *creates a filled MVar, that is emptied by a spawned thread, and then again filled by the main thread.*

We say that a *CH*-process P is *successful* if $P \equiv \nu x_1 \ldots x_n.(\overset{\mathsf{main}}{\Longleftarrow} \texttt{return}\ e \mid P')$ and if P is well-formed. This captures Haskell's behavior that termination of the main-thread terminates all threads.

Definition 3.3. *Let P be a CH-process. Then P* may-converges *(denoted as $P{\downarrow}$), iff P is well-formed and $\exists P' : P \xrightarrow{sr,*} P'$ such that P' is successful. If $P{\downarrow}$ does not hold, then P* must-diverges *and we write $P{\Uparrow}$. Process P* should-converges *(written as $P{\Downarrow}$), iff P is well-formed and $\forall P' : P \xrightarrow{sr,*} P' \implies P'{\downarrow}$. If P is not should-convergent, then we say P* may-diverges *written as $P{\uparrow}$.*

$$
\begin{array}{ll}
\tau_0(P) & = \overset{\mathsf{main}}{\Longleftarrow} \mathbf{do}\ \{stop \leftarrow \texttt{newMVar}\ ();\texttt{forkIO}\ \tau(P);\texttt{putMVar}\ stop\ ()\} \\
\tau(\bar{x}y.P) & = \mathbf{do}\ \{checkx \leftarrow \texttt{newMVar}\ ();\texttt{putMVar}\ (unchan\ x)\ (y,checkx);\texttt{putMVar}\ checkx\ ();\tau(P)\} \\
\tau(x(y).P) & = \mathbf{do}\ \{(y,checkx) \leftarrow \texttt{takeMVar}\ (unchan\ x);\texttt{takeMVar}\ checkx;\tau(P)\}\} \\
\tau(P \mid Q) & = \mathbf{do}\ \{\texttt{forkIO}\ \tau(Q);\tau(P)\} \\
\tau(\nu x.P) & = \mathbf{do}\{chanx \leftarrow \texttt{newEmptyMVar};\texttt{letrec}\ x = \texttt{Chan}\, chanx\ \texttt{in}\ \tau(P)\} \\
\tau(0) & = \texttt{return}\ () \\
\tau(\texttt{Stop}) & = \texttt{takeMVar}\ stop \\
\tau(!P) & = \texttt{letrec}\ f = \mathbf{do}\ \{\texttt{forkIO}\ \tau(P);f\}\ \texttt{in}\ f
\end{array}
$$

Figure 6: Translations τ_0 and τ

Definition 3.4. *Contextual approximation $\leq_c$ and equivalence $\sim_c$ on CH-processes are defined as $\leq_c := \leq_{c,\downarrow} \cap \leq_{c,\Downarrow}$ and $\sim_c := \leq_c \cap \geq_c$ where $P_1 \leq_{c,\downarrow} P_2$ iff $\forall \mathbb{D} \in PCtxt_{CH} : \mathbb{D}[P_1]\downarrow \implies \mathbb{D}[P_2]\downarrow$ and $P_1 \leq_{c,\Downarrow} P_2$ iff $\forall \mathbb{D} \in PCtxt_{CH} : \mathbb{D}[P_1]\Downarrow \implies \mathbb{D}[P_2]\Downarrow$. For CH-*expressions*, let $e_1 \leq_c e_2$ iff for all process-contexts C with a hole at expression position: $C[e_1] \leq_c C[e_2]$ and $e_1 \sim_c e_2$ iff $e_1 \leq_c e_2 \wedge e_2 \leq_c e_1$.*

As an example, we consider the processes

$$
\begin{array}{lcl}
P_1 & := & \nu m.(\overset{\mathsf{main}}{\Longleftarrow} \texttt{takeMVar}\ m \mid \Leftarrow \texttt{takeMVar}\ m \mid m\,\mathbf{m}\,()) \\
P_2 & := & \overset{\mathsf{main}}{\Longleftarrow} \texttt{return}\ () \\
P_3 & := & \overset{\mathsf{main}}{\Longleftarrow} \texttt{letrec}\ x = x\ \texttt{in}\ x
\end{array}
$$

Process P_1 is may-convergent and may-divergent (and thus not should-convergent), since either the main-thread succeeds in emptying the MVar m, or (if the other threads empties the MVar m) the main-thread is blocked forever. The process P_2 is sucessful. The process P_3 is must-divergent. The equivalence $P_1 \sim_{c,\downarrow} P_2$ holds, but $P_1 \not\sim_c P_2$, since P_2 is should-convergent and thus $P_1 \not\sim_{c,\Downarrow} P_2$. As a further example, it is easy to verify that $P_1 \sim_{c,\Downarrow} P_3$ holds, since both processes are not should-convergent and a surrounding context cannot change this. However, $P_1 \not\sim_{c,\downarrow} P_3$, since $P_3\Uparrow$.

Contextual approximation and equivalence are (pre)-congruences. The following equivalence will help to prove properties of our translation.

Lemma 3.5. *The relations in Definition 3.4 are unchanged, if we add closedness: for $\xi \in \{\downarrow,\Downarrow\}$, let $P_1 \leq_{c,\xi} P_2$ iff $\forall \mathbb{D} \in PCtxt_{CH}$ such that $\mathbb{D}[P_1], \mathbb{D}[P_2]$ are closed: $\mathbb{D}[P_1]\xi \implies \mathbb{D}[P_2]\xi$.*

4 The Translation τ_0 with Private MVars

We present a translation τ_0 that encodes $\Pi_{\texttt{Stop}}$-processes as *CH*-programs. It establishes correct synchronous communication by using a private MVar, which is created by the sender and its name is sent to the receiver. The receiver uses it to acknowledge that the message was received. Since only the sender and the receiver know this MVar, no other thread can interfere the communication.The approach has similarities with Boudol's translation [3] from the π-calculus into an asynchronous one, where a private channel name of the π-calculus was used to guarantee safe communication between sender and receiver.

For translating π-calculus channels into *CH*, we use a recursive data type `Channel` (with constructor `Chan`), which is defined in Haskell-syntax as

$$\texttt{data Channel} = \texttt{Chan}\ (\texttt{MVar}\ (\texttt{Channel}, (\texttt{MVar}\ ())))$$

We abbreviate $(\texttt{case}_{\texttt{Chan}}\ e\ \texttt{of}\ (\texttt{Chan}\ m\ \texttt{->}\ m))$ as $(\mathit{unchan}\ e)$.

