EPTCS 255

Proceedings of the

Combined 24th International Workshop on Expressiveness in Concurrency

and 14th Workshop on Structural Operational Semantics

Berlin, Germany, 4th September 2017

Edited by: Kirstin Peters and Simone Tini

Published: 31st August 2017
DOI: 10.4204/EPTCS.255
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Table of Contents i

Preface ii

Sequential Composition in the Presence of Intermediate Termination (Extended Abstract) 1
Jos Baeten, Bas Luttik and Fei Yang

Analysing Mutual Exclusion using Process Algebra with Signals 18
Victor Dyseryn, Rob van Glabbeek and Peter Höfner

Bisimulation and Hennessy-Milner Logic for Generalized Synchronization Trees 35
James Ferlez, Rance Cleaveland and Steve Marcus

Reversing Imperative Parallel Programs 51
James Hoey, Irek Ulidowski and Shoji Yuen

Using Session Types for Reasoning About Boundedness in the Pi-Calculus 67
Hans Hüttel

Distributive Laws for Monotone Specifications 83
Jurriaan Rot

Preface

This volume contains the proceedings of the Combined 24th International Workshop on Expressiveness in Concurrency and the 14th Workshop on Structural Operational Semantics (EXPRESS/SOS 2017) which was held on 4 September 2017 in Berlin, Germany, as an affiliated workshop of CONCUR 2017, the 28th International Conference on Concurrency Theory.

The EXPRESS workshops aim at bringing together researchers interested in the expressiveness of various formal systems and semantic notions, particularly in the field of concurrency. Their focus has traditionally been on the comparison between programming concepts (such as concurrent, functional, imperative, logic and object-oriented programming) and between mathematical models of computation (such as process algebras, Petri nets, event structures, modal logics, and rewrite systems) on the basis of their relative expressive power. The EXPRESS workshop series has run successfully since 1994 and over the years this focus has become broadly construed.

The SOS workshops aim at being a forum for researchers, students and practitioners interested in new developments, and directions for future investigation, in the field of structural operational semantics. One of the specific goals of the SOS workshop series is to establish synergies between the concurrency and programming language communities working on the theory and practice of SOS. Reports on applications of SOS to other fields are also most welcome, including: modelling and analysis of biological systems, security of computer systems programming, modelling and analysis of embedded systems, specification of middle-ware and coordination languages, programming language semantics and implementation, static analysis software and hardware verification, semantics for domain-specific languages and model-based engineering.

Since 2012, the EXPRESS and SOS communities have organized an annual combined EXPRESS/SOS workshop on the expressiveness of mathematical models of computation and the formal semantics of systems and programming concepts.

We received ten full paper submissions out of which the programme committee selected six full paper submissions for publication and presentation at the workshop. These proceedings contain the selected contributions. The workshop had two invited presentations:

SOS with Data: Bisimulation, Rule Formats, and Axiomatization,
by Mohammad R. Mousavi (University of Leicester, UK and Halmstad University, Sweden)
and
Rule Formats in Structural Operational Semantics,
by Rob van Glabbeek (CSIRO, Sydney, Australia)

Moreover, we had two presentations of short papers:

Delayed-choice semantics for pomset families and message sequence graphs,
by Clemens Dubslaff and Christel Baier
and
Branching Cell Decomposition, Confusion Freeness and Probabilistic Nets,
by Roberto Bruni, Hernan Melgratti, and Ugo Montanari

We would like to thank the authors of the submitted papers, the invited speakers, the members of the programme committee, and their subreviewers for their contribution to both the meeting and this volume. We also thank the CONCUR 2017 organizing committee for hosting EXPRESS/SOS 2017. Finally, we would like to thank our EPTCS editor Rob van Glabbeek for publishing these proceedings and his help during the preparation.

Kirstin Peters and Simone Tini,
August 2017.

Program Committee

Additional Reviewers

Sequential Composition in the Presence of Intermediate Termination (Extended Abstract)

Jos Baeten
CWI
Amsterdam, the Netherlands
University of Amsterdam,
Amsterdam, the Netherlands
Jos.Baeten@cwi.nl

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

Fei Yang
Eindhoven University of Technology
Eindhoven, the Netherlands
f.yang@tue.nl

The standard operational semantics of the sequential composition operator gives rise to unbounded branching and forgetfulness when transparent process expressions are put in sequence. Due to transparency, the correspondence between context-free and pushdown processes fails modulo bisimilarity, and it is not clear how to specify an always terminating half counter. We propose a revised operational semantics for the sequential composition operator in the context of intermediate termination. With the revised operational semantics, we eliminate transparency, allowing us to establish a close correspondence between context-free processes and pushdown processes. Moreover, we prove the reactive Turing powerfulness of TCP with iteration and nesting with the revised operational semantics for sequential composition.

1 Introduction

Sequential composition is a standard operator in many process calculi. The functionality of the sequential composition operator is to concatenate the behaviours of two systems. It has been widely used in many process calculi with the notation "$\cdot$". We illustrate its operational semantics by a process $P \cdot Q$ in TCP [2]. If the process P has a transition $P \xrightarrow{a} P'$ for some action label a, then the composition $P \cdot Q$ has the transition $P \cdot Q \xrightarrow{a} P' \cdot Q$. Termination is an important behaviour for models of computation [2]. A semantic distinction between successful and unsuccessful termination in concurrency theory (CT) is especially important for a smooth incorporation of the classical theory of automata and formal languages (AFT): the distinction is used to express whether a state in an automaton is accepting or not. Automata may even have states that are accepting and may still perform transitions; this phenomenon we call *intermediate termination*. From a concurrency-theoretic point of view, such behaviour is perhaps somewhat unnatural. To be able to express it nevertheless, we let an alternative composition inherit the option to terminate from just one of its components. The expression $a.(b+1)$ then denotes the process that does an a-transition and subsequently enters a state that is successfully terminated but can also do a b-transition.

To specify the operational semantics of sequential composition in a setting with a explicit successful termination, usually the following three rules are added: the first one states that the sequential composition $P \cdot Q$ terminates if both P and Q terminate; the second one states that if P admits a transition $P \xrightarrow{a} P'$, then $P \cdot Q$ admits a transition $P \cdot Q \xrightarrow{a} P' \cdot Q$; and the third one states that if P terminates, and there is a transition $Q \xrightarrow{a} Q'$, then we have the transition $P \cdot Q \xrightarrow{a} Q'$.

In this paper, we discuss a complication stemming from these operational semantics of the sequential composition operator. The complication is that a process expression P with the option to terminate is *transparent* in a sequential context $P \cdot Q$: if P may still perform observable behaviour other than termination, then this may be skipped by doing a transition from Q. There are two disadvantages of

K. Peters and S. Tini (Eds.): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics (EXPRESS/SOS 2017).
EPTCS 255, 2017, pp. 1–17, doi:10.4204/EPTCS.255.1

transparency in our attempts to achieve a smooth integration of process theory and the classical theory of automata and formal languages [7]:

The relationship between context-free processes (i.e., processes that can be specified with a guarded recursive specification over a language with action constants, constants for deadlock (**0**) and successful termination (**1**), and binary operations for sequential and alternative composition) and pushdown automata has been extensively discussed in the literature [4]. It has been shown that every context-free process is equivalent to the behaviour of some pushdown automaton (i.e., a pushdown process) modulo contra simulation, but not modulo rooted branching bisimulation. By stacking unboundedly many transparent terms with sequential composition, we would get an unboundedly branching transition system. It was shown that unboundedly branching behaviour cannot be specified by any pushdown process modulo rooted branching bisimulation [4]. In order to improve the result to a finer notion of behaviour equivalence, we need to eliminate the problem of unbounded branching.

Transparency also complicates matters if one wants to specify some form of memory (e.g., a counter, a stack, or a tape) that always has the option to terminate, but at the same time does not lose data. If the standard process algebraic specifications of such memory processes are generalised to a setting with intermediate termination, then either they are not always terminating, or they are 'forgetful' and may non-deterministically lose data. This is a concern when one tries to specify the behaviour of a pushdown automaton or a Reactive Turing machine in a process calculus [5, 19, 20]. The process calculus TCP with iteration and nesting is Turing complete [11, 12]. Moreover, it follows from the result in [12] that it is reactively Turing powerful if intermediate termination is not considered. However, it is not clear to us how to reconstruct the proof of reactive Turing powerfulness if termination is considered. Due to the forgetfulness on the stacking of transparent process expressions, it is not clear to us how to define a counter that is always terminating, which is crucial for establishing the reactive Turing powerfulness.

In order to avoid the (in some cases) undesirable feature of unbounded branching and forgetfulness, we propose a revised operational semantics for the sequential composition operator. The modification consists of disallowing a transition from the second component of a sequential composition if the first component is able to perform a transition. Thus, we avoid the problems mentioned above with the revised operator. We shall prove that every context-free process is bisimilar to a pushdown process, and that TCP with iteration and nesting is reactively Turing powerful modulo divergence-preserving branching bisimilarity (without resorting to recursion) in the revised semantics.

The research presented in this article is part of an attempt to achieve a smoother integration of the classical theory of automata and formal languages (AFT) within concurrency theory (CT). The idea is to recognise that a finite automaton is just a special type of labelled transition system, that more complicate automata (pushdown automata, Turing machines) naturally generate transition systems, and that there is a natural correspondence between regular expressions and grammars on the one hand and certain process calculi on the other hand. In [7, 9, 10] we have studied the various notions of automata from AFT modulo branching bisimilarity. In [8] we have explored the correspondence between finite automata and regular expressions extended with parallel composition modulo strong bisimilarity. In [5] we have proposed reactive Turing machines as an extension of Turing machines with concurrency-style interaction.

The paper is structured as follows. We first introduce TCP with the standard version of sequential composition in Section 2. Next, we discuss the complications caused by transparency in Section 3. Then, in Section 4, we propose the revised operational semantics of the sequential composition operator, and show that rooted divergence-preserving branching bisimulation is a congruence. In Section 5, we revisit the relationship between context-free processes and pushdown automata, and show that every context-free process is bisimilar to a pushdown process in our revised semantics. In Section 6, we prove that TCP with iteration and nesting is reactively Turing powerful in the revised semantics. In Section 7, we draw

some conclusions and propose some future work. The full version of this extended abstract, including proofs of the results, is available as [6].

2 Preliminaries

We start with introducing the notion of labelled transition system, which is used as the standard mathematical representation of behaviour. We consider transition systems with a subset of states marked as terminating states. We let $\mathcal{A}$ be a set of *action symbols*, and we extend $\mathcal{A}$ with a special symbol $\tau \notin \mathcal{A}$, which intuitively denotes unobservable internal activity of the system. We shall abbreviate $\mathcal{A} \cup \{\tau\}$ by $\mathcal{A}_\tau$.

Definition 1. *An* $\mathcal{A}_\tau$-labelled transition system *is a tuple* $(\mathcal{S}, \longrightarrow, \uparrow, \downarrow)$*, where*

1. $\mathcal{S}$ *is a set of* states*,*
2. $\longrightarrow \subseteq \mathcal{S} \times \mathcal{A}_\tau \times \mathcal{S}$ *is an* $\mathcal{A}_\tau$*-labelled* transition relation*,*
3. $\uparrow \in \mathcal{S}$ *is the initial state, and*
4. $\downarrow \subseteq \mathcal{S}$ *is a set of terminating states.*

Next, we shall introduce the process calculus *Theory of Sequential Processes* (TSP) that allows us to describe transition systems.

Let $\mathcal{N}$ be a countably infinite set of names. The set of process expressions $\mathcal{P}$ is generated by the following grammar ($a \in \mathcal{A}_\tau$, $N \in \mathcal{N}$):

$$P := \mathbf{0} \mid \mathbf{1} \mid a.P \mid P \cdot P \mid P + P \mid N \ .$$

We briefly comment on the operators in this syntax. The constant $\mathbf{0}$ denotes *deadlock*, the unsuccessfully terminated process. The constant $\mathbf{1}$ denotes *termination*, the successfully terminated process. For each action $a \in \mathcal{A}_\tau$ there is a unary operator $a.$ denoting action prefix; the process denoted by $a.P$ can do an a-labelled transition to the process P. The binary operator $+$ denotes alternative composition or choice. The binary operator $\cdot$ represents the sequential composition of two processes.

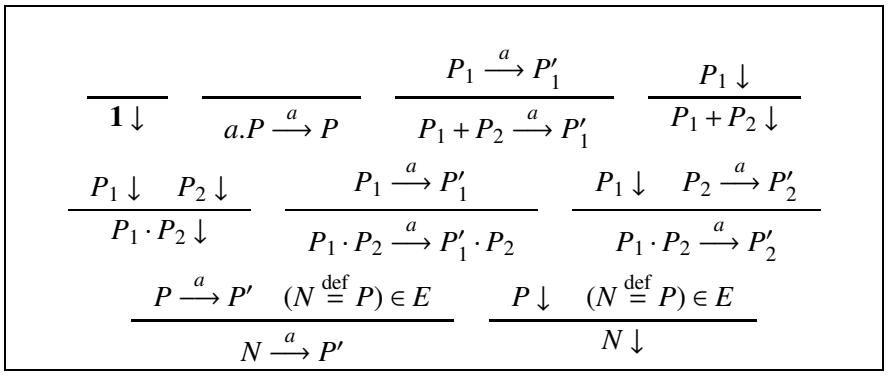
Let P be an arbitrary process expression; and we use an abbreviation inductively defined by: $P^0 = \mathbf{1}$; and $P^{n+1} = P \cdot P^n$ for all $n \in \mathbb{N}$.