We use $a \gg b$ for $a \texttt{>>=} (\lambda\_.b)$ and also use Haskell's do-notation with the following meaning:

$$\begin{array}{ll} \textbf{do}\ \{x \leftarrow e_1; e_2\} = e_1 \texttt{>>=} \lambda x.(\textbf{do}\ \{e_2\}) & \textbf{do}\ \{e_1; e_2\} = e_1 \gg (\textbf{do}\ \{e_2\}) \\ \textbf{do}\ \{(x,y) \leftarrow e_1; e_2\} = e_1 \texttt{>>=} \lambda z.\texttt{case}_{\texttt{Pair}}\ z\ \texttt{of}\ (x,y)\ \texttt{->}\ (\textbf{do}\ \{e_2\}) & \textbf{do}\ \{e\} = e \end{array}$$

As a further abbreviation, we write $y \leftarrow \texttt{newEmptyMVar}$ inside a **do**-block to abbreviate the sequence $y \leftarrow \texttt{newMVar}\ \bot; \texttt{takeMVar}\ y$, where $\bot$ is a must-divergent expression. Our translation uses one MVar per channel which contains a pair consisting of the (translated) name of the channel and a further MVar used for the synchronization, which is private, i.e. only the sender and the receiver know it. Privacy is established by the sender: it creates a new MVar for every send operation. Message y is sent over channel x by sending a pair (y, check) where *check* is an MVar containing (). The receiver waits for a message (y, check) by the sender. After sending the message, the sender waits until *check* is emptied, and the receiver acknowledges by emptying the MVar *check*[3]

Definition 4.1. *We define the translation τ_0 and its inner translation τ from the $\Pi_{\texttt{Stop}}$-calculus into the CH-calculus in Fig. 6. For contexts, the translations are the same where the context hole is treated like a constant and translated as $\tau([\cdot]) = [\cdot]$.*

The translation τ_0 generates a main-thread and an MVar *stop*. The main thread is then waiting for the MVar *stop* to be emptied. The inner translation τ translates the constructs and constants of the $\Pi_{\texttt{Stop}}$-calculus into *CH*-expressions. Except for the main-thread (and using keyword `let` instead of `letrec`), the translation τ_0 generates a valid Concurrent Haskell-program, i.e. if we write $\tau_0(P) = \overset{\mathsf{main}}{\Longleftarrow} e$ as $\texttt{main} = e$, we can execute the translation in the Haskell-interpreter.

Example 4.2. *We consider the $\Pi_{\texttt{Stop}}$-process $P := \nu x, y_1, y_2, z.(x(y_1).0 \mid x(y_2).\texttt{Stop} \mid \bar{x}z.0)$ which is may-convergent and may-divergent: depending on which receiver communicates with the sender, the result is the successful process $\nu x, y_1.(x(y_1).0 \mid \texttt{Stop})$ or the must-divergent process $\nu x, y_2.(x(y_2).\texttt{Stop})$. The CH-process $\tau_0(P)$ reduces after several steps to the process*

$$\begin{array}{l} \nu \mathit{stop}, \mathit{chanx}, \mathit{chany}_1, \mathit{chany}_2, \mathit{chanz}, \mathit{checkx}, x, y_1, y_2, z.(\\ \quad \mathit{chanx}\,\mathbf{m}\,(z, \mathit{checkx}) \mid \mathit{chany}_1\,\mathbf{m}\,- \mid \mathit{chany}_2\,\mathbf{m}\,- \mid \mathit{chanz}\,\mathbf{m}\,- \mid \mathit{checkx}\,\mathbf{m}\,() \mid \mathit{stop}\,\mathbf{m}\,() \\ \quad \mid x{=}\texttt{Chan}\,\mathit{chanx} \mid z{=}\texttt{Chan}\,\mathit{chanz} \mid y_1{=}\texttt{Chan}\,\mathit{chany}_1 \mid y_2{=}\texttt{Chan}\,\mathit{chany}_2 \mid \overset{\mathsf{main}}{\Longleftarrow} \texttt{putMVar}\ \mathit{stop}\ () \\ \quad \mid \Leftarrow \textbf{do}\ \{\texttt{putMVar}\ \mathit{checkx}\ (); \texttt{return}\ ()\} \\ \quad \mid \Leftarrow \textbf{do}\ \{(y_1, \mathit{checkx}) \leftarrow \texttt{takeMVar}\ \mathit{chanx}; \texttt{takeMVar}\ \mathit{checkx}; \texttt{return}\ ()\} \\ \quad \mid \Leftarrow \textbf{do}\ \{(y_2, \mathit{checkx}) \leftarrow \texttt{takeMVar}\ \mathit{chanx}; \texttt{takeMVar}\ \mathit{checkx}; \texttt{takeMVar}\ \mathit{stop}\}) \end{array}$$

Now the first thread (which is the translation of sender $\bar{x}z.0$) is blocked, since it tries to fill the full MVar checkx. The second thread (the encoding of $x(y_1).0$) and the third thread (the encoding of $x(y_2).\texttt{Stop}$) race for emptying the MVar chanx. If the second thread wins, then it will fill the MVar checkx which is then emptied by the first thread, and all other threads are blocked forever. If the third thread wins, then it will fill the MVar checkx which is then emptied by the first thread, and then the second thread will empty the MVar stop. This allows the main-thread to fill it, resulting in a successful process.

For the following definition of τ being compositional, adequate, or fully abstract, we adopt the view that τ is a translation from $\Pi_{\texttt{Stop}}$ into the *CH*-language with a special initial evaluation context C^{τ}_{out}.

[3] A variant of the translation would be to change the roles for the acknowledgement such that an empty MVar is created, which is filled by the receiver and emptied by the sender. The reasoning on the correctness of the translation is very similar to the one presented here.

Definition 4.3. *Let* $C^{\tau}_{out} := \nu stop.\overset{\mathsf{main}}{\Longleftarrow}$ **do** $\{stop \leftarrow \texttt{newMVar}\ ();\texttt{forkIO}\ [\cdot];\texttt{putMVar}\ stop\ ()\}$. *Variants* $\downarrow_0,\Downarrow_0$ *of may- and should-convergence of expressions e within the context* C^{τ}_{out} *in CH are defined as* $e\downarrow_0$ *iff* $C^{\tau}_{out}[e]\downarrow$ *and* $e\Downarrow_0$ *iff* $C^{\tau}_{out}[e]\Downarrow$. *The relation* $\sim_{c,\tau_0}$ *is defined by* $\sim_{c,\tau_0} := \leq_{c,\tau_0} \cap \geq_{c,\tau_0}$, *where* $e_1 \leq_{c,\tau_0} e_2$ *iff* $\forall C$: *if* $FV(C[e_1],C[e_2]) \subseteq \{stop\}$, *then* $C[e_1]\downarrow_0 \implies C[e_2]\downarrow_0$ *and* $C[e_1]\Downarrow_0 \implies C[e_2]\Downarrow_0$.

Since $\leq_{c,CH}$ is a subset of $\leq_{c,\tau_0}$, we often can use the more general relations for reasoning.