A recursive specification E is a set of equations $E = \{N \stackrel{\text{def}}{=} P | N \in \mathcal{N}, P \in \mathcal{P}\}$, satisfying:

1. for every $N \in \mathcal{N}$ it includes at most one equation with N as left-hand side, which is referred to as the *defining equation* for N; and
2. if some name N' occurs in the right-hand side P' of some equation $N' = P'$ in E, then E must include a defining equation for N'.

An occurrence of a name N in a process expression is *guarded* if the occurrence is within the scope of an action prefix $a.$ for some $a \in A$ (τ cannot be a guard). A recursive specification E is guarded if all occurrences of names in right-hand sides of equations in E are guarded.

We use structural operational semantics to associate a transition relation with process expressions defined in TSP. A term is *closed* if it does not contain any free variables. Structural operational semantics induces a transition relation on closed terms. We let $\longrightarrow$ be the $\mathcal{A}_\tau$-labelled transition relation induced on the set of process expressions by operational rules in Figure 1. Note that we presuppose a recursive specification E, and we omit the symmetrical rules for $+$.



$$\frac{}{\mathbf{1}\downarrow} \qquad \frac{}{a.P \xrightarrow{a} P} \qquad \frac{P_1 \xrightarrow{a} P_1'}{P_1 + P_2 \xrightarrow{a} P_1'} \qquad \frac{P_1\downarrow}{P_1 + P_2\downarrow}$$

$$\frac{P_1\downarrow \quad P_2\downarrow}{P_1 \cdot P_2\downarrow} \qquad \frac{P_1 \xrightarrow{a} P_1'}{P_1 \cdot P_2 \xrightarrow{a} P_1' \cdot P_2} \qquad \frac{P_1\downarrow \quad P_2 \xrightarrow{a} P_2'}{P_1 \cdot P_2 \xrightarrow{a} P_2'}$$

$$\frac{P \xrightarrow{a} P' \quad (N \stackrel{\text{def}}{=} P) \in E}{N \xrightarrow{a} P'} \qquad \frac{P\downarrow \quad (N \stackrel{\text{def}}{=} P) \in E}{N\downarrow}$$

Figure 1: The operational semantics of TSP

Here we use $P \xrightarrow{a} P'$ to denote an a-labelled transition $(P, a, P') \in \longrightarrow$. We say a process expression P' is *reachable* from P is there exist process expressions $P_0, \ldots, P_n$ and labels $a_1, \ldots, a_n$ such that $P = P_0 \xrightarrow{a_1} \cdots \xrightarrow{a_n} P_n = P'$.

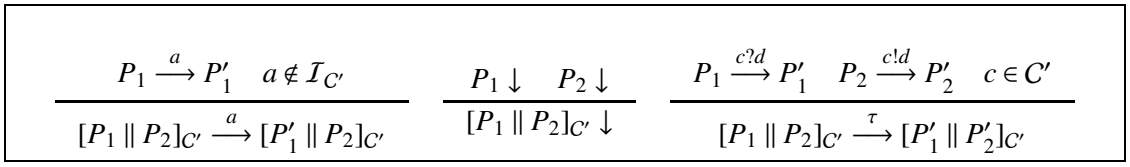
Given a TSP process expression P, the transition system $\mathcal{T}(P) = (\mathcal{S}_P, \longrightarrow_P, \uparrow_P, \downarrow_P)$ associated with P is defined as follows:

1. the set of states $\mathcal{S}_P$ consists of all process expressions reachable from P;
2. the transition relation $\longrightarrow_P$ is the restriction to $\mathcal{S}_P$ of the transition relation defined on all process expressions by the structural operational semantics, i.e., $\longrightarrow_P = \longrightarrow \cap (\mathcal{S}_P \times \mathcal{A}_\tau \times \mathcal{S}_P)$;
3. $\uparrow_P = P$; and
4. the set of final states $\downarrow_P$ consists of all process expressions $Q \in \mathcal{S}_P$ such that $Q\downarrow$, i.e., $\downarrow_P = \downarrow \cap \mathcal{S}_P$.

We also use (a restricted variant of) the process calculus TCP in later sections. It is obtained by adding a parallel composition operator to TSP. Let C be a set of *channels* and $\mathcal{D}_\Box$ be a set of *data symbols*. For every subset $C' \subseteq C$, we propose a special set of actions $\mathcal{I}_{C'} \subseteq \mathcal{A}_\tau$ defined by: $\mathcal{I}_{C'} = \{c?d, c!d \mid d \in \mathcal{D}_\Box, c \in C'\}$.

The actions $c?d$ and $c!d$ denote the events that a datum d is received or sent along channel c, respectively. We include binary parallel composition operators $[\_\|\_]_{C'}$ ($C \subseteq C$). Communication along the channels in C' is enforced and communication results in τ.

The operational semantics of the parallel composition operators is presented in Figure 2 (We omit the symmetrical rules).



$$\frac{P_1 \xrightarrow{a} P_1' \quad a \notin \mathcal{I}_{C'}}{[P_1 \parallel P_2]_{C'} \xrightarrow{a} [P_1' \parallel P_2]_{C'}} \qquad \frac{P_1\downarrow \quad P_2\downarrow}{[P_1 \parallel P_2]_{C'}\downarrow} \qquad \frac{P_1 \xrightarrow{c?d} P_1' \quad P_2 \xrightarrow{c!d} P_2' \quad c \in C'}{[P_1 \parallel P_2]_{C'} \xrightarrow{\tau} [P_1' \parallel P_2']_{C'}}$$

Figure 2: The operational semantics of parallel composition

The notion of behavioural equivalence has been used extensively in the theory of process calculi. We first introduce the notion of strong bisimilarity [21, 23], which does not distinguish τ-transitions from other labelled transitions.

Definition 2. *A binary symmetric relation $\mathcal{R}$ on a transition system $(\mathcal{S}, \longrightarrow, \uparrow, \downarrow)$ is a* strong bisimulation *if, for all states $s,t \in \mathcal{S}$, $s\mathcal{R}t$ implies*

1. *if $s \xrightarrow{a} s'$, then there exist $t' \in \mathcal{S}$, such that $t \xrightarrow{a} t'$, and $s'\mathcal{R}t'$;*
2. *if $s\downarrow$, then $t\downarrow$.*

The states s and t are strongly bisimilar *(notation: $s \leftrightarroweq t$) if there exists a strong bisimulation $\mathcal{R}$ s.t. $s\mathcal{R}t$.*

The notion of strong bisimilarity does not take into account the intuition associated with τ that it stands for unobservable internal activity. We proceed to introduce the notion of (divergence-preserving) branching bisimilarity, which does treat τ-transitions as unobservable. Divergence-preserving branching bisimilarity is the finest behavioural equivalence in van Glabbeek's linear time - branching time spectrum [16], and, moreover, the coarsest behavioural equivalence compatible with parallel composition that preserves validity of formulas from the well-known modal logic CTL minus the next-time modality X [18]. Let $\longrightarrow$ be an $\mathcal{A}_\tau$-labelled transition relation on a set $\mathcal{S}$, and let $a \in \mathcal{A}_\tau$; we write $s \xrightarrow{(a)} t$ for the formula "$s \xrightarrow{a} t \vee (a = \tau \wedge s = t)$". Furthermore, we denote the transitive closure of $\xrightarrow{\tau}$ by $\longrightarrow^+$ and the reflexive-transitive closure of $\xrightarrow{\tau}$ by $\longrightarrow^*$.

Definition 3. *Let $T = (\mathcal{S}, \longrightarrow, \uparrow, \downarrow)$ be a transition system. A branching bisimulation is a symmetric relation $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ such that for all states $s,t \in \mathcal{S}$, $s\mathcal{R}t$ implies*

1. *if $s \xrightarrow{a} s'$, then there exist $t', t'' \in \mathcal{S}$, such that $t \longrightarrow^* t'' \xrightarrow{(a)} t'$, $s\mathcal{R}t''$ and $s'\mathcal{R}t'$;*
2. *if $s\downarrow$, then there exists $t' \in \mathcal{S}$ such that $t \longrightarrow^* t'$, $t'\downarrow$ and $s\mathcal{R}t'$.*

The states s and t are branching bisimilar *(notation: $s \leftrightarroweq_b t$) if there exists a branching bisimulation $\mathcal{R}$ such that $s\mathcal{R}t$.*

A branching bisimulation $\mathcal{R}$ is divergence-preserving *if, for all states s and t, $s\mathcal{R}t$ implies*

3. *if there exists an infinite sequence $(s_i)_{i\in\mathbb{N}}$ such that $s = s_0$, $s_i \xrightarrow{\tau} s_{i+1}$ and $s_i\mathcal{R}t$ for all $i \in \mathbb{N}$, then there exists a state t' such that $t \longrightarrow^+ t'$ and $s_i\mathcal{R}t'$ for some $i \in \mathbb{N}$.*

The states s and t are divergence-preserving branching bisimilar *(notation: $s \leftrightarroweq_b^\Delta t$) if there exists a divergence-preserving branching bisimulation $\mathcal{R}$ such that $s\mathcal{R}t$.*

The relation $\leftrightarroweq_{\mathrm{b}}^{\Delta}$ satisfies the conditions of Definition 3, and is, in fact, the largest divergence-preserving branching bisimulation relation. Divergence-preserving branching bisimilarity is an equivalence relation [15].

Divergence-preserving branching bisimilarity is not a congruence for TSP; it is well-known that it is not compatible with alternative composition.. A rootedness condition needs to be introduced.

Definition 4. *Let $T = (\mathcal{S}, \longrightarrow, \uparrow, \downarrow)$ be a transition system. A divergence-preserving branching bismulation relation $\mathcal{R}$ on T satisfies the* rootedness *condition for a pair of states $s_1, s_2 \in \mathcal{S}$, if $s_1\mathcal{R}s_2$ and*

1. *if $s_1 \xrightarrow{a} s_1'$, then $s_2 \xrightarrow{a} s_2'$ for some s_2' such that $s_1'\mathcal{R}s_2'$;*
2. *if $s_1\downarrow$, then $s_2\downarrow$.*

s_1 and s_2 are rooted divergence-preserving branching bisimilar *(notation: $s_1 \leftrightarroweq_{rb}^{\Delta} s_2$) if there exists a divergence-preserving branching bisimulation $\mathcal{R}$ that satisfies rootedness condition for s_1 and s_2.*

We can extend the above relations ($\leftrightarroweq$, $\leftrightarroweq_{\mathrm{b}}$, $\leftrightarroweq_{\mathrm{b}}^{\Delta}$, and $\leftrightarroweq_{\mathrm{rb}}^{\Delta}$) to relations over two transition systems by defining that they are bisimilar if their initial states are bisimilar in their disjoint union. Namely, for two transition systems $T_1 = (\mathcal{S}_1, \longrightarrow_1, \uparrow_1, \downarrow_1)$ and $T_2 = (\mathcal{S}_2, \longrightarrow_2, \uparrow_2, \downarrow_2)$, we make the following pairing on their states. We pair every state $s \in \mathcal{S}_1$ with 1 and every state $s \in \mathcal{S}_2$ with 2. We have $T_i' = (\mathcal{S}_i', \longrightarrow_i', \uparrow_i', \downarrow_i')$ for $i = 1,2$ where $\mathcal{S}_i' = \{(s,i) \mid s \in \mathcal{S}_i\}$, $\longrightarrow_i' = \{((s,i),a,(t,i)) \mid (s,a,t) \in \longrightarrow_i\}$, $\uparrow_i' = (\uparrow_i, i)$, and $\downarrow_i' = \{(s,i) \mid s \in \downarrow_i\}$. We say $T_1 \equiv T_2$ if in $T = (\mathcal{S}_1' \cup \mathcal{S}_2', \longrightarrow_1' \cup \longrightarrow_2', \uparrow_1', \downarrow_1' \cup \downarrow_2')$ we have $\uparrow_1' \equiv \uparrow_2'$.

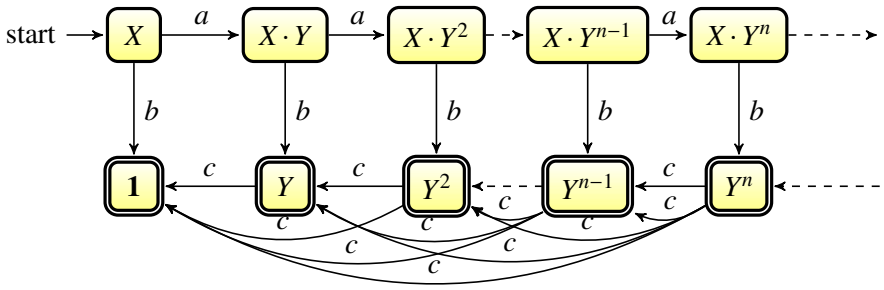


Figure 3: A transition system with unboundedly branching behaviour

3 Transparency

Process expressions that have the option to terminate are *transparent* in a sequential context: if P has the option to terminate and $Q \xrightarrow{a} Q'$, then $P \cdot Q \xrightarrow{a} Q'$ even if P can still do transitions. In this section we shall explain how transparency gives rise to two phenomena that are undesirable in certain circumstances. First, it facilitates the specification of unboundedly branching behaviour with a guarded recursive specification over TSP. Second, it gives rise to forgetful stacking of variables, and as a consequence it is not clear how to specify an always terminating half-counter.

We first discuss process expressions with unbounded branching. It is well-known from formal language theory that the context-free languages are exactly the languages accepted by pushdown automata. The process-theoretic formulation of this result is that every transition system specified by a TSP specification is language equivalent to the transition system associated with a pushdown automaton and, vice versa, every transition system associated with a pushdown automaton is language equivalent to the transition system associated with some TSP specification. The correspondence fails, however, when language equivalence is replaced by (strong) bisimilarity. The currently tightest result is that for every context-free process there is a pushdown process to simulate it modulo contra simulation [4]; we conjecture that not every context-free process is simulated by a pushdown process modulo branching bisimilarity. The reason is that context-free processes may have an unbounded branching degree. Consider the following process:

$$X = a.X \cdot Y + b.\mathbf{1} \quad Y = c.\mathbf{1} + \mathbf{1} \ .$$

The transition system associated with X is illustrated in Figure 3. Note that every state in the second row is a terminating state. The state Y^n has n c-labelled transitions to $\mathbf{1}, Y, Y^2, \ldots, Y^{n-1}$, respectively. Therefore, every state in this transition system has finitely many transitions leading to distinct states, but there is no upper bound on the number of transitions from each state. Therefore, we say that this transition system has an unbounded branching degree.

We can prove that the process defined by the TSP specification above is not strongly bisimilar to a pushdown process since it has an unbounded branching degree, whereas a pushdown process is always boundedly branching. The correspondence does hold modulo contra simulation [4], and it is an open problem as to whether the correspondence holds modulo branching bisimilarity. In Section 5, we show that with a revised operational semantics for sequential composition, we eliminate such unbounded branching and indeed obtain a correspondence between pushdown processes and context-free processes modulo strong bisimilarity.

Next, we discuss the phenomenon of forgetfulness. Bergstra, Bethke and Ponse introduce a process calculus with iteration and nesting [11, 12] in which a binary nesting operator $\sharp$ and a Kleene star operator * are added. In this paper, we add these two operators to TCP (Strictly speaking, we use an unary variant Kleene star operator). We give the operational semantics of these two operators in Figure 4.

$$\frac{}{P^*\downarrow} \qquad \frac{P \xrightarrow{a} P'}{P^* \xrightarrow{a} P' \cdot P^*}$$

$$\frac{P_1 \xrightarrow{a} P_1'}{P_1 \sharp P_2 \xrightarrow{a} P_1' \cdot (P_1 \sharp P_2) \cdot P_1} \qquad \frac{P_2 \xrightarrow{a} P_2'}{P_1 \sharp P_2 \xrightarrow{a} P_2'} \qquad \frac{P_2\downarrow}{P_1 \sharp P_2\downarrow} \ .$$

Figure 4: The operational semantics of nesting and iteration

To get some intuition for the operational interpretation of these operators, note that the processes P^* and $P_1 \sharp P_2$ respectively satisfy the following equations modulo strong bisimilarity:

$$P^* = P \cdot P^* + \mathbf{1} \quad P_1 \sharp P_2 = P_1 \cdot (P_1 \sharp P_2) \cdot P_1 + P_2$$

Bergstra et al. show how one can specify a half counter using iteration and nesting, which then allows them to conclude that the behaviour of a Turing machine can be simulated in the calculus with iteration and nesting (not including recursion) [11, 12].

The half counter is specified as follows:

$$\begin{aligned} CC_n &= a.CC_{n+1} + b.BB_n (n \in \mathbb{N}) \\ BB_n &= a.BB_{n-1} (n \geq 1) \\ BB_0 &= c.CC_0 \ . \end{aligned}$$

The behaviour of a half counter is illustrated in Figure 5. The initial state is CC_0. From CC_0 an arbitrary number of a transitions is possible. After a b-labelled transition, the process performs the same number of a-labelled transitions as before the b-labelled transition, to the state BB_0. In state BB_0, a zero testing transition, labelled by c is enabled, leading back to the state CC_0.

An implementation in a calculus with iteration and nesting is provided in [12] as follows:

$$HCC = ((a \sharp b) \cdot c)^* \ .$$

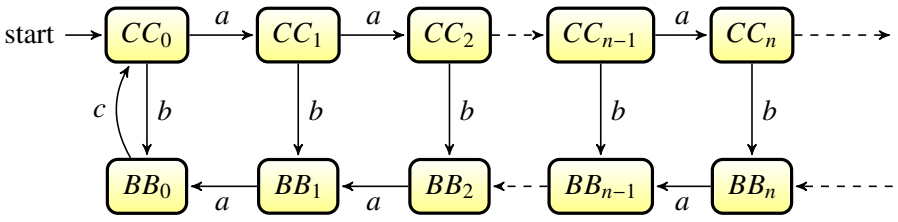


Figure 5: The transition system of a half counter

It is straightforward to establish that $((a^\sharp b)\cdot a^n\cdot c)\cdot HCC$ is equivalent to CC_n for all $n\geq 1$ modulo strong bisimilarity, and $(a^n\cdot c)\cdot HCC$ is equivalent to BB_n for all $n\in\mathbb{N}$ modulo strong bisimilarity.

In a context with intermediate termination, one may wonder if it is possible to generalize their result. It is, however, not clear how to specify an always terminating half counter. At least, a naive generalisation of the specification of Bergstra et al. does not do the job. The culprit is forgetfulness. We define a half counter that terminates in every state as follows:

$$\begin{aligned} C_n &= a.C_{n+1}+b.B_n+\mathbf{1} \quad (n\in\mathbb{N})\\ B_n &= a.B_{n-1}+\mathbf{1} \quad (n\geq 1)\\ B_0 &= c.C_0+\mathbf{1}\ . \end{aligned}$$

Now consider the process HC defined by:

$$HC=((a+\mathbf{1})^\sharp(b+\mathbf{1})\cdot(c+\mathbf{1}))^*\ .$$

Note that due to transparency, $((a+\mathbf{1})^n\cdot(c+\mathbf{1}))\cdot HC$ is not equivalent to B_n modulo any reasonable behavioural equivalence for $n>1$ since B_n only has an a-labelled transition to B_{n-1} whereas the other process has at least $n+1$ transitions leading to $HC, (c+\mathbf{1})\cdot HC, (a+\mathbf{1})\cdot(c+\mathbf{1})\cdot HC, \ldots, (a+\mathbf{1})^{n-1}\cdot(c+\mathbf{1})\cdot HC$, respectively. This process may choose to "forget" the transparent process expressions that have been stacked using the sequential composition operator. We conjecture that, due to forgetfulness, the always terminating half counter cannot be specified in $\mathrm{TCP}^\sharp$.

In Section 6, we show that with the revised semantics, it is possible to specify an always terminating half counter and we shall prove that TCP extended with $*$ and $\sharp$ (but without recursion) is reactively Turing powerful.

4 A Revised Semantics of the Sequential Composition Operator

Inspired by the work in [1] and [13], we revise the operational semantics for sequential composition and propose a calculus $\mathrm{TCP}^;$. Its syntax is obtained by replacing the sequential composition operator $\cdot$ by ; in the syntax of TCP. Note that we also use the abbreviation of P^n as we did for the standard version of the sequential composition operator.

The operational rules for ; are givem in Figure 6. Note that the third rule has a negative premise

$$\frac{P_1\downarrow \quad P_2\downarrow}{P_1;P_2\downarrow} \qquad \frac{P_1\xrightarrow{a}P_1'}{P_1;P_2\xrightarrow{a}P_1';P_2} \qquad \frac{P_1\downarrow \quad P_2\xrightarrow{a}P_2' \quad P_1\not\longrightarrow}{P_1;P_2\xrightarrow{a}P_2'}\ .$$

Figure 6: The revised semantics of sequential composition

$P_1\not\longrightarrow$. Intuitively, this rule is only applicable if there does not exist a closed term P_1' and an action $a\in\mathcal{A}_\tau$ such that the transition $P_1\xrightarrow{a}P_1'$ is derivable. For a sound formalisation of this intuition, using the notions of irredundant and well-supported proof, see [17]. As a consequence, the branching degree of a context-free process is bounded and sequential compositions may have the option to terminate, without being forgetful.

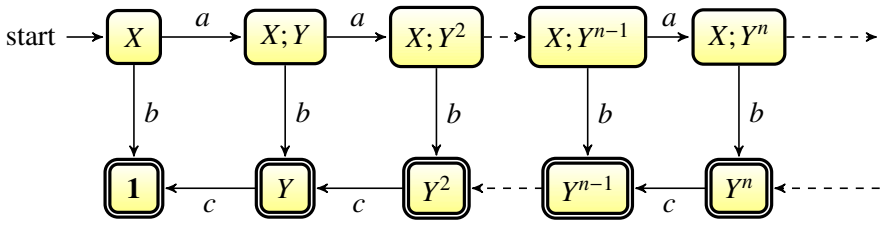


Figure 7: The transition system in the revised semantics

Let us revisit the first example in Section 3. We rewrite it with the revised sequential composition operator:

$$X = a.X;Y + b.\mathbf{1} \quad Y = c.\mathbf{1} + \mathbf{1} \ .$$

Its transition system is illustrated in Figure 7. Every state in the transition system now has a bounded branching degree. For instance, a transition from Y^5 to Y^2 is abandoned because Y has a transition and only the transition from the first Y in the sequential composition is allowed.

Congruence is an important property to fit a behavioural equivalence into an axiomatic framework. We have that in the revised semantics, $\leftrightarrows^{\Delta}_{\mathrm{rb}}$ is a congruence. Note that the congruence property can also be inferred from a recent result of Fokkink, van Glabbeek and Luttik [14].

Theorem 1. *$\leftrightarrows^{\Delta}_{rb}$ is a congruence with respect to TCP$^{;}$.*

As a remark, unlike the divergence-preserving variant of rooted branching bisimilarity, the more standard variant that does not require divergence-preservation ($\leftrightarrows_{\mathrm{rb}}$) is not a congruence for TCP$^{;}$. Consider

$$P_1 = \tau.\mathbf{1} \quad P_2 = (\tau.\mathbf{1})^* \quad Q = a.\mathbf{1} \ .$$

We have $P_1 \leftrightarrows_{\mathrm{rb}} P_2$ but not $P_1;Q \leftrightarrows_{\mathrm{rb}} P_2;Q$, for $P_1;Q$ can do a a-transition after the τ-transition, whereas $P_2;Q$ can only do τ transitions.

We also define a version of TCP with iteration and nesting (TCP$^{\sharp}$) in the revised semantics. By removing the facility of recursive specification and the operations $*$ and $\sharp$, we get TCP$^{\sharp}$. The operational rules for $*$ and $\sharp$ are obtained by replacing, in the rules in Figure 4, all occurrences of $\cdot$ by ;.

5 Context-free Processes and Pushdown Processes

The relationship between context-free processes and pushdown processes has been studied in the literature [4]. We consider the process calculus *Theory of Sequential Processes* (TSP$^{;}$). We define context-free processes as follows:

Definition 5. *A* context-free process *is the strong bisimulation equivalence class of the transition system generated by a finite guarded recursive specification over TSP$^{;}$.*

Note that there is a method to rewrite every context-free process into Greibach normal form [3], which is also valid in the revised semantics. In this paper, we only consider context-free processes in Greibach normal form ,i.e., defined by guarded recursive specifications of the form

$$X = \sum_{i \in I_X} \alpha_i.\xi_i(+\mathbf{1}) \ .$$

In this form, every right-hand side of every equation consists of a number of summands, indexed by a finite set I_X (the empty sum denotes $\mathbf{0}$), each of which is $\mathbf{1}$, or of the form $\alpha_i.\xi_i$, where ξ_i is the sequential composition of names (the empty sequence denotes $\mathbf{1}$).

We shall show that every context-free process is equivalent to a pushdown process modulo strong bisimilarity. The notion of pushdown automaton is defined as follows:

Definition 6. *A pushdown automaton (PDA) is a* 7*-tuple* $(\mathcal{S},\Sigma,\mathcal{D},\longrightarrow,\uparrow,Z,\downarrow)$*, where*

1. $\mathcal{S}$ *is a finite set of* states*,*
2. Σ *is a finite set of* input symbols*,*
3. $\mathcal{D}$ *is a finite set of* stack symbols*,*
4. $\longrightarrow \subseteq \mathcal{S}\times\mathcal{D}\times\Sigma\times\mathcal{D}^*\times\mathcal{S}$ *is a finite* transition relation*, (we write* $s \stackrel{a[d/\delta]}{\longrightarrow} t$ *for* $(s,d,a,\delta,t)\in\longrightarrow$*),*
5. $\uparrow \in \mathcal{S}$ *is the* initial state*,*
6. $Z \in \mathcal{D}$ *is the* initial stack symbol*, and*
7. $\downarrow \subseteq \mathcal{S}$ *is a set of* accepting states*.*

We use a sequence of stack symbols $\delta \in \mathcal{D}^*$ to represent the contents of a stack. We associate with every pushdown automaton a labelled transition system. The bisimulation equivalence classes of transition systems associated with pushdown automata are referred to as *pushdown processes*.