Definition 4.4. *Let* $\Pi_{\texttt{Stop},C}$ *be the contexts of* $\Pi_{\texttt{Stop}}$. *We define the following properties for* τ_0 *and* τ *(see [31] for a general framework of properties of translations under observational semantics). For open processes* P,P', *we say that translation* τ *is*

convergence-equivalent *iff for all* $P \in \Pi_{\texttt{Stop}}$: $P\downarrow \iff \tau(P)\downarrow_0$ *and* $P\Downarrow \iff \tau(P)\Downarrow_0$,

compositional upto $\{\downarrow_0,\Downarrow_0\}$ *iff for all* $P \in \Pi_{\texttt{Stop}}$, *all* $C \in \Pi_{\texttt{Stop},C}$, *and all* $\xi \in \{\downarrow_0,\Downarrow_0\}$:
if $FV(C[P]) \subseteq \{stop\}$, *then* $\tau(C[P])\xi \iff \tau(C)[\tau(P)]\xi$,

adequate *iff for all processes* $P,P' \in \Pi_{\texttt{Stop}}$: $\tau(P) \leq_{c,\tau_0} \tau(P') \implies P \leq_c P'$, *and*

fully abstract *iff for all processes* $P,P' \in \Pi_{\texttt{Stop}}$: $P \leq_c P' \iff \tau(P) \leq_{c,\tau_0} \tau(P')$.

Convergence-equivalence of translation τ_0 for may- and should-convergence holds. For readability the proof is omitted, but given in the technical report [29], where we show:

Theorem 4.5. *Let* $P \in \Pi_{\texttt{Stop}}$ *be closed. Then* τ_0 *is convergence-equivalent for* $\downarrow$ *and* $\Downarrow$, *i.e.* $P\downarrow$ *is equivalent to* $\tau_0(P)\downarrow$. *and* $P\Downarrow$ *is equivalent to* $\tau_0(P)\Downarrow$. *This also shows convergence-equivalence of* τ *w.r.t.* $\downarrow_0,\Downarrow_0$, *i.e.* $P\downarrow \iff \tau(P)\downarrow_0$ *and* $P\Downarrow \iff \tau(P)\Downarrow_0$.

We show that the translation is adequate (see Theorem 4.8 below). The interpretation of this result is that the π-calculus with the concurrent semantics is semantically represented within *CH*. This result is on a more abstract level, since it is based on the property whether the programs (in all contexts) produce values or may run into failure, or get stuck; or not. Since the π-calculus does not have a notion of values, also the translated processes cannot be compared w.r.t. values other than a single form of value.

The translation τ_0 is not fully abstract (see Theorem 4.9 below), which is rather natural, since it only means that it is mapped into a subset of the *CH*-expressions and that this is a proper subset w.r.t. the semantics. For proving both theorems, we first use a simple form of a context lemma:

Lemma 4.6. *Let* e,e' *be CH-expressions, where the only free variable in* e,e' *is stop.*
Then $C^{\tau}_{out}[e] \leq_c C^{\tau}_{out}[e']$ *holds, if and only if* $C^{\tau}_{out}[e]\downarrow \implies C^{\tau}_{out}[e']\downarrow$ *and* $C^{\tau}_{out}[e]\Downarrow \implies C^{\tau}_{out}[e']\Downarrow$.

Proposition 4.7. *The translation* τ *is compositional upto* $\{\downarrow_0,\Downarrow_0\}$.

We show that the translation τ transports $\Pi_{\texttt{Stop}}$-processes into *CH*, such that adequacy holds. Thus the translated processes also correctly mimic the behavior of the original $\Pi_{\texttt{Stop}}$-processes when plugged into contexts. If the translated open processes cannot be distinguished by $\leq_{c,\tau_0}$, i.e. there is no test that detects a difference w.r.t. may- and should-convergence, then the original processes are equivalent in the π-calculus. However, this open translation is not fully abstract, which means that there are *CH*-contexts (not in the image of the translation) that can see and exploit too much of the details of the translation.

Theorem 4.8. *The translation* τ *is adequate.*

Proof. We prove the adequacy for the preorder $\leq_{c,\tau_0}$, for $\sim_{c,\tau_0}$ and $\sim_c$ the claim follows by symmetry. Let P,P' be $\Pi_{\texttt{Stop}}$-processes, such that $\tau(P) \leq_{c,\tau_0} \tau(P')$. We show that $P \leq_c P'$. We use Lemma 3.5 to restrict considerations to closed $C[P],C[P']$ below. Let C be a context in $\Pi_{\texttt{Stop}}$, such that $C[P],C[P']$ are closed and $C[P]\downarrow$. Then $\tau_0(C[P]) = C^{\tau}_{out}[\tau(C[P])]$. Closed convergence-equivalence implies $C^{\tau}_{out}[\tau(C[P])]\downarrow$. By Proposition 4.7. we have $C^{\tau}_{out}[\tau(C)[\tau(P)]]\downarrow$. Now $\tau(P) \leq_{c,\tau_0} \tau(P')$ implies $C^{\tau}_{out}[\tau(C)[\tau(P')]]\downarrow$, which is the same as $C^{\tau}_{out}[\tau(C[P'])]\downarrow$ using Proposition 4.7. Closed convergence-equivalence implies $C[P']\downarrow$. The same arguments hold for $\Downarrow$ instead of $\downarrow$. In summary, we obtain $P \leq_c P'$. □

Theorem 4.9. *The translation τ is not fully abstract, but it is fully abstract on closed processes, i.e. for closed processes $P_1, P_2 \in \Pi_{\mathtt{Stop}}$, we have $P_1 \leq_c P_2 \iff \tau(P_1) \leq_{c,\tau_0} \tau(P_2)$.*

Proof. The first part holds, since an open translation can be closed by a context without initializing the ν-bound MVars. For $P = \bar{x}(y).\mathtt{Stop} \mid x(z).\mathtt{Stop}$, we have $P \sim_c \mathtt{Stop}$ but $\tau(P) \not\sim_{c,0} \tau(\mathtt{Stop})$: let $\mathbb{D}$ be a context that does not initialize the MVars for x (as the translation does). Then $\mathbb{D}[\tau(P)]\Uparrow_0$, but $\mathbb{D}[\tau(\mathtt{Stop})]\Downarrow_0$. Restricted to closed processes, full abstraction holds: $P_1 \leq_c P_2 \Longrightarrow \tau(P_1) \leq_{c,\tau_0} \tau(P_2)$ follows from Lemma 4.6, since τ_0 produces closed processes in context C^{τ}_{out}. Theorem 4.8 implies the other direction. □

5 Translations with Global MVars

In this section we investigate translations that do not use private MVars, but use a fixed number of global MVars. We first motivate this investigation. The translation τ is quite complex and thus we want to figure out whether there are simpler translations. A further reason is that τ is not optimal, since it generates one MVar per communication which can be garbage-collected after the communication, however, generation and garbage collection require resources and thus the translation τ may be inefficient in practice.