Definition 7. *Let* $\mathcal{M}=(\mathcal{S},\Sigma,\mathcal{D},\longrightarrow,\uparrow,Z,\downarrow)$ *be a PDA. The* transition system $\mathcal{T}(\mathcal{M})=(\mathcal{S}_\mathcal{T},\longrightarrow_\mathcal{T},\uparrow_\mathcal{T},\downarrow_\mathcal{T})$ *associated with* $\mathcal{M}$ *is defined as follows:*

1. *its set of states is the set* $\mathcal{S}_\mathcal{T}=\{(s,\delta)\mid s\in\mathcal{S},\delta\in\mathcal{D}^*\}$ *of all configurations of* $\mathcal{M}$*,*
2. *its transition relation* $\longrightarrow_\mathcal{T}\subseteq\mathcal{S}_\mathcal{T}\times\mathcal{A}_\tau\times\mathcal{S}_\mathcal{T}$ *is the relation satisfying, for all* $a\in\Sigma$*,* $d\in\mathcal{D}$*,* $\delta,\delta'\in\mathcal{D}^*$*:* $(s,d\delta)\stackrel{a}{\longrightarrow}_\mathcal{T}(t,\delta'\delta)$ *iff* $s\stackrel{a[d/\delta']}{\longrightarrow}t$*,*
3. *its initial state is the configuration* $\uparrow_\mathcal{T}=(\uparrow,Z)$*, and*
4. *its set of terminating states is the set* $\downarrow_\mathcal{T}=\{(s,\delta)\mid s\in\mathcal{S},\, s\downarrow,\,\delta\in\mathcal{D}^*\}$*.*

Recall that a context-free process is defined by a recursive specification in Greibach normal form; all states of the context-free process are denoted by sequences of names defined in this recursive specification. Note that a sequence of names denotes a terminating state only if all names have the option to terminate. Hence, to be able to determine whether a configuration of the pushdown automaton should have the option to terminate, we need to know whether all names currently on the stack have the option to terminate. We annotate the states of the pushdown automaton with the subset of names currently on the stack. We shall use the stack to record the sequence of names corresponding to the current state. The deepest occurrence of a name on the stack is marked and we shall include special transitions in the automaton for the treatment of marked names. If a marked name is removed from the stack, then , intuitively, it should be removed from the set annotating the state from the set. On the other hand, if a name not in the set is added to the stack, then we shall mark that name and add that name to the set annotating the state. As an example, we introduce a PDA as in Figure 8 to simulate the process in Figure 7 modulo $\underline{\leftrightarrow}$.

To obtain a general result, we consider a context-free process defined by a set of names $\mathcal{V}=\{X_0,X_1,\ldots,X_m\}$ with X_0 as the initial state, where

$$X_j=\sum_{i\in I_{X_j}}\alpha_{ij}.\xi_{ij}(+\mathbf{1})\ .$$

We introduce the following auxiliary functions:

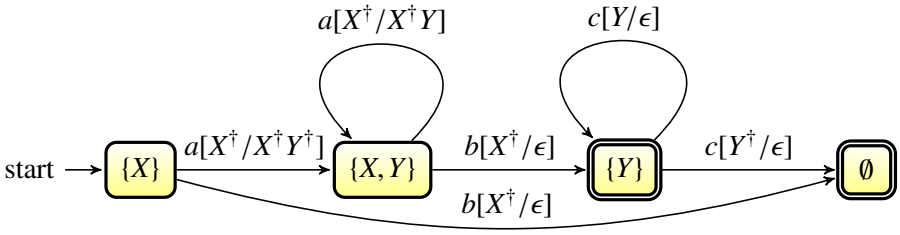


Figure 8: A PDA to simulate the process in Figure 7

1. $length : \mathcal{V}^* \to \mathbb{N}$, $length(\xi)$ is the length of ξ;
2. $get : \mathcal{V}^* \times \mathbb{N} \to \mathcal{V}$, $get(\xi, i)$ is the i-th name of ξ;
3. $suffset : \mathcal{V}^* \times \mathbb{N} \to 2^{|\mathcal{V}|}$, $suffset(\xi, i) = \{get(\xi, j) \mid j = i+1, \ldots length(\xi)\}$ computes the set that contains all the names in the suffix which starts from the i-th name of ξ.

We define a PDA $\mathcal{M} = (\mathcal{S}, \Sigma, \mathcal{D}, \longrightarrow, \uparrow, Z, \downarrow)$ to simulate the transition system associated with X_0 as follows: $\mathcal{S} = \{D \mid D \subseteq \mathcal{V}\}$; $\Sigma = \mathcal{A}_\tau$; $\mathcal{D} = \mathcal{V} \cup \{X^\dagger \mid X \in \mathcal{V}\}$; $\uparrow = \{X_0\}$; $Z = X_0^\dagger$; $\downarrow = \{D \mid \text{if for all } X \in D, X \downarrow\}$; and the transition relation $\longrightarrow$ is defined as follows:

$$\begin{aligned} \longrightarrow \quad &= \quad \{(D, X_j^\dagger, \alpha_{ij}, \delta(D, X_j^\dagger, \xi_{ij}), merge(D, X_j^\dagger, \xi_{ij})) \mid i \in I_{X_j},\ j = 1, \ldots, n,\ D \subseteq \mathcal{V}\} \\ &\cup \quad \{(D, X_j, \alpha_{ij}, \delta(D, X_j, \xi_{ij}), merge(D, X_j, \xi_{ij})) \mid i \in I_{X_j},\ j = 1, \ldots, n,\ D \subseteq \mathcal{V}\} \ , \end{aligned}$$

where $\delta(D, X_j^\dagger, \xi_{ij})$ is the string of length $length(\xi_{ij})$ defined as follows: for $k = 1, \ldots, length(\xi_{ij})$, we let $X_k = get(\xi_{ij}, k)$,

1. if $X_k \notin (D/\{X_j\}) \cup suffset(\xi_{ij}, k)$, then the k-th symbol of $\delta(D, X_j^\dagger, \xi_{ij})$ is $X_k^\dagger$,
2. otherwise, the k-th symbol of $\delta(D, X_j^\dagger, \xi_{ij})$ is X_k,

$\delta(D, X_j, \xi_{ij})$ is a string of length $length(\xi_{ij})$ defined as follows: for $k = 1, \ldots, length(\xi_{ij})$, we let $X_k = get(\xi_{ij}, k)$,

1. if $X_k \notin D \cup suffset(\xi_{ij}, k)$, then the k-th symbol of $\delta(D, X_j, \xi_{ij})$ is $X_k^\dagger$,
2. otherwise, the k-th symbol of $\delta(D, X_j, \xi_{ij})$ is X_k, and

we also define $merge(D, X_j^\dagger, \xi_{ij}) = (D/\{X_j\}) \cup suffset(\xi_{ij}, 0)$ and $merge(D, X_j, \xi_{ij}) = D \cup suffset(\xi_{ij}, 0)$;

We have the following result:

Lemma 1. $\mathcal{T}(X_0) \leftrightarrows \mathcal{T}(\mathcal{M})$.

We have the following theorem.

Theorem 2. *For every name X defined in a guarded recursive specification in Greibach normal form there exists a PDA $\mathcal{M}$, such that $\mathcal{T}(X) \leftrightarrows \mathcal{T}(\mathcal{M})$.*

Note that the converse of this theorem does not hold in general, a counterexample was established by F. Moller in [22], and we conjecture that it is also valid modulo $\leftrightarrows_b$ in the revised semantics.

6 Executability in the Context of Termination

The notion of reactive Turing machine (RTM) [5] was introduced as an extension of Turing machines to define which behaviour is executable by a computing system. The definition of RTM is parameterised with the set $\mathcal{A}_\tau$, which we now assume to be finite, and with another finite set $\mathcal{D}$ of *data symbols*. We extend $\mathcal{D}$ with a special symbol $\Box \notin \mathcal{D}$ to denote a blank tape cell, and denote the set $\mathcal{D} \cup \{\Box\}$ of *tape symbols* by $\mathcal{D}_\Box$.

Definition 8 (Reactive Turing Machine)**.** *A* reactive Turing machine *(RTM) is a quadruple* $(\mathcal{S}, \longrightarrow, \uparrow, \downarrow)$*, where*

1. $\mathcal{S}$ *is a finite set of* states*,*
2. $\longrightarrow \subseteq \mathcal{S} \times \mathcal{D}_\Box \times \mathcal{A}_\tau \times \mathcal{D}_\Box \times \{L,R\} \times \mathcal{S}$ *is a finite collection of* $(\mathcal{D}_\Box \times \mathcal{A}_\tau \times \mathcal{D}_\Box \times \{L,R\})$*-labelled* transitions *(we write* $s \xrightarrow{a[d/e]M} t$ *for* $(s,d,a,e,M,t) \in \longrightarrow$*),*
3. $\uparrow \in \mathcal{S}$ *is a distinguished* initial state*, and*
4. $\downarrow \subseteq \mathcal{S}$ *is a finite set of* final states*.*

Intuitively, the meaning of a transition $s \xrightarrow{a[d/e]M} t$ is that whenever the RTM is in state s, and d is the symbol currently read by the tape head, then it may execute the action a, write symbol e on the tape (replacing d), move the read/write head one position to the left or the right on the tape (depending on whether $M = L$ or $M = R$), and then end up in state t.

To formalise the intuitive understanding of the operational behaviour of RTMs, we associate with every RTM $\mathcal{M}$ an $\mathcal{A}_\tau$-labelled transition system $\mathcal{T}(\mathcal{M})$. The states of $\mathcal{T}(\mathcal{M})$ are the configurations of $\mathcal{M}$, which consist of a state from $\mathcal{S}$, its tape contents, and the position of the read/write head. We denote by $\check{\mathcal{D}}_\Box = \{\check{d} \mid d \in \mathcal{D}_\Box\}$ the set of *marked* symbols; a *tape instance* is a sequence $\delta \in (\mathcal{D}_\Box \cup \check{\mathcal{D}}_\Box)^*$ such that δ contains exactly one element of the set of marked symbols $\check{\mathcal{D}}_\Box$, indicating the position of the read/write head. We adopt a convention to concisely denote an update of the placement of the tape head marker. Let δ be an element of $\mathcal{D}_\Box^*$. Then by $\delta^<$ we denote the element of $(\mathcal{D}_\Box \cup \check{\mathcal{D}}_\Box)^*$ obtained by placing the tape head marker on the right-most symbol of δ (if that exists; otherwise $\delta^<$ denotes $\check{\Box}$). Similarly, ${}^>\delta$ is obtained by placing the tape head marker on the left-most symbol of δ (if that exists; otherwise ${}^>\delta$ denotes $\check{\Box}$).

Definition 9. *Let* $\mathcal{M} = (\mathcal{S}, \longrightarrow, \uparrow, \downarrow)$ *be an RTM. The* transition system $\mathcal{T}(\mathcal{M})$ associated with $\mathcal{M}$ *is defined as follows:*

1. *its set of states is the set* $C_\mathcal{M} = \{(s,\delta) \mid s \in \mathcal{S},\ \delta$ *a tape instance*$\}$ *of all configurations of* $\mathcal{M}$*;*
2. *its transition relation* $\longrightarrow \subseteq C_\mathcal{M} \times \mathcal{A}_\tau \times C_\mathcal{M}$ *is the relation satisfying, for all* $a \in \mathcal{A}_\tau$*,* $d,e \in \mathcal{D}_\Box$ *and* $\delta_L, \delta_R \in \mathcal{D}_\Box^*$*:* $(s, \delta_L \check{d} \delta_R) \xrightarrow{a} (t, \delta_L{}^< e \delta_R)$ *iff* $s \xrightarrow{a[d/e]L} t$*, and* $(s, \delta_L \check{d} \delta_R) \xrightarrow{a} (t, \delta_L e {}^> \delta_R)$ *iff* $s \xrightarrow{a[d/e]R} t$*;*
3. *its initial state is the configuration* $(\uparrow, \check{\Box})$*; and*
4. *its set of final states is the set* $\{(s,\delta) \mid \delta$ *a tape instance*$, s \downarrow\}$*.*

Turing introduced his machines to define the notion of *effectively computable function* in [24]. By analogy, the notion of RTM can be used to define a notion of *effectively executable behaviour*.

Definition 10 (Executability)**.** *A transition system is* executable *if it is the transition system associated with some RTM.*

Executability can be used to characterise the absolute expressiveness of process calculi in two ways. On the one hand, if every transition system associated with a process expression specified in a process calculus is executable modulo some behavioural equivalence, then we say that the process calculus is *executable* modulo that behavioural equivalence. On the other hand, if every executable transition system is behaviourally equivalent to some transition system associated with a process expression specified in a process calculus modulo some behavioural equivalence, then we say that the process calculus is *reactively Turing powerful* modulo that behavioural equivalence.

Our aim in this section is to prove that all executable processes can be specified, up to divergence-preserving branching bisimilarity in $\mathrm{TCP}^\sharp$. $\mathrm{TCP}^\sharp$ is obtained from TCP by removing recursive definitions and adding the iteration and nesting operators.

To see that $\mathrm{TCP}^\sharp$ is executable modulo branching bisimilarity, it suffices to observe that their transition systems are effective. Thus we can apply the result from [5] and conclude that they are executable modulo $\leftrightarrows_{\mathrm{b}}$.

Now we show that $\mathrm{TCP}^\sharp$ is reactively Turing powerful modulo $\leftrightarrows_{\mathrm{b}}^{\Delta}$.

We first introduce the notion of bisimulation up to $\leftrightarrows_{\mathrm{b}}$, which is a useful tool to establish the proofs in this section. Note that we adopt a non-symmetric bisimulation up to relation.

Definition 11. *Let $T = (\mathcal{S}, \longrightarrow, \uparrow, \downarrow)$ a transition system. A relation $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ is a bisimulation up to $\leftrightarrows_b$ if, whenever $s_1 \mathcal{R} s_2$, then for all $a \in \mathcal{A}_\tau$:*

1. *if $s_1 \longrightarrow^* s_1'' \xrightarrow{a} s_1'$, with $s_1 \leftrightarrows_b s_1''$ and $a \neq \tau \vee s_1'' \not\leftrightarrows_b s_1'$, then there exists s_2' such that $s_2 \xrightarrow{a} s_2'$, $s_1'' \leftrightarrows_b \circ \mathcal{R}\, s_2$ and $s_1' \leftrightarrows_b \circ \mathcal{R}\, s_2'$;*
2. *if $s_2 \xrightarrow{a} s_2'$, then there exist s_1', s_1'' such that $s_1 \longrightarrow^* s_1'' \xrightarrow{a} s_1'$, $s_1'' \leftrightarrows_b s_1$ and $s_1' \leftrightarrows_b \circ \mathcal{R}\, s_2'$;*
3. *if $s_1 \downarrow$, then there exists s_2' such that $s_2 \longrightarrow^* s_2'$, $s_2' \downarrow$ and $s_1 \mathcal{R} s_2'$; and*
4. *if $s_2 \downarrow$, then there exists s_1' such that $s_1 \longrightarrow^* s_1'$, $s_1' \downarrow$ and $s_1' \mathcal{R} s_2$.*

Lemma 2. *If $\mathcal{R}$ is a bisimulation up to $\leftrightarrows_b$, then $\mathcal{R} \subseteq \leftrightarrows_b$.*

Next we show that $\mathrm{TCP}^\sharp$ is reactively Turing powerful by writing a specification of the transition system associated with a reactive Turing machine in $\mathrm{TCP}^\sharp$ modulo $\leftrightarrows_{\mathrm{b}}^{\Delta}$. The proof proceeds in five steps:

1. We first specify an always terminating half counter.
2. Then we show that every regular process can be specified in $\mathrm{TCP}^\sharp$.
3. Next we use two half counters and a regular process to encode a terminating stack.
4. With two stacks and a regular process we can specify a tape.
5. Finally we use a tape and a regular control process to specify an RTM.

We first recall the infinite specification in $\mathrm{TSP}^;$ of a terminating half counter from Section 3. We provide a specification of a counter in $\mathrm{TCP}^\sharp$ as follows:

$$HC = ((a + \mathbf{1})^\sharp (b + \mathbf{1}); (c + \mathbf{1}))^*$$

We have the following lemma:

Lemma 3. $C_0 \leftrightarrows_b^{\Delta} HC$.

Next we show that every regular process can be specified in $\mathrm{TCP}^\sharp$ modulo $\leftrightarrows_{\mathrm{b}}^{\Delta}$. A regular process is given by $P_i = \sum_{j=1}^{n} \alpha_{ij} ; P_j + \beta_i (i = 1, \ldots, n)$ where α_{ij} and β_i are finite sums of actions from $\mathcal{A}_\tau$ and possibly with a $\mathbf{1}$-summand. We have the following lemma.

Lemma 4. *Every regular process can be specified in TCP$^\sharp$ modulo $\leftrightarrow_b^\Delta$.*

Now we show that a stack can be specified by a regular process and two half counters. We first give an infinite specification in TSP$^\cdot$ of a stack as follows:

$$\begin{aligned} S_\epsilon &= \Sigma_{d\in\mathcal{D}_\Box} push?d.S_d + pop!\Box.S_\epsilon + \mathbf{1} \\ S_{d\delta} &= pop!d.S_\delta + \Sigma_{e\in\mathcal{D}_\Box} push?e.S_{ed\delta} + \mathbf{1} \ . \end{aligned}$$

Note that $\mathcal{D}_\Box$ is a finite set of symbols. We suppose that $\mathcal{D}_\Box$ contains N symbols (including $\Box$). We use ϵ to denote the empty sequence. We inductively define an encoding from a sequence of symbols to a natural number $\lceil\_\rceil : \mathcal{D}_\Box{}^* \to \mathbb{N}$ as follows:

$$\lceil\epsilon\rceil = 0 \quad \lceil d_k\rceil = k \quad (k = 1,2,\ldots,N) \quad \lceil d_k\sigma\rceil = k + N \times \lceil\sigma\rceil \ .$$

Hence we are able to encode the contents of a stack in terms of natural numbers recorded by half counters. We define a stack in TCP$^\sharp$ as follows:

$$\begin{aligned} S &= [X_\emptyset \parallel P_1 \parallel P_2]_{\{a_1,a_2,b_1,b_2,c_1,c_2\}} \\ P_j &= ((a_j!a+\mathbf{1})^\sharp(b_j!b+\mathbf{1});(c_j!c+\mathbf{1}))^* \ (j=1,2) \\ X_\emptyset &= (\Sigma_{j=1}^N((push?d_j+\mathbf{1});(a_1?a+\mathbf{1})^j;(b_1+\mathbf{1});X_j)+pop!\Box)^* \\ X_k &= \Sigma_{j=1}^N((push?d_j+\mathbf{1});Push_j)+(pop!d_k+\mathbf{1});Pop_k \quad (k=1,2,\ldots,N) \\ Push_k &= Shift1to2;(a_1?a+\mathbf{1})^k;NShift2to1;X_k \quad (k=1,2,\ldots,N) \\ Pop_k &= (a_1?a+\mathbf{1})^k;1/NShift1to2;Test_\emptyset \\ Shift1to2 &= ((a_1?a+\mathbf{1});(a_2?a+\mathbf{1}))^*;(c_1?c+\mathbf{1});(b_2?b+\mathbf{1}) \\ NShift2to1 &= ((a_2?a+\mathbf{1});(a_1?a+\mathbf{1})^N)^*;(c_2?c+\mathbf{1});(b_1?b+\mathbf{1}) \\ 1/NShift1to2 &= ((a_1?a+\mathbf{1})^N;(a_2?a+\mathbf{1}))^*;(c_1?c+\mathbf{1});(b_2?b+\mathbf{1}) \\ Test_\emptyset &= (a_2?a+\mathbf{1});(a_1?a+\mathbf{1});Test_1+(c_2?c+\mathbf{1});X_\emptyset \\ Test_1 &= (a_2?a+\mathbf{1});(a_1?a+\mathbf{1});Test_2+(c_2?c+\mathbf{1});X_1 \\ Test_2 &= (a_2?a+\mathbf{1});(a_1?a+\mathbf{1});Test_3+(c_2?c+\mathbf{1});X_2 \\ &\cdots \\ Test_N &= (a_2?a+\mathbf{1});(a_1?a+\mathbf{1});Test_1+(c_2?c+\mathbf{1});X_N \ . \end{aligned}$$

We have the following result.

Lemma 5. $S_\epsilon \leftrightarrow_b^\Delta S$.

Next we proceed to define the tape by means of two stacks. We consider the following infinite specification in TSP$^\cdot$ of a tape:

$$T_{\delta_L\check{d}\delta_R} = r!d.T_{\delta_L\check{d}\delta_R} + \Sigma_{e\in\mathcal{D}_\Box} w?e.T_{\delta_L\check{e}\delta_R} + L?m.T_{\delta_L{}^<d\delta_R} + R?m.T_{\delta_L d^>\delta_R} + \mathbf{1} \ .$$

We define the tape process in TCP$^\sharp$ as follows:

$$\begin{aligned} T &= [T_\Box \parallel S_1 \parallel S_2]_{\{push_1,pop_1,push_2,pop_2\}} \\ T_d &= r!d.T_d + \Sigma_{e\in\mathcal{D}_\Box} w?e.T_e + L?m.Left_d + R?m.Right_d + \mathbf{1} \quad (d\in\mathcal{D}_\Box) \\ Left_d &= \Sigma_{e\in\mathcal{D}_\Box}((pop_1?e+\mathbf{1});(push_2!d+\mathbf{1});T_e) \\ Right_d &= \Sigma_{e\in\mathcal{D}_\Box}((pop_2?e+\mathbf{1});(push_1!d+\mathbf{1});T_e) \ , \end{aligned}$$

where S_1 and S_2 are two stacks obtained by renaming *push* and *pop* in S to $push_1, pop_1, push_2$ and pop_2, respectively. We establish the following result.

Lemma 6. $T_{\breve{\Box}} \leftrightarrows_b^\Delta T$.

Finally, we construct a finite control process for an RTM $\mathcal{M} = (\mathcal{S}_\mathcal{M}, \longrightarrow_\mathcal{M}, \uparrow_\mathcal{M}, \downarrow_\mathcal{M})$ as follows:

$$C_{s,d} = \Sigma_{(s,d,a,e,M,t)\in\longrightarrow_\mathcal{M}}(a.w!e.M!m.\Sigma_{f\in\mathcal{D}_\Box} r?f.C_{t,f})[+\mathbf{1}]_{s\downarrow_\mathcal{M}} (s \in \mathcal{S}_\mathcal{M}, d \in \mathcal{D}_\Box) \ .$$

We prove the following lemma.

Lemma 7. $\mathcal{T}(\mathcal{M}) \leftrightarrows_b^\Delta [C_{\uparrow_\mathcal{M},\Box} \parallel T]_{\{r,w,L,R\}}$.

We have the following theorem.

Theorem 3. *TCP$^\sharp$ is reactively Turing powerful modulo* $\leftrightarrows_b^\Delta$.

7 Conclusion

The results established in this paper show that a revision of the operational semantics of sequential composition leads to a smoother integration of process theory and the classical theory of automata and formal languages. In particular, the correspondence between context-free processes and pushdown processes can be established up to strong bisimilarity, which does not hold with the more standard operational semantics of sequential composition in a setting with intermediate termination [2]. Furthermore, the revised operational semantics of sequential composition also seems to work better in combination with the recursive operations of [12]. We conjecture that it is not possible to specify an always terminating counter or stack in a process calculus with iteration and nesting if the original operational semantics of sequential composition is used.

There are also some disadvantages to the revised operational semantics.

First of all, the negative premise in the operational semantics gives well-known formal complications in determining whether some process does, or does not, admit a transition. For instance, consider the following unguarded recursive specification:

$$X = X;Y + \mathbf{1} \quad Y = a.\mathbf{1} \ .$$

It is not a priori clear whether an a-transition is possible from X: if X *only* has the option to terminate, then $X;Y$ can do the a-transition from Y, but then also X can do the a-transition, contradicting the assumption that X *only* has the option to terminate.

Second, as we have illustrated in Section 4, rooted branching bisimilarity is not compatible with respect to the new sequential composition operation. The divergence-preserving condition is required for the congruence property.

Finally, note that $(a+\mathbf{1});b$ is not strongly bisimilar to $(a;b)+(\mathbf{1};b)$, and hence ; does not distribute from the right over +. It is to be expected that there is no finite sound and ground-complete set of equational axioms for the process calculus TCP$_;$ with respect to strong bisimilarity. We leave for future work to further investigate the equational theory of sequential composition.

Another interesting future work is to establish the reactive Turing powerfulness on other process calculi with non-regular iterators based on the revised semantics of the sequential composition operator. For instance, we could consider the pushdown operator "\$" and the back-and-forth operator "$\leftrightarrows$" introduced by Bergstra and Ponse in [12]. They are given by the following equations:

$$P_1{}^\$P_2 = P_1;(P_1{}^\$P_2);(P_1{}^\$P_2)+P_2 \quad P_1{}^\leftrightarrows P_2 = P_1;(P_1{}^\leftrightarrows P_2);P_2+P_2 \ .$$

By analogy to the nesting operator, we shall also give them some proper rules of operational semantics, and then use the calculus obtained by the revised semantics to define other versions of terminating counters. In a way, we should be able to establish their reactive Turing powerfulness.

References

[1] Luca Aceto & Matthew Hennessy (1992): *Termination, deadlock, and divergence*. *Journal of the ACM (JACM)* 39(1), pp. 147–187, doi:10.1145/147508.147527.

[2] Jos Baeten, Twan Basten & Michel Reniers (2010): *Process algebra: equational theories of communicating processes*. *Cambridge Tracts in Theoretical Computer Science* 50, Cambridge University Press, doi:10.1017/CBO9781139195003.

[3] Jos Baeten, Jan Bergstra & Jan Klop (1993): *Decidability of bisimulation equivalence for processes generating context-free languages*. *J. ACM* 40(3), pp. 653–682, doi:10.1145/174130.174141.

[4] Jos Baeten, Pieter Cuijpers & Paul van Tilburg (2008): *A context-free process as a pushdown automaton*. In: *International Conference on Concurrency Theory*, Springer, pp. 98–113, doi:10.1007/978-3-540-85361-9_11.

[5] Jos Baeten, Bas Luttik & Paul van Tilburg (2013): *Reactive Turing Machines*. *Inform. Comput.* 231, pp. 143–166, doi:10.1016/j.ic.2013.08.010.

[6] Jos Baeten, Bas Luttik & Fei Yang (2017): *Sequential composition in the presence of intermediate termination*. *CoRR* abs/1706.08401. Available at http://arxiv.org/abs/1706.08401.