To systematically search for small global translations we implemented an automated tool. It searches for translations with global MVars (abstracting from a lot of other aspects of the translation) and tries to refute the correctness. As we show, most of the small translations are shown as incorrect by our tool. Analyzing correctness of the remaining translations can then be done by hand.

We only consider the aspect of how to encode the synchronous message passing of the π-calculus, the other aspects (encoding parallel composition, replication and the `Stop`-constant) are not discussed and we assume that they are encoded as before (as the translation τ did). We also keep the main idea to translate a channel of the π-calculus into *CH* by representing it as an object of a user-defined data type `Channel` that consists of an MVar for transferring the message (which again is a `Channel`), and additional MVars for implementing a correct synchronization mechanism. For the translation τ, we used a private MVar (created by the sender, and transferred together with the message). Now we investigate translations where this mechanism is replaced by one or several public MVars, which are created once together with the channel object. To restrict the search space for translations, only the synchronization mechanism of MVars (by emptying and filling them) is used, but we forbid to transfer specific data (like numbers etc.). Hence, we restrict these MVars (which we call *check-MVars*) to be of type `MVar ()`. Such MVars are comparable to binary semaphores, where filling and emptying correspond to operations signal and wait. In summary, we analyze translations of π-calculus channels into a *CH*-data type `Channel` defined in Haskell-syntax as

```
data Channel = Chan (MVar Channel) (MVar ()) ... (MVar ())
```

A π-calculus channel x is represented as a *CH*-binding $x =$ `Chan` *content check*$_1$ ... *check*$_n$ where *content*, *check*$_1, \ldots,$ *check*$_n$ are appropriately initialized (i.e. empty) MVars. The MVars are public (or global), since all processes which know x have access to the components of the channel. After fixing this representation of a π-channel in *CH*, the task is to translate the input- and output-actions $x(y)$ and $\bar{x}z$ into *CH*-programs such that the interaction reduction is performed correctly and synchronously. We call the translation of $x(y)$, the *receiver* (program) and the translation of $\bar{x}z$ the *sender* (program). As a simplification, we restrict the allowed operations of the sender and receiver allowing only the operations:

$$
\begin{array}{ll}
\phi_{0,T}(P) & = \overset{\mathsf{main}}{\Longleftarrow} \mathbf{do}\ \{stop \leftarrow \texttt{newMVar}\ ();\texttt{forkIO}\ \phi_T(P);\texttt{putMVar}\ stop\ ()\} \\
\phi_T(\bar{x}y.P) & = \mathbf{do}\ \{T^{x,y}_{send};\phi_T(P)\} \\
\phi_T(x(y).P) & = \mathbf{do}\ \{T^{x,y}_{receive};\phi_T(P)\} \\
\phi_T(P \mid Q) & = \mathbf{do}\ \{\texttt{forkIO}\ \phi_T(Q);\phi_T(P)\} \\
\phi_T(\nu x.P) & = \mathbf{do}\{contx \leftarrow \texttt{newEmptyMVar};checkx_1 \leftarrow \texttt{newEmptyMVar};\ldots;checkx_n \leftarrow \texttt{newEmptyMVar}; \\
 & \qquad \texttt{letrec}\ x = \texttt{Chan}\, contx\ checkx_1 \ldots checkx_n \texttt{in}\ \phi_T(P)\} \\
\phi_T(0) & = \texttt{return}\ () \\
\phi_T(\texttt{Stop}) & = \texttt{takeMVar}\ stop \\
\phi_T(!P) & = \texttt{letrec}\ f = \mathbf{do}\ \{\texttt{forkIO}\ \phi_T(P);f\}\ \texttt{in}\ f
\end{array}
$$

Figure 7: Induced translations ϕ_T and $\phi_{0,T}$ where $T = (T_{send}, T_{recieve})$ uses n check-MVars

`putS`**:** The sender puts its message into the *contents*-MVar of the channel. It represents the expression $\texttt{case}_{\texttt{Channel}}\ x\ \texttt{of}\ (\texttt{Chan}\ c\ a_1\ \ldots\ a_n \texttt{->}\ \texttt{putMVar}\ c\ z \gg e)$ in *CH* where e is the remaining program of the sender. The operation occurs exactly once in the sender program. We write it as $\texttt{putS}_x\ z$, or as `putS`, if x and z are clear.

`takeS`**:** The receiver takes the message from the *contents*-MVar of channel x and replaces name y by the received name in the subsequent program. The operation occurs exactly once in the receiver program. We write it as $\texttt{takeS}_x\ y$, or as `takeS`, if x and y are clear. It represents the *CH*-expression $\texttt{case}_{\texttt{Channel}}\ x\ \texttt{of}\ (\texttt{Chan}\ c\ a_1\ \ldots\ a_n \texttt{->}\ \texttt{takeMVar}\ c \gg\texttt{=}\ \lambda y.e)$ where e is the remaining program of the receiver. In **do**-notation, we write $\mathbf{do}\ \{y \leftarrow \texttt{takeS}_x;e\}$ to abbreviate the above *CH*-expression.

`putC` **and** `takeC`**:** The sender and the receiver may synchronize on a check-MVar $check_i$ by putting $()$ into it or by emptying the MVar. These operations are written as $\texttt{putC}^i_x$ and $\texttt{takeC}^i_x$, or also as $\texttt{putC}^i, \texttt{takeC}^i$ if the name x is clear. We write `putC` and `takeC` if there is only one check-MVar. Let e be the remaining program of the sender or receiver. Then $\texttt{putC}^i_x$ represents the *CH*-expression $\texttt{case}_{\texttt{Channel}}\ x\ \texttt{of}\ (\texttt{Chan}\ c\ a_1\ \ldots\ a_n \texttt{->}\ \texttt{putMVar}\ a_i\ ()\ \gg e)$ and $\texttt{takeC}^i_x$ represents the expression $\texttt{case}_{\texttt{Channel}}\ x\ \texttt{of}\ (\texttt{Chan}\ c\ a_1\ \ldots\ a_n \texttt{->}\ \texttt{takeMVar}\ a_i \gg e)$.

We restrict our search for translations to the case that the sender and the receiver programs are sequences of the above operations, assuming that they are independent of the channel name x. With this restriction, we can abstractly write the translation of the sender and the receiver as a pair of sequences, where only $\texttt{putS}, \texttt{takeS}, \texttt{putC}^i$ and $\texttt{takeC}^i$ operations are used. We make some more restrictions:

Definition 5.1. *Let $n > 0$ be a number of check-MVars. A* standard global synchronized-to-buffer translation *(or* gstb-translation*) is a pair $(T_{send}, T_{receive})$ of a send-sequence T_{send} and a receive-sequence $T_{receive}$ consisting of* `putS`*,* `takeS`*,* $\texttt{putC}^i$ *and* $\texttt{takeC}^i$ *operations, where the send-sequence contains* `putS` *once, the receive-sequence contains* `takeS` *once, and for every* $\texttt{putC}^i$*-action in $(T_{send}, T_{receive})$, there is also a* $\texttt{takeC}^i$*-action in $(T_{send}, T_{receive})$. W.l.o.g., we assume that in the send-sequence the indices i are ascending. I.e. if* $\texttt{putC}^i$ *or* $\texttt{takeC}^i$ *is before* $\texttt{putC}^j$ *or* $\texttt{takeC}^j$*, then $i < j$ holds.*

We often say translation instead of gstb-translation, if this is clear from the context.