[7] Jos C. M. Baeten, Pieter J. L. Cuijpers, Bas Luttik & P. J. A. van Tilburg (2009): *A process-theoretic look at automata*. In Farhad Arbab & Marjan Sirjani, editors: *Fundamentals of Software Engineering, Third IPM International Conference, FSEN 2009, Kish Island, Iran, April 15-17, 2009, Revised Selected Papers*, *Lecture Notes in Computer Science* 5961, Springer, pp. 1–33, doi:10.1007/978-3-642-11623-0_1.

[8] Jos C. M. Baeten, Bas Luttik, Tim Muller & Paul van Tilburg (2016): *Expressiveness modulo bisimilarity of regular expressions with parallel composition*. *Mathematical Structures in Computer Science* 26, pp. 933–968, doi:10.1017/S0960129514000309.

[9] Jos C. M. Baeten, Bas Luttik & Paul van Tilburg (2011): *Computations and interaction*. In Raja Natarajan & Adegboyega K. Ojo, editors: *ICDCIT*, *Lecture Notes in Computer Science* 6536, Springer, pp. 35–54, doi:10.1007/978-3-642-19056-8_3.

[10] Jos C. M. Baeten, Bas Luttik & Paul van Tilburg (2012): *Turing Meets Milner*. In Maciej Koutny & Irek Ulidowski, editors: *CONCUR*, *Lecture Notes in Computer Science* 7454, Springer, pp. 1–20, doi:10.1007/978-3-642-32940-1_1.

[11] Jan Bergstra, Inge Bethke & Alban Ponse (1994): *Process algebra with iteration and nesting*. *The Computer Journal* 37(4), pp. 243–258, doi:10.1093/comjnl/37.4.243.

[12] Jan Bergstra & Alban Ponse (2001): *Non-regular iterators in process algebra*. *Theoretical Computer Science* 269(1), pp. 203–229, doi:10.1016/S0304-3975(00)00413-8.

[13] Bard Bloom (1994): *When is partial trace equivalence adequate?* *Formal Aspects of Computing* 6(3), pp. 317–338, doi:10.1007/BF01215409.

[14] Wan Fokkink, Rob van Glabbeek & Bas Luttik (2017): *Divide and congruence III: stability & divergence*. In: *Proceedings 28th International Conference on Concurrency Theory (CONCUR 2017)*, *Leibniz International Proceedings in Informatics (LIPIcs)* 85, pp. 11:1–11:15.

[15] Rob van Glabbeek, Bas Luttik & Nikola Trčka (2009): *Branching bisimilarity with explicit divergence*. *Fundamenta Informaticae* 93(4), pp. 371–392, doi:10.3233/FI-2009-109.

[16] Rob J. van Glabbeek (1993): *The linear time - branching time spectrum II*. In Eike Best, editor: *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, *Lecture Notes in Computer Science* 715, Springer, pp. 66–81, doi:10.1007/3-540-57208-2_6.

[17] Rob J. van Glabbeek (2004): *The meaning of negative premises in transition system specifications II*. *J. Log. Algebr. Program.* 60-61, pp. 229–258, doi:10.1016/j.jlap.2004.03.007.

[18] Rob J. van Glabbeek, Bas Luttik & Nikola Trčka (2009): *Computation Tree Logic with deadlock detection*. *Logical Methods in Computer Science* 5(4), doi:10.2168/LMCS-5(4:5)2009.

[19] Bas Luttik & Fei Yang (2015): *Executable behaviour and the π-calculus (extended abstract)*. In Sophia Knight, Ivan Lanese, Alberto Lluch-Lafuente & Hugo Torres Vieira, editors: *Proceedings 8th Interaction and Concurrency Experience, ICE 2015, Grenoble, France, 4-5th June 2015.*, *EPTCS* 189, pp. 37–52, doi:10.4204/EPTCS.189.5.

[20] Bas Luttik & Fei Yang (2016): *On the executability of interactive computation*. In Arnold Beckmann, Laurent Bienvenu & Natasa Jonoska, editors: *Pursuit of the Universal - 12th Conference on Computability in Europe, CiE 2016, Paris, France, June 27 - July 1, 2016, Proceedings*, *Lecture Notes in Computer Science* 9709, Springer, pp. 312–322, doi:10.1007/978-3-319-40189-8_32.

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

[22] Faron Moller (1996): *Infinite results*. In Ugo Montanari & Vladimiro Sassone, editors: *CONCUR '96, Concurrency Theory, 7th International Conference, Pisa, Italy, August 26-29, 1996, Proceedings*, *Lecture Notes in Computer Science* 1119, Springer, pp. 195–216, doi:10.1007/3-540-61604-7_56.

[23] David Park (1981): *Concurrency and automata on infinite sequences*. In P. Deussen, editor: *Theoretical Computer Science*, *Lectures Notes in Computer Science* 104, Springer, pp. 167–183, doi:10.1007/BFb0017309.

[24] Alan Turing (1937): *On Computable Numbers, with an Application to the Entscheidungsproblem*. *Proceedings of the London Mathematical Society* s2-42(1), pp. 230–265, doi:10.1112/plms/s2-42.1.230.

Analysing Mutual Exclusion using Process Algebra with Signals

Victor Dyseryn
Ecole Polytechnique, Paris, France
Data61, CSIRO, Sydney, Australia
victor.dyseryn-fostier@polytechnique.edu

Rob van Glabbeek & Peter Höfner
Data61, CSIRO, Sydney, Australia
School of Computer Science and Engineering
University of New South Wales, Sydney, Australia
rvg@cs.stanford.edu Peter.Hoefner@data61.csiro.au

In contrast to common belief, the Calculus of Communicating Systems (CCS) and similar process algebras lack the expressive power to accurately capture mutual exclusion protocols without enriching the language with fairness assumptions. Adding a fairness assumption to implement a mutual exclusion protocol seems counter-intuitive. We employ a signalling operator, which can be combined with CCS, or other process calculi, and show that this minimal extension is expressive enough to model mutual exclusion: we confirm the correctness of Peterson's mutual exclusion algorithm for two processes, as well as Lamport's bakery algorithm, under reasonable assumptions on the underlying memory model. The correctness of Peterson's algorithm for more than two processes requires stronger, less realistic assumptions on the underlying memory model.

1 Introduction

In the process algebra community it is common belief that, on some level of abstraction, any distributed system can be modelled in standard process-algebraic specification formalisms like the Calculus of Communicating Systems (CCS) [26].

However, this sentiment has been proven incorrect [20]: two of the authors presented a simple fair scheduler—one that in suitable variations occurs in many distributed systems—of which *no* implementation can be expressed in CCS, unless CCS is enriched with a fairness assumption. Instances of our fair scheduler, that hence cannot be rendered correctly, are the *First in First out*[1], *Round Robin*, and *Fair Queueing* scheduling algorithms[2] as used in network routers [28, 29] and operating systems [23], or the *Completely Fair Scheduler*[3], which is the default scheduler of the Linux kernel since version 2.6.23. Since fair schedulers can be implemented in terms of mutual exclusion, this result implies that mutual exclusion protocols, such as the ones by Dekker [13, 15], Peterson [31] and Lamport [25], cannot be rendered correctly in CCS without imposing a fairness assumption.

Close approximations of Dekker's and Peterson's protocols rendered in CCS or similar formalisms abound in the literature [34, 5, 32, 16, 2]. Unless one makes a fairness assumption these renderings do not possess the liveness property that when a process leaves its non-critical section, and thus wants to enter the critical section, it will eventually succeed in doing so. When assuming fairness, this problem disappears [9]. However, since mutual exclusion protocols are often employed to ensure that each of several tasks gets allocated a fair amount of a shared resource, assuming fairness to implement mutual exclusion appears counter-intuitive.

Informally speaking, the reason why the CCS rendering of algorithms such as Peterson's does not work, is that it is possible that a process never gets a chance to write to a shared variable to indicate interest in entering the critical section. This is because other processes running in parallel and competing

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

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

[3] http://en.wikipedia.org/wiki/Completely_Fair_Scheduler

K. Peters and S. Tini (Eds.): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics (EXPRESS/SOS 2017).
EPTCS 255, 2017, pp. 18–34, doi:10.4204/EPTCS.255.2

$$\alpha.P \xrightarrow{\alpha} P \qquad \frac{P_j \xrightarrow{\alpha} P'}{\sum_{i\in I} P_i \xrightarrow{\alpha} P'}\ (j\in I)$$

$$\frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \qquad \frac{P \xrightarrow{a} P',\ Q \xrightarrow{\bar{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q'} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P|Q \xrightarrow{\alpha} P|Q'}$$

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

Table 1: Structural operational semantics of CCS

for the critical section are 'too busy' reading the shared variable all the time.

In this paper we extend CCS with *signals*. This extension is able to express mutual exclusion protocols *without* the use of fairness assumptions. To prove correctness, one only needs basic assumptions such as progress and justness.

We will use this extension to analyse the correctness of some of the most famous protocols for mutual exclusion, namely Peterson's algorithm, the filter lock algorithm—Peterson's algorithm for more than two processes—and Lamport's bakery algorithm. With regards to the filter lock algorithm our analysis reveals some surprising protocol behaviour.

2 Preliminaries: The Calculus of Communicating Systems

The Calculus of Communicating Systems (CCS) [26] is a process algebra, which is used to describe concurrent processes.

It is parametrised with sets $\mathscr{A}$ and $\mathscr{K}$ of *names* and *agent identifiers*. We define the set of *handshake actions* as $\mathscr{H} := \mathscr{A} \cup \bar{\mathscr{A}}$, where $\bar{\mathscr{A}} := \{\bar{a} \mid a \in \mathscr{A}\}$ is the set of *co-names*. Complementation is extended to $\mathscr{H}$ by setting $\bar{\bar{a}} = a$. Finally, $Act := \mathscr{H} \cup \{\tau\}$ is the set of *actions*, where τ is a special *internal action*. In this paper $a, b, c, \ldots$ range over $\mathscr{H}$, α, β over Act, and A, B range over $\mathscr{K}$. A *relabelling* is a function $f : \mathscr{H} \to \mathscr{H}$ satisfying $f(\bar{a}) = \overline{f(a)}$; it extends to Act by $f(\tau) := \tau$. Each $A \in \mathscr{K}$ comes with a defining equation $A \overset{def}{=} P$ with P being a CCS expression as defined below.

The class $\mathscr{T}_{\mathrm{CCS}}$ of *CCS expressions* is defined as the smallest class that includes

- *agent identifiers* $A \in \mathscr{K}$;
- *prefixes* $\alpha.P$;
- *(infinite) choices* $\sum_{i\in I} P_i$;
- *parallel compositions* $P|Q$;
- *restrictions* $P\backslash L$;
- *relabellings* $P[f]$;

where $P, P_i, Q \in \mathscr{T}_{\mathrm{CCS}}$ are CCS expressions, I an index set, $L \subseteq \mathscr{A}$ a set of names, and f an arbitrary relabelling function. In case $I = \{1,2\}$, we write $P_1 + P_2$ for $\sum_{i\in I} P_i$. The *inactive process* $\mathbf{0}$ is defined by $\sum_{i\in} P_i$; it is not capable to perform any action.

The semantics of CCS is given by the labelled transition relation $\to \subseteq \mathscr{T}_{\mathrm{CCS}} \times Act \times \mathscr{T}_{\mathrm{CCS}}$, where transitions $P \xrightarrow{\alpha} Q$ are derived from the rules of Table 1. The process $\alpha.P$ performs the action α first and subsequently acts as P. The choice operator $\sum_{i\in I} P_i$ may act as any of the P_i, depending on which of the processes is able to act at all. The parallel composition $P|Q$ executes an action from P, an action from Q, or in the case where P and Q can perform complementary actions a and $\bar{a}$, the process can perform a synchronisation, resulting in an internal action τ. The restriction operator $P\backslash L$ inhibits execution of the actions from L and their complements. The only way for a subprocess of $P\backslash L$ to perform an action $a \in L$ is through synchronisation with another subprocess of $P\backslash L$, which performs $\bar{a}$. The relabelling $P[f]$ acts

like process P with all labels a replaced by $f(a)$. Last, the rule for agent identifiers says that an agent A has the same transitions as the body P of its defining equation.

As usual, to avoid parentheses, we assume that the operators have decreasing binding strength in the following order: restriction and relabelling, prefixing, parallel composition, choice.

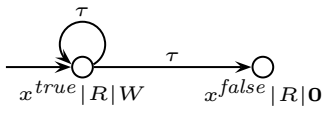
The pair $\langle \mathscr{T}_{\mathrm{CCS}}, \rightarrow \rangle$ is called the *labelled transition system* (LTS) of CCS.

Example 1 We describe a simple shared memory system in CCS, using the name $asgn_x^v$ for the assignment of value v to the variable x, and n_x^v for noticing or notifying that the variable x has the value v. The action $\overline{asgn_x^v}$ communicates the assignment $x := v$ to the shared memory, whereas $asgn_x^v$ is the action of the shared memory of accepting this communication. Likewise, $\overline{n_x^v}$ is a notification by the shared memory that x equals v; it synchronises with the complementary action n_x^v of noticing that $x = v$.

We consider the process $(x^{true} \,|\, R \,|\, W)\backslash\{asgn_x^{true}, asgn_x^{false}, n_x^{true}, n_x^{false}\}$, where

$$x^{true} \stackrel{def}{=} asgn_x^{true} \,.\, x^{true} + asgn_x^{false} \,.\, x^{false} + \overline{n_x^{true}} \,.\, x^{true} \,,$$
$$x^{false} \stackrel{def}{=} asgn_x^{true} \,.\, x^{true} + asgn_x^{false} \,.\, x^{false} + \overline{n_x^{false}} \,.\, x^{false} \,,$$
$$R \stackrel{def}{=} n_x^{true} \,.\, R \qquad \text{and} \qquad W \stackrel{def}{=} \overline{asgn_x^{false}} \,.\, \mathbf{0} \,.$$

The processes x^{true} and x^{false} model the two states of a shared Boolean variable x (*true* and *false*, respectively). Both accept assignment actions, changing their state accordingly. They also provide their respective value to a potential reader. The process R (*reader*) is an infinite loop which permanently tries to read value *true* from variable x, and the process W (*writer*) tries once to set variable x to false. Since the overall process uses the restriction operator, its transition system, depicted on the right, has only two transitions, a τ-loop of R reading the value, and a transition to $x^{false}|R|\mathbf{0}$ of W assigning x to false—after that no further transition is possible. The justness assumption, to be described in Sects. 4 and 5, is not sufficient to ensure that the writer eventually performs its transition, and this fact is one of the motivations why we introduce signals in Sect. 6.



3 Peterson's Mutual Exclusion Protocol—Part I

In [20] it is shown that Peterson's mutual exclusion protocol [31] cannot be expressed in CCS without assuming fairness. In this section we briefly recapitulate the protocol itself and present an optimal rendering in CCS. In the next section we discuss what the problems are with such a rendering.

The 'classical' Peterson's mutual exclusion protocol deals with two concurrent processes A and B that want to alternate critical and noncritical sections.

Each of the processes will stay only a finite amount of time in the critical section, although it is allowed to stay forever in its noncritical section. The purpose of the algorithm is to ensure that the processes are never simultaneously in the critical section, and to guarantee that both processes keep making progress; in particular the latter means that if a process wants to access the critical section it will eventually do so.

A pseudocode rendering of Peterson's protocol is depicted in Fig. 1. The processes use three shared variables: *readyA*, *readyB* and *turn*. The Boolean variable *readyA* can be written by Process A and read by Process B, whereas *readyB* can be written by B and read by A. By setting *readyA* to *true*, Process A signals to Process B that it wants to enter the critical section. The variable *turn* can be written and read by both processes. Its carefully designed functionality guarantees mutual exclusion as well as deadlock-freedom. Both *readyA* and *readyB* are initialised with *false* and *turn* with *A*.

Process A

repeat forever

ℓ_1 **noncritical section**
ℓ_2 $readyA := true$
ℓ_3 $turn := B$
ℓ_4 **await** $(readyB = false \vee turn = A)$
ℓ_5 **critical section**
ℓ_6 $readyA := false$

Process B

repeat forever

m_1 **noncritical section**
m_2 $readyB := true$
m_3 $turn := A$
m_4 **await** $(readyA = false \vee turn = B)$
m_5 **critical section**
m_6 $readyB := false$

Figure 1: Peterson's algorithm (pseudocode)

In order to model this protocol in CCS, we use the names **noncritA**, **critA**, **noncritB**, and **critB**, for Processes A and B executing their (non)critical section. The names $asgn^v_x$ and n^v_x for the interactions of A and B with a shared memory have been defined in Ex. 1. The Processes A and B can be modelled as

$$\mathrm{A} \stackrel{def}{=} \mathbf{noncritA} \,.\, \overline{asgn^{true}_{readyA}} \,.\, \overline{asgn^{\mathrm{B}}_{turn}} \,.\, (n^{false}_{readyB} + n^{\mathrm{A}}_{turn}) \,.\, \mathbf{critA} \,.\, \overline{asgn^{false}_{readyA}} \,.\, \mathrm{A} \;,$$
$$\mathrm{B} \stackrel{def}{=} \mathbf{noncritB} \,.\, \overline{asgn^{true}_{readyB}} \,.\, \overline{asgn^{\mathrm{A}}_{turn}} \,.\, (n^{false}_{readyA} + n^{\mathrm{B}}_{turn}) \,.\, \mathbf{critB} \,.\, \overline{asgn^{false}_{readyB}} \,.\, \mathrm{B} \;,$$

where $(a+b).P$ is a shorthand for $a.P + b.P$. This CCS rendering naturally captures the **await** statement, requiring Process A to wait at instruction ℓ_4 until it can read that $readyB = false$ or $turn = A$. We use two agent identifiers for each Boolean variable, one for each value, similar to Ex. 1. For example, we have $Turn^{\mathrm{A}} \stackrel{def}{=} asgn^{\mathrm{A}}_{turn} \,.\, Turn^{\mathrm{A}} + asgn^{\mathrm{B}}_{turn} \,.\, Turn^{\mathrm{B}} + \overline{n^{\mathrm{A}}_{turn}} \,.\, Turn^{\mathrm{A}}$. Peterson's algorithm is the parallel composition of all these processes, restricting all the communications

$$(\mathrm{A} \,|\, \mathrm{B} \,|\, ReadyA^{false} \,|\, ReadyB^{false} \,|\, Turn^{A}) \backslash L \;,$$

where L is the set of all names except **noncritA**, **critA**, **noncritB**, and **critB**.

It is well known that Peterson's protocol satisfies the safety property that both processes are never in the critical section at the same time. In terms of Fig. 1, there is no reachable state where A and B have already executed lines ℓ_4 and m_4 but have not yet executed ℓ_6 or m_6 [31, 34]. The validity of the liveness property, that any process leaving its noncritical section will eventually enter the critical section, depends on its precise formalisation, as discussed in Sects. 4 and 5.

4 Why the CCS Rendering of Peterson's Algorithm is Unsatisfactory

Liveness properties generally only hold under some assumptions. The intended liveness property for Peterson's algorithm may already be violated if both processes come to a permanent halt for no apparent reason. This behaviour should be considered unrealistic. To rule it out one usually makes a *progress* assumption, formalised in Sect. 5, which can be formulated as follows [17, 21]:

Any process in a state that admits a non-blocking action will eventually perform an action.

Another example is an execution path ρ in which first Process A completes instruction ℓ_1; leaving its noncritical section it implicitly wishes to enter the critical section. Subsequently, Process B cycles through its complete list of instructions in perpetuity without A making any further progress. This is possible because *readyA* is never updated and always evaluates to false. This execution path, if admitted, would be another counterexample to the intended liveness property. However, progress is not sufficient to rule out such a path; after all the whole system is making progress. To rule it out as a valid system run, we need the stronger assumption of *justness* [21], or an even stronger *fairness* assumption [18].

We formalise justness in the next section. Here we sketch the general idea, and the difference with fairness, by an example.

Suppose we have a vending machine with a single slot for inserting coins, and there are two customers: one intends to insert an infinite supply of quarters, and the other an infinite supply of dimes. No customer intends to ever extract something from the vending machine. Since a quarter and a dime cannot be entered simultaneously, the two customers compete for a shared resource. Should it be allowed for one customer to enter an unending sequence of quarters, while the other does not even get in a single dime? The assumption of (strong or weak) fairness rules out this realistic behaviour, while weaker assumptions like progress and justness allow this as one of the valid ways such interactions between the customers and the vending machine may play out.

Alternatively, assume that the same two customers have access to a vending machine each, and that each of these vending machines serves that customer only. In that case the assumption of progress is not strong enough to rule out that one customer enters an unending sequence of quarters, while the other does not even get in a single dime; after all the whole system keeps making progress at all times. The assumption of justness guarantees that the customer will get a chance to enter his dimes by applying the idea of progress to isolated components of a system; it entails that the perpetual insertion of quarters by one customer in one machine in no way prevents the other customer to insert dimes in the other machine.

In [34] Walker shows that once Process A executes instruction ℓ_2, it will in fact enter the critical section, i.e. execute ℓ_4. This proof assumes progress, but not justness, let alone fairness. The only question left is whether we can guarantee that execution of ℓ_1 is always followed by ℓ_2. Thus, when assuming progress, the only possible counterexample to the intended liveness property of Peterson's algorithm is the execution path ρ sketched above, and its symmetric counterpart. This execution represents a battle for the shared variable *readyA*. Process A tries to assign a value to this variable, whereas Process B engages in an unending sequence of read actions of this variable (as part of infinitely many instructions m_4). If we assume that the central memory in which variable *readyA* is stored implements its own mutual exclusion protocol, which prevents two processes from reading and writing the same variable at the same time (but guarantees no liveness property), we may have the situation that Process A has to wait before setting *readyA* to *true* until Process B is done reading this variable. However, Process B may be so quick that each time it is done reading *readyA* it executes m_5–m_3 in the blink of an eye and grabs hold of the same variable for reading it again before Process A gets a chance to write to it. Under this assumption we would conclude that Peterson's algorithm does not have the required liveness property, since Process A may never get a chance to write to *readyA* because Process B is too busy reading it, and hence never ever enters the critical section.

However, it is reasonable to assume that in the intended setting where Peterson's algorithm would be employed, the central memory does *not* employ its own mutual exclusion protocol that prevents one process from writing a variable while another is reading it.[4] With this view of the central memory in mind, instruction ℓ_2 cannot be blocked by Process B, and hence the assumption of justness is sufficient to ensure that Peterson's protocol *does* have the required liveness property.

The same conclusion cannot be drawn for the rendering of Peterson's algorithm in CCS. Here the write action $\ell_2 = \overline{asgn^{true}_{readyA}}$ needs to synchronise with the action $asgn^{true}_{readyA}$ of the shared memory storing variable *readyA*. That process has to make a choice between executing $asgn^{true}_{readyA}$ and executing $\overline{n^{false}_{readyA}}$, the latter in synchronisation with Process B . When it chooses $\overline{n^{false}_{readyA}}$ it has to wait until this instruction is terminated before the same choice arises again. Hence the write action can be blocked by the read action, and justness is not strong enough an assumption to ensure that eventually the assignment will

[4] Without such a protocol, it could be argued that the reading process may read *anything* when reading overlaps with writing the same variable. However, the variable *readyA* has only two possible values that can be read, and depending on which of these is returned, the overlapping read action may just as well be thought to occur before or after the write action.

take place. Assuming fairness is of course enough to achieve this, but risky since it has the potential to rule out realistic behaviour (see above).

The above reasoning merely shows that the given implementation of Peterson's mutual exclusion protocol in CCS requires fairness to be correct. In [20] we show that the same holds for any implementation of any mutual exclusion protocol in CCS, and the same argument applies to a wider class of process algebras. Peterson expressed his protocol in pseudocode without resorting to a fairness assumption. We understand that he assumes progress and justness implicitly, and accordingly his protocol and liveness claim are correct. It follows that Peterson's pseudocode does not admit an accurate translation into CCS.

5 Formalising Progress and Justness

Liveness properties are naturally expressed as properties of execution paths. A *path* of a process P is an alternating sequence $P \xrightarrow{\alpha_1} P_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} P_n \xrightarrow{\alpha_{n+1}} \ldots$ of states and transitions. A path can be finite or infinite. A possible formulation of the liveness property of Peterson's algorithm, applied to paths π, is that each occurrence of a transition labelled with **noncritA** in π is followed by an occurrence of **critA**, and similarly for B. To express when a liveness property holds for a system P, we need the notion of a *complete* path: one that describes a complete execution of P, rather than a partial one. The property holds for P iff it holds for all its complete paths. Progress, justness and fairness assumptions rule out certain paths from being considered complete—those that are in disagreement with the assumption. The stronger the assumption, the more paths are ruled out, and the more likely it is that a given liveness property holds.

A complete path ending in a state P_n models a system run in which no further activity takes place after P_n has been reached. A complete path ending in a transition models a system run where transitions are considered to have a duration, and the final transition commenced, but never finishes.

One assumption we adopt in this paper is that "atomic actions always terminate" [30]. It rules out all paths ending in a transition. To check whether Peterson's algorithm is compatible with this assumption, we note that processes are not allowed to stay forever in their critical sections, so the actions **critA** and **critB** can be assumed to terminate. Read and write actions of variables terminate as well. However, a process is allowed to stay forever in its noncritical section, so the actions **noncritA** and **noncritB** need not terminate. To make our formalisation of Peterson's algorithm compatible with the assumption that actions terminate, we could split the action **noncritA** into **start**(**noncritA**) and **end**(**noncritA**). Both these actions terminate, and an execution in which Process A stays in its noncritical section corresponds with a complete path that ends in the state between these transitions. To save the effort of rewriting the protocol from Sect. 3, we shall identify instruction ℓ_6 with entering the noncritical section, and interpret ℓ_1 as leaving the noncritical section. Thus, the processes start out being in their noncritical sections.

To formalise the assumptions of progress and justness, we need the concept of a *non-blocking* action. A process of the form $\tau.P$ should surely execute the internal action τ, and not stay forever in its initial state. However, a process $a.P$ running in the environment $(\_ \mid E)\backslash\{a\}$ may very well stay in its initial state, namely when the environment E never provides a signal $\bar{a}$ that the process can read. With this in mind we assume a classification of the set of actions into blocking and non-blocking actions [21]. The internal action τ is always non-blocking, and any action a classified as non-blocking shall never be put in the scope of a restriction operator $\backslash L$ with $a \in L$, and never be renamed into a blocking action [20]. The transition system of Peterson's algorithm features actions **critA**, **critB**, **noncritA**, **noncritB** and τ—other names are forbidden by the restriction operator. We classify **critA**, **critB** and τ as non-blocking, but the actions **noncritA** and **noncritB** of leaving the noncritical sections may block. Our progress

assumption rules out as complete any path ending in a state in which a non-blocking action is enabled, i.e. any system state except for the ones where both Processes A and B are (back) in their initial state.