Definition 5.2. *Let $T = (T_{send}, T_{receive})$ be a gstb-translation. We write $T^{x,y}_{send}$ for the program T_{send} instantiated for $\bar{x}y$, i.e.* `putS` *is* $\texttt{putS}_x\ y$*, and all other operations are indexed with x. We write $T^{x,y}_{receive}$ for the program $T_{receive}$ instantiated for $x(y)$, i.e.* `takeS` *is* $\texttt{takeS}_x\ y$*, and all other operations are indexed with x. The* induced translations *$\phi_{0,T}$ and ϕ_T of $(T_{send}, T_{receive})$ are defined in Fig. 7.*

The induced translations are defined similar to the translations τ_0 and τ, where the differences are the representations of the channel. The translation of νx, $x(y)$, and $\bar{x}y$ is different, but the other cases remain the same. Since $\phi_{0,T}(P) = C^{\tau}_{out}[\phi_T(P)]$ and by the same arguments as in Theorem 4.8, we have:

Proposition 5.3. *If ϕ_T is closed convergence-equivalent, then ϕ_T is adequate.*

An *execution* of a translation $(T_{send}, T_{receive})$ for name x is the simulation of the *abstract* program, i.e. a program that starts with empty MVars $x, x_1, \ldots, x_n$, and is an interleaved sequence of actions from the send and receive-sequence T_{send} and $T_{receive}$, respectively.

To speak about the translations we make further classifications: We say that a translation allows *multiple uses*, if a check-MVar is used more than once, i.e. the sender or receiver may contain $\texttt{takeC}^i$ or $\texttt{putC}^i$ more than once for the same i. A translation has the *interprocess check restriction*, if for every i: $\texttt{takeC}^i$ and $\texttt{putC}^i$ do not occur both in T_{send}, and also not both in $T_{receive}$.

Definition 5.4. *A translation $T = (T_{send}, T_{receive})$ according to Definition 5.1 is*

- executable *if there is a deadlock free execution of T;*
- communicating*, if T_{send} contains at least one $\texttt{takeC}^i$-action;*
- overlap-free *if for a fixed name x, starting with empty MVars, every interleaved (concurrent) execution of $(T_{send}, T_{receive})$ cannot be disturbed by starting another execution of T_{send} and/or $T_{receive}$. More formally, let $((s_1;\ldots;s_i);(r_1;\ldots.r_j))$ and $((s'_1;\ldots;s'_i);(r'_1;\ldots.r'_j))$ be two copies of $(T_{send}, T_{receive})$ for a fixed name x. We call a command a_k from one of the four sequences, an a-action for $a \in \{s, s', r, r'\}$. The translation T is overlap-free if every execution of the four sequences has the property that it can be split into a prefix and a suffix (called parts in the following) such that one of the following properties holds*
  1. *One part contains only s- and r-actions, and the other part contains only s'- and r'-actions.*
  2. *One part contains only s- and r'-actions and the other part contains only s'- and r-actions.*

We implemented a tool to enumerate translations and to test whether each translation preserves and reflects may- and should-convergence for a (given) finite set of processes. Hence, our tool can refute the correctness of translations, but it can also output (usually small) sets of translations which are not refuted and which are promising candidates for correct translations. The above mentioned parallel execution of T_{send} and $T_{receive}$ is not sufficient to refute most of the translations, since it corresponds to the evaluation of the π-program $\nu x.(x(y) \mid \bar{x}z)$ (which is must-divergent). Thus, we apply the translation to a subset of π-processes, which we view as critical and for which we can automatically decide may- and should-convergence (before and after the translation). We consider only π-programs of the form $(\nu x_1, \ldots, x_n.P)$ where P contains only 0, `Stop`, parallel composition, and input- and output-prefixes. These programs are replication free and the ν-binders are on the top, and hence terminate. In the following we omit the ν-binders, but always mean them to be present. We also implemented techniques to generates all such programs until a bound on the size of the program is reached.

Our simulation tool[4] can execute all possible evaluations of those π-processes and – since all evaluation paths are finite – the tool can check for may- and should-convergence of the π-program. For the translated program, we do not generate a full *CH*-program, but generate a sequence of sequences of $\texttt{takeS}_x, \texttt{putS}_x, \texttt{takeC}^i_x, \texttt{putC}^i_x\ z$ and `Stop`-operations by applying the translation to all action prefixes in the π-program and by encoding `Stop` as `Stop`, 0 into an empty sequence. We get a sequence of sequences, since we have several threads and each thread is represented by one sequence. For executing the translated program, we simulate the global store (of MVars) and execute all possible interleavings

[4] Available via https://gitlab.com/davidsabel/refute-pi.

where we check for may- and should-convergence by looking whether the Stop eventually occurs at the beginning of the sequence. This simulates the behavior of the real *CH*-program in a controllable manner.

With the encoding of the sender- and receiver program and a π-calculus process P we

1. translate P with the encodings in the sequence of sequences consisting of $\texttt{takeS}_x$, $\texttt{putS}_x$, $\texttt{takeC}_x^i$, $\texttt{putC}_x^i\ z$ and Stop-operations;
2. simulate the execution on all interleavings;
3. test may- and should convergence of the original π-program P as well as the encoded program (w.r.t. the simulation);
4. compare the convergence before and after the translation. If there is a difference in the convergence behavior, then P is a counter-example for the correctness of the encodings.

Example 5.5. *Let us consider the gstb-translation* $(T_{send}, T_{receive}) = ([\texttt{takeC}, \texttt{putS}], [\texttt{putC}, \texttt{takeS}])$ *and the* π*-process* $P = \nu x, y, z, w(\bar{x}y.x(z).\texttt{Stop} \mid x(w).0)$. *Our tool recognizes that* $P\!\uparrow$ *and* $P\!\Uparrow$ *holds, since P reduces to the must-divergent process* $\nu x, z.(x(z).\texttt{Stop})$ *and there are no other reduction possibilities.*

Applying $(T_{send}, T_{receive})$ *to P yields the abstract program*

$$q := [[\texttt{takeC}_x, \texttt{putS}_x\ y, \texttt{putC}_x, \texttt{takeS}_x\ z, \texttt{Stop}], [\texttt{putC}_x, \texttt{takeS}_x\ w]].$$

For q, our tool inspects all executions. Among them there is the sequence

$$\texttt{putC}_x; \texttt{takeC}_x; \texttt{putS}_x\ y; \texttt{putC}_x; \texttt{takeS}_x\ z; \texttt{Stop}$$

which can be executed ending in Stop. *Thus q is may-convergent, and thus the process P is a counter-example that refutes the correctness of the translation.*

The case that there is no check-MVar leads to one possible translation $([\texttt{putS}], [\texttt{takeS}])$ which means that $\bar{x}z$ is translated into $\texttt{putS}_x\ y$ and $x(y)$ is translated into $\texttt{takeS}_x\ y$. This translation is not correct, since for instance the π-process $\bar{x}z.x(y).\texttt{Stop}$ is neither may- nor should-convergent, but the translation (written as an abstract program) is $[[\texttt{putS}_x\ z, \texttt{takeS}_x\ y, \texttt{Stop}]]$. I.e., it consists of one process which is may- and should-convergent (since $\texttt{putS}_x\ z; \texttt{takeS}_x\ y; \texttt{Stop}$ is the only evaluation sequence and its execution ends in Stop). Note that the translation into *CH* will generate two threads: the main threads that will wait until the MVar *stop* is filled, and a concurrent thread that will do the above operations.

5.1 Translations with Interprocess Check Restriction

We consider translations with the interprocess check restriction (each $\texttt{takeC}^i$ and $\texttt{putC}^i$ must be distributed between the sender and the receiver). There are $n! \cdot 2^n \cdot (n+1)^2$ different translations (for n check-MVars). For a single check-MVar, all 8 translations are rejected by our tool, Table 1 shows the translations and the obtained counter examples. For 2 check-MVars, our tool refutes all 72 translations. Compared to Table 1, there are two further π-programs used as counterexample. However, also the programs $\bar{x}y.\texttt{Stop} \mid x(y)$ and $\bar{x}y.x(z).\bar{z}q \mid x(z) \mid x(z) \mid \bar{x}z \mid y(u).\texttt{Stop}$ suffice to refute all 72 translations.

Theorem 5.6. *There is no valid gstb-translation with the interprocess check restriction for less than three check-MVars.*

A reason for the failure of translations with less than three check-MVars may be:

Theorem 5.7. *There is no executable, communicating, and overlap-free gstb-translation with the interprocess check restriction for* $n < 3$.

Translation (sender,receiver)	Counter-example (π-process)	$\downarrow$ before	$\Downarrow$ before	$\downarrow$ after	$\Downarrow$ after
$([\texttt{putC},\texttt{putS}],[\texttt{takeC},\texttt{takeS}])$	$\bar{x}y.x(y).\texttt{Stop}$	N	N	Y	Y
$([\texttt{putC},\texttt{putS}],[\texttt{takeS},\texttt{takeC}])$	$\bar{x}y.x(y).\texttt{Stop}$	N	N	Y	Y
$([\texttt{putS},\texttt{putC}],[\texttt{takeC},\texttt{takeS}])$	$\bar{x}y.x(y).\texttt{Stop}$	N	N	Y	Y
$([\texttt{putS},\texttt{putC}],[\texttt{takeS},\texttt{takeC}])$	$\bar{x}y.x(y).\texttt{Stop}$	N	N	Y	Y
$([\texttt{takeC},\texttt{putS}],[\texttt{putC},\texttt{takeS}])$	$\bar{x}y.x(z).\texttt{Stop} \mid x(w)$	N	N	Y	N
$([\texttt{takeC},\texttt{putS}],[\texttt{takeS},\texttt{putC}])$	$\bar{x}y.\texttt{Stop} \mid x(y)$	Y	Y	N	N
$([\texttt{putS},\texttt{takeC}],[\texttt{putC},\texttt{takeS}])$	$\bar{x}y.x(z).\texttt{Stop} \mid x(w)$	N	N	Y	N
$([\texttt{putS},\texttt{takeC}],[\texttt{takeS},\texttt{putC}])$	$\bar{x}z.\bar{z}a.\texttt{Stop} \mid \bar{x}w.\overline{w}a.\texttt{Stop} \mid x(y).y(u)$	Y	Y	Y	N

Table 1: Translations using one check-MVar and with the interprocess check restriction

Proof. For $n = 1$, we check the translations in Table 1. The first four are non-communicating. For the translation $([\texttt{takeC},\texttt{putS}],[\texttt{takeS},\texttt{putC}])$ a deadlock occurs. For $([\texttt{takeC},\texttt{putS}],[\texttt{putC},\texttt{takeS}])$, after putC, takeC, we can execute putC again. For $([\texttt{putS},\texttt{takeC}],[\texttt{takeS},\texttt{putC}])$, after executing putS, takeS we can execute putS again. For $([\texttt{putS},\texttt{takeC}],[\texttt{putC},\texttt{takeS}])$, after putC,putS, takeC we can execute putC again. For $n = 2$, the simulator finds no executable, communicating, and overlap-free translation: 18 translations are non-communicating, 21 lead to a deadlock, and 33 may lead to an overlap. □

For 3 MVars, our tool rejects 762 out of 768 translations (using the same counter examples as for 2 check-MVars) and the following 6 translations remain:

$$\mathfrak{T}_1 = ([\texttt{putS},\texttt{putC}^1,\texttt{takeC}^2,\texttt{putC}^3],[\texttt{takeC}^1,\texttt{putC}^2,\texttt{takeC}^3,\texttt{takeS}])$$
$$\mathfrak{T}_2 = ([\texttt{takeC}^1,\texttt{putS},\texttt{takeC}^2,\texttt{takeC}^3],[\texttt{putC}^3,\texttt{putC}^1,\texttt{takeS},\texttt{putC}^2])$$
$$\mathfrak{T}_3 = ([\texttt{putC}^1,\texttt{putS},\texttt{takeC}^2,\texttt{putC}^3],[\texttt{takeS},\texttt{putC}^2,\texttt{takeC}^3,\texttt{takeC}^1])$$
$$\mathfrak{T}_4 = ([\texttt{putC}^1,\texttt{putC}^2,\texttt{takeC}^3,\texttt{putS}],[\texttt{takeC}^2,\texttt{putC}^3,\texttt{takeS},\texttt{takeC}^1])$$
$$\mathfrak{T}_5 = ([\texttt{takeC}^1,\texttt{putS},\texttt{takeC}^2,\texttt{takeC}^3],[\texttt{putC}^1,\texttt{putC}^2,\texttt{takeS},\texttt{putC}^3])$$
$$\mathfrak{T}_6 = ([\texttt{putC}^1,\texttt{takeC}^2,\texttt{putS},\texttt{takeC}^3],[\texttt{takeC}^1,\texttt{putC}^2,\texttt{takeS},\texttt{putC}^3])$$

Proposition 5.8. *The translations $\mathfrak{T}_1,\mathfrak{T}_2,\mathfrak{T}_3$, and $\mathfrak{T}_4$ are executable, communicating, and overlap-free, whereas the translations $\mathfrak{T}_5$ and $\mathfrak{T}_6$ are executable, communicating, but overlapping.*

Proof. We only consider overlaps. For $\mathfrak{T}_1$ - $\mathfrak{T}_4$, only if all 8 actions are finished, the next send or receive can start. For $\mathfrak{T}_5,\mathfrak{T}_6$, after executing $\texttt{putC}^1$, $\texttt{takeC}^1$, we can again execute $\texttt{putC}^1$. □

In [29] we argue that the induced translation $\phi_{\mathfrak{T}_1}$ leaves may- and should-convergence invariant. The main help in reasoning is that there is no unintended interleaving of send and receive sequences according to Proposition 5.8. Application of Proposition 5.3 then shows:

Theorem 5.9. *Translation $\phi_{\mathfrak{T}_1}$ is adequate.*

For 4 MVars, our tool refutes 9266 and there remain 334 candidates for correct translations.

5.2 Dropping the Interprocess Check Restriction

We now consider gstb-translations without the interprocess check restriction, i.e. $\texttt{putC}^i$ and $\texttt{takeC}^i$ both may occur in the sender-program (or the receiver program, resp.). If we allow one check-MVar without reuse, then there are 20 candidates for translations. All are refuted by our simulation. Allowing reuse of the single check-MVar seems not to help to construct a correct translation: We simulated this for up to 6 uses, leading to 420420 candidates for a correct translation – our simulation refutes all of them.

Conjecture 5.10. *We conjecture that there is no correct translation for one check-MVar where re-uses are permitted and the interprocess check restriction is dropped, i.e., T_{send} is a word over* $\{\mathtt{putS}, \mathtt{putC}, \mathtt{takeC}\}$ *and $T_{receive}$ a word over* $\{\mathtt{takeS}, \mathtt{putC}, \mathtt{takeC}\}$*, where* $\mathtt{putS}, \mathtt{takeS}$ *occur exactly once.* [5]

For two MVars, one use and without the interprocess check restriction there are 420 translations. Our tool refutes all except for two: $\mathfrak{T}_7 = ([\mathtt{putC}^1, \mathtt{putS}, \mathtt{takeC}^2, \mathtt{takeC}^1], [\mathtt{takeS}, \mathtt{putC}^2])$ and $\mathfrak{T}_8 = ([\mathtt{takeC}^1, \mathtt{putS}], [\mathtt{putC}^2, \mathtt{putC}^1, \mathtt{takeS}, \mathtt{takeC}^2])$. In $\mathfrak{T}_7$ the second check-MVar is used as a mutex for the senders, ensuring that only one sender can operate at a time. $\mathfrak{T}_8$ does the same on the receiver side.

Proposition 5.11. *The translations $\mathfrak{T}_7, \mathfrak{T}_8$ are executable, communicating, overlap-free.*

Proof. The translations are executable and communicating. For $\mathfrak{T}_7$, $\mathtt{putC}^1, \mathtt{putS}$ and $\mathtt{takeS}$ are performed in this order. An additional sender cannot execute its first command before the original sender performs $\mathtt{takeC}^1$ and this again is only possible after the receiver program is finished. An additional receiver can only be executed after a $\mathtt{putS}$ is performed, which cannot be done by the current sender and receivers. For $\mathfrak{T}_8$, $\mathtt{putC}^2, \mathtt{putC}^1$ and $\mathtt{takeC}^1$ are performed in this order. An additional receiver can only start after $\mathtt{takeC}^2$ was executed by the original receiver, which can only occur after the original sender and receiver program are fully evaluated. An additional sender can only start after $\mathtt{putC}^1$ has been executed again, but the current sender and receiver cannot execute this command. □

The induced translation $\phi_{\mathfrak{T}_7}$ is (closed) convergence-equivalent [29]. With Proposition 5.3 this shows:

Theorem 5.12. *Translation $\phi_{\mathfrak{T}_7}$ is adequate.*

We are convinced that the same holds for $\mathfrak{T}_8$. We conclude the statistics of our search for translations without the interprocess restriction: For 3 MVars, there are 10080 translations and 9992 are refuted, i.e. 98 are potentially correct. One is $([\mathtt{putC}^1, \mathtt{putS}, \mathtt{takeC}^2, \mathtt{takeC}^1], [\mathtt{putC}^3, \mathtt{takeS}, \mathtt{putC}^2, \mathtt{takeC}^3])$ which is quite intuitive: check-MVar 1 is used as a mutex for all senders on the same channel, check-MVar 3 is used as a mutex for all receivers, and check-MVar 2 is used to send an acknowledgement. For 4 MVars, there are 277200 translations and 273210 are refuted, i.e. 3990 are potentially correct.

6 Discussion and Conclusion

We investigated translating the π-calculus into *CH* and showed correctness and adequacy of a translation τ_0 with private MVars for every translated communication. For translations with global names, we started an investigation on exhibiting (potentially) correct translations. We identified several minimal potentially correct translations and characterized all incorrect "small" translations. For two particular global translations, we have shown that they are convergence-equivalent and we proved their adequacy on open processes. The exact form of the translations were found by our tool to search for translations and to refute their correctness. The tool showed that there is no correct gstb-translation with the interprocess check restriction for less than 3 check-MVars. We also may consider extended variants of the π-calculus. We are convinced that adding recursion and sums can easily be built into the translation, while it might be challenging to encode mixed sums or (name) matching operators. For name matching operators, our current translation would require to test usual bindings in *CH* for equality which is not available in core-Haskell. Solutions may either use an adapted translation or a target language that supports observable sharing [6, 8]. The translation of mixed-sums into *CH* appears to require more complex translations, where the send- and receive-parts are not linear lists of actions.

[5] We already have a proof in the meantime, not yet published.

Acknowledgments We thank the anonymous reviewers for their valuable comments. In particular, we thank an anonymous reviewer for advises to improve the construction of translations, and for providing the counter-example in the last row of Table 1.

References

[1] Martín Abadi & Andrew D. Gordon (1997): *A Calculus for Cryptographic Protocols: The Spi Calculus*. In: *CCS 1997*, ACM, pp. 36–47, doi:10.1145/266420.266432.

[2] Richard Banach, J. Balázs & George A. Papadopoulos (1995): *A Translation of the Pi-Calculus Into MONSTR*. *J.UCS* 1(6), pp. 339–398, doi:10.3217/jucs-001-06-0339.

[3] Gérard Boudol (1992): *Asynchrony and the Pi-calculus*. Technical Report Research Report RR-1702,inria-00076939, INRIA, France. Available at `https://hal.inria.fr/inria-00076939`.

[4] Mauricio Cano, Jaime Arias & Jorge A. Pérez (2017): *Session-Based Concurrency, Reactively*. In: *FORTE 2017*, *LNCS* 10321, Springer, pp. 74–91, doi:10.1007/978-3-319-60225-7_6.

[5] Avik Chaudhuri (2009): *A concurrent ML library in concurrent Haskell*. In: *ICFP 2009*, ACM, pp. 269–280, doi:10.1145/1596550.1596589.

[6] Koen Claessen & David Sands (1999): *Observable Sharing for Functional Circuit Description*. In: *ASIAN 1999*, *LNCS* 1742, Springer, pp. 62–73, doi:10.1007/3-540-46674-6_7.

[7] Cédric Fournet & Georges Gonthier (2002): *The Join Calculus: A Language for Distributed Mobile Programming*. In: *APPSEM 2000*, *LNCS* 2395, Springer, pp. 268–332, doi:10.1007/3-540-45699-6_6.

[8] Andy Gill (2009): *Type-safe observable sharing in Haskell*. In: *Haskell 2009*, ACM, pp. 117–128, doi:10.1145/1596638.1596653.

[9] Rob van Glabbeek, Ursula Goltz, Christopher Lippert & Stephan Mennicke (2019): *Stronger Validity Criteria for Encoding Synchrony*. In: *The Art of Modelling Computational Systems: A Journey from Logic and Concurrency to Security and Privacy - Essays Dedicated to Catuscia Palamidessi on the Occasion of Her 60th Birthday*, *LNCS* 11760, Springer, pp. 182–205, doi:10.1007/978-3-030-31175-9_11.

[10] Rob J. van Glabbeek (2018): *On the validity of encodings of the synchronous in the asynchronous π-calculus*. *Inf. Process. Lett.* 137, pp. 17–25, doi:10.1016/j.ipl.2018.04.015.

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

[12] Kohei Honda & Mario Tokoro (1991): *An Object Calculus for Asynchronous Communication*. In: *ECOOP 1991*, Springer-Verlag, pp. 133–147, doi:10.1007/BFb0057019.

[13] Cosimo Laneve (1996): *On testing equivalence: May and Must Testing in the Join-Calculus*. Technical Report UBLCS 96-04, University of Bologna. Available at `https://www.cs.unibo.it/~laneve/papers/laneve96may.pdf`.

[14] Robin Milner (1999): *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press.

[15] Robin Milner, Joachim Parrow & David Walker (1992): *A Calculus of Mobile Processes, I & II*. *Inform. and Comput.* 100(1), pp. 1–77, doi:10.1016/0890-5401(92)90008-4.

[16] Joachim Niehren, David Sabel, Manfred Schmidt-Schauß & Jan Schwinghammer (2007): *Observational Semantics for a Concurrent Lambda Calculus with Reference Cells and Futures*. *Electron. Notes Theor. Comput. Sci.* 173, pp. 313–337, doi:10.1016/j.entcs.2007.02.041.

[17] Dominic A. Orchard & Nobuko Yoshida (2016): *Effects as sessions, sessions as effects*. In: *POPL 2016*, ACM, pp. 568–581, doi:10.1145/2837614.2837634.

[18] Catuscia Palamidessi (1997): *Comparing the Expressive Power of the Synchronous and the Asynchronous pi-calculus*. In: *POPL 1997*, ACM Press, pp. 256–265, doi:10.1145/263699.263731.

[19] Catuscia Palamidessi (2003): *Comparing The Expressive Power Of The Synchronous And Asynchronous Pi-Calculi*. *Math. Structures Comput. Sci.* 13(5), pp. 685–719, doi:10.1017/S0960129503004043.

[20] Simon L. Peyton Jones, Andrew Gordon & Sigbjorn Finne (1996): *Concurrent Haskell*. In: *POPL 1996*, ACM, pp. 295–308, doi:10.1145/237721.237794.

[21] Corrado Priami (1995): *Stochastic pi-Calculus*. *Comput. J.* 38(7), pp. 578–589, doi:10.1093/comjnl/38.7.578.

[22] George Russell (2001): *Events in Haskell, and How to Implement Them*. In: *ICFP 2001*, ACM, pp. 157–168, doi:10.1145/507635.507655.

[23] David Sabel & Manfred Schmidt-Schauß (2011): *A contextual semantics for Concurrent Haskell with futures*. In: *PPDP 2011*, ACM, pp. 101–112, doi:10.1145/2003476.2003492.

[24] David Sabel & Manfred Schmidt-Schauß (2012): *Conservative Concurrency in Haskell*. In: *LICS 2012*, IEEE, pp. 561–570, doi:10.1109/LICS.2012.66.

[25] David Sabel & Manfred Schmidt-Schauß (2015): *Observing Success in the Pi-Calculus*. In: *WPTE 2015*, *OASICS* 46, pp. 31–46, doi:10.4230/OASIcs.WPTE.2015.31.

[26] Davide Sangiorgi & David Walker (2001): *On Barbed Equivalences in pi-Calculus*. In: *CONCUR 200*, *LNCS* 2154, Springer, pp. 292–304, doi:10.1007/3-540-44685-0_20.

[27] Davide Sangiorgi & David Walker (2001): *The π-calculus: a theory of mobile processes*. Cambridge university press.

[28] Manfred Schmidt-Schauß, Joachim Niehren, Jan Schwinghammer & David Sabel (2008): *Adequacy of Compositional Translations for Observational Semantics*. In: *IFIP TCS 2008*, *IFIP* 273, Springer, pp. 521–535, doi:10.1007/978-0-387-09680-3_35.

[29] Manfred Schmidt-Schauß & David Sabel (2020): *Embedding the Pi-Calculus into a Concurrent Functional Programming Language*. Frank report 60, Institut für Informatik. Fachbereich Informatik und Mathematik. J. W. Goethe-Universität Frankfurt am Main. Available at `http://www.ki.informatik.uni-frankfurt.de/papers/frank/frank-60v5.pdf`.

[30] Manfred Schmidt-Schauß, David Sabel & Nils Dallmeyer (2018): *Sequential and Parallel Improvements in a Concurrent Functional Programming Language*. In: *PPDP 2018*, ACM, pp. 20:1–20:13, doi:10.1145/3236950.3236952.

[31] Manfred Schmidt-Schauß, David Sabel, Joachim Niehren & Jan Schwinghammer (2015): *Observational program calculi and the correctness of translations*. *Theor. Comput. Sci.* 577, pp. 98–124, doi:10.1016/j.tcs.2015.02.027.

[32] Jan Schwinghammer, David Sabel, Manfred Schmidt-Schauß & Joachim Niehren (2009): *Correctly translating concurrency primitives*. In: *ML 2009*, ACM, pp. 27–38, doi:10.1145/1596627.1596633.

[33] Ping Yang, C. R. Ramakrishnan & Scott A. Smolka (2004): *A logical encoding of the pi-calculus: model checking mobile processes using tabled resolution*. *STTT* 6(1), pp. 38–66, doi:10.1007/s10009-003-0136-3.