The (stronger) *justness assumption* from [21] is:

> *If a combination of components in a parallel composition is in a state that admits a non-blocking action, then one (or more) of them will eventually partake in an action.*

Its formalisation uses *decomposition*: a transition $P|Q \xrightarrow{\alpha} R$ derives, through the rules of Table 1, from

- a transition $P \xrightarrow{\alpha} P'$ and a state Q, where $R = P'|Q$,
- two transitions $P \xrightarrow{a} P'$ and $Q \xrightarrow{\bar{a}} Q'$, where $R = P'|Q'$ and $\alpha = \tau$,
- or from a state P and a transition $Q \xrightarrow{\alpha} Q'$, where $R = P|Q'$.

This transition/state, transition/transition or state/transition pair is called a *decomposition* of $P|Q \xrightarrow{\alpha} R$; it need not be unique. A *decomposition* of a path π of $P|Q$ into paths π_1 and π_2 of P and Q, respectively, is obtained by decomposing each transition in the path, and concatenating all left-projections into a path of P—the *decomposition of π along P*—and all right-projections into a path of Q. It could be that a path π is infinite, yet either π_1 or π_2 (but not both) are finite. Decomposition of paths need not be unique.

Similarly, any transition $P[f] \xrightarrow{\alpha} R$ stems from a transition $P \xrightarrow{\beta} P'$, where $R = P'[f]$ and $\alpha = f(\beta)$. This transition is called a decomposition of $P[f] \xrightarrow{\alpha} R$. A *decomposition* of a path π of $P[f]$ is obtained by decomposing each transition in the path, and concatenating all transitions so obtained into a path of P. A decomposition of a path of $P\backslash L$ is defined likewise.

Definition 1 The class of *Y-just* paths, for $Y \subseteq \mathscr{H}$, is the largest class of paths in $\mathscr{T}_{\mathrm{CCS}}$ such that

- a finite Y-just path ends in a state that admits actions from Y only;
- a Y-just path of a process $P|Q$ can be decomposed into an X-just path of P and a Z-just path of Q such that $Y \supseteq X \cup Z$ and $X \cap \bar{Z} = \emptyset$—here $\bar{Z} := \{\bar{c} \mid c \in Z\}$;
- a Y-just path of $P\backslash L$ can be decomposed into a $Y \cup L \cup \bar{L}$-just path of P;
- a Y-just path of $P[f]$ can be decomposed into an $f^{-1}(Y)$-just path of P;
- and each suffix of a Y-just path is Y-just.

A path π is *just* if it is Y-just for some set of blocking actions $Y \subseteq \mathscr{H}$. A just path π is *a-enabled* for an action $a \in \mathscr{H}$ if $a \in Y$ for all Y such that π is Y-just.

Intuitively, a Y-just path models a run in which Y is an upper bound of the set of labels of abstract transitions[5] that from some point onwards are continuously enabled but never taken. Here an abstract transition with a label from $\mathscr{H}$ is deemed to be continuously enabled but never taken iff it is enabled in a parallel component that performs no further actions. Such a run can occur in the modelled system if the environment from some point onwards blocks the actions in Y.

Now consider the path ρ violating the intended liveness property. The decompositions of ρ along Processes A and B were mentioned in Sect. 4. These paths are $\{\overline{asgn^{true}_{readyA}}\}$-just and $\emptyset$-just, respectively. The decomposition along $Turn^A$ is an infinite path taking action $asgn^A_{turn}$ only ($\emptyset$-just). The decomposition along $ReadyB^{false}$ is an infinite path alternatingly taking actions $asgn^{true}_{readyB}$ and $asgn^{false}_{readyB}$ (also $\emptyset$-just) and the decomposition along $ReadyA^{false}$ is an infinite path taking action $\overline{n^{false}_{readyA}}$ only (again $\emptyset$-just). It follows that the composition ρ of these five paths is $\emptyset$-just. Intuitively this is the case because no communication is permanently enabled and never taken. In particular, the communication $\overline{asgn^{true}_{readyA}}$ is disabled each time the component $ReadyA^{false}$ does the action $\overline{n^{false}_{readyA}}$ instead.

[5] The CCS process $a.0|b.0$ has two transitions labelled a, namely $a.0|b.0 \xrightarrow{a} 0|b.0$ and $a.0|0 \xrightarrow{a} 0|0$. The only difference between these two transitions is that one occurs before the action b is performed by the parallel component and the other afterwards. In [21] we formalise a notion of an *abstract transition* that identifies these two concrete transitions.

$$(P\hat{}s)\overset{\sim}{}^{s} \qquad \frac{P \xrightarrow{\alpha} P'}{P\hat{}s \xrightarrow{\alpha} P'} \qquad \frac{P\overset{\sim}{}^{s}}{(P\hat{}t)\overset{\sim}{}^{s}} \qquad \frac{P_j\overset{\sim}{}^{s}}{(\sum_{i\in I} P_i)\overset{\sim}{}^{s}}\ (j\in I)$$

$$\frac{P\overset{\sim}{}^{s}}{(P|Q)\overset{\sim}{}^{s}} \qquad \frac{P\overset{\sim}{}^{s},\ Q \xrightarrow{s} Q'}{P|Q \xrightarrow{\tau} P|Q'} \qquad \frac{P \xrightarrow{s} P',\ Q\overset{\sim}{}^{s}}{P|Q \xrightarrow{\tau} P'|Q} \qquad \frac{Q\overset{\sim}{}^{s}}{(P|Q)\overset{\sim}{}^{s}}$$

$$\frac{P\overset{\sim}{}^{s}}{(P\backslash L)\overset{\sim}{}^{s}}\ (s\notin L) \qquad \frac{P\overset{\sim}{}^{s}}{P[f]\overset{\sim}{}^{f(s)}} \qquad \frac{P\overset{\sim}{}^{s}}{A\overset{\sim}{}^{s}}\ (A \overset{def}{=} P)$$

Table 2: Structural operational semantics for signals of CCSS

6 CCS with Signals

We would like to prevent such a path to be complete in a CCS model of Peterson's algorithm. In order to achieve this, we propose to replace an action such as $\overline{n^{false}_{readyB}}$, which makes the variable busy even if it is only read, by a state predicate providing its value. This mode of communication is called *signalling*.

CCS with signals (CCSS) is CCS extended with a signalling operator. Informally, the signalling operator $P\hat{}s$ emits the signal s to be read by another process. Signal emission cannot block other actions. Formally, CCS is extended with a set $\mathscr{S}$ of *signals*, ranged over by $s, t, \ldots$. In CCSS the set of actions is defined as $Act := \mathscr{S} \cup \mathscr{H} \cup \{\tau\}$. A relabelling is a function $f : (\mathscr{S} \to \mathscr{S}) \cup (\mathscr{H} \to \mathscr{H})$ satisfying $f(\bar{a}) = \overline{f(a)}$. As before it extends to Act by $f(\tau) = \tau$.

The class $\mathscr{T}_{\mathrm{CCSS}}$ of *CCSS expressions* is defined as the smallest class that includes

- *agent identifiers* $A \in \mathscr{K}$;
- *parallel compositions* $P|Q$;
- *signallings* $P\hat{}s$
- *prefixes* $\alpha.P$;
- *restrictions* $P\backslash L$;
- *(infinite) choices* $\sum_{i\in I} P_i$;
- *relabellings* $P[f]$;

where $P, P_i, Q \in \mathscr{T}_{\mathrm{CCSS}}$ are CCSS expressions, I an index set, $L \subseteq \mathscr{A} \cup \mathscr{S}$ a set of handshake names and signals, f an arbitrary relabelling function, and $s \in \mathscr{S}$ a signal. The new operator $\hat{}$ binds as strong as relabelling and restriction.

The semantics of CCSS is given by the labelled transition relation $\to \subseteq \mathscr{T}_{\mathrm{CCSS}} \times Act \times \mathscr{T}_{\mathrm{CCSS}}$ and a predicate $\overset{\sim}{} \subseteq \mathscr{T}_{\mathrm{CCSS}} \times \mathscr{S}$ that are derived from the rules of CCS (Table 1, where α can also be a signal) and the new rules of Table 2. The predicate $P\overset{\sim}{}^{s}$ indicates that process P emits the signal s, whereas a transition $P \xrightarrow{s} P'$ indicates that P reads the signal s and thereby turns into P'. The first rule is the base case showing that a process $P\hat{}s$ emits the signal s. The second rule models the fact that signalling cannot prevent a process from making progress. After having taken an action, the signalling process loses its ability to emit the signal. It is essentially this rule which fixes the read/write problem presented in the previous section. The two rules in the middle of Table 2 state that the action of reading a signal by one component in (parallel) composition together with the emission of the same signal by another component, results in an internal transition τ; similar to the case of handshake communication. Note that the component emitting the signal does not change through this interaction. All the other rules of Table 2 lift the emission of s by a subprocess P to the overall process. Table 2 can easily be adapted to other process calculi, hence our extension is not limited to CCS.

We give an example similar to the one at the end of Sect. 2 to illustrate the use of signals.

Example 2 We describe once again a one-variable shared memory system with an infinite reader R and a single writer W. But this time communication actions n^v_x and $\overline{n^v_x}$ are replaced with signals n^v_x. The variable x now emits a signal notifying its value, so we have: $x^{true} \overset{def}{=} (asgn^{true}_x . x^{true} + asgn^{false}_x . x^{false})\hat{}n^{true}_x$

and $x^{false} \stackrel{def}{=} (asgn_x^{true} \,.\, x^{true} + asgn_x^{false} \,.\, x^{false}) \hat{} n_x^{false}$; the rest of the example remains unchanged. The transition system is exactly the same, but now justness guarantees that the variable x will eventually be set to *false*. This is in contrast to Ex. 1, where it is not guaranteed that x will eventually be *false*, even when assuming justness. More precisely, if only the reader takes actions, x^{true} is now not progressing because it is emitting a signal only, and then, assuming justness, it must eventually enter into communication with the writer.

As we have extended CCS with a novel operator, we have to make sure that our extension behaves 'naturally', in the way one would expect.

Theorem 1 *Strong bisimilarity [19] is a congruence for all operators of CCSS.*

Proof. We get the result directly from the existing theory on structural operational semantics, as a result of carefully designing our language. All rules of Tables 1 and 2 are in the *path* format of Baeten and Verhoef [3], and hence the theorem holds [3]. □

Theorem 2 *The operator | is associative and commutative, and the operator ˆ is pseudo-commutative, i.e. $P\hat{}s\hat{}t = P\hat{}t\hat{}s$, all up to bisimilarity.*

Proof. Our process algebra with predicates can easily be encoded in a process algebra without, by writing

$$P \xrightarrow{\bar{s}} P \text{ for } P^{\curvearrowright s} \ .$$

On the level of the structural operational semantics, this amounts to letting α range over $Act \cup \{\bar{s} \mid s \in \mathscr{S}\}$ in the rules of Table 1, and changing the first rule of Table 2 into $P\hat{}s \xrightarrow{\bar{s}} P\hat{}s$. The third rule of Table 2 becomes an instance of the second (with $\alpha \in Act \cup \{\bar{s} \mid s \in \mathscr{S}\}$), and the remaining rules of Table 2 become special cases of the rules of Table 1.

Clearly, two processes are bisimilar in the original CCSS iff they are bisimilar in this encoding. Since the parallel composition of the encoded CCSS is the same as the one of CCS, it is known to be associative and commutative up to bisimilarity [26].

To prove pseudo-commutativity of ˆ, we note that $P\hat{}s\hat{}t$ and $P\hat{}t\hat{}s$ have exactly the same outgoing transitions and signals, thereby being trivially equal up to bisimilarity. □

Since we extended CCS, we also have to extend our definition of justness. The decomposition of paths remains unchanged, except that a transition $P|Q \xrightarrow{\tau} R$ can now derive, through the rules of Table 2, from signal communication. In that case we consider the decomposition along the signalling process empty, just as if it was an application of the left- or right-parallel composition rule. Because processes can communicate through signalling, we first introduce the definition of signalling paths. Informally, a path emits signal s if one component in the parallel composition ends in a state where signal s is activated. A Y-signalling path is a path where Y is an upper bound on the signals emitted by the path.

Definition 2 The class of *Y-signalling* paths, for $Y \subseteq \mathscr{S}$, is the largest class of paths in $\mathscr{T}_{\mathrm{CCS}}$ such that

- a finite Y-signalling path ends in a state that admits signals from Y only;
- a Y-signalling path of a process $P \,|\, Q$ can be decomposed into an X-signalling path of P and a Z-signalling path of Q such that $Y \supseteq X \cup Z$;
- a Y-signalling path of $P \backslash L$ can be decomposed into a $Y \cup L_{\mathscr{S}}$-signalling path of P—here $L_{\mathscr{S}} := L \cap \mathscr{S}$ restricts the set L to signals;
- a Y-signalling path of $P[f]$ can be decomposed into an $f^{-1}(Y)$-signalling path of P;
- and each suffix of a Y-signalling path is Y-signalling.

Using this definition, we can adapt the definition of justness of Sect. 5.

Definition 3 The class of *Y-just* paths, for $Y \subseteq \mathscr{H} \cup \mathscr{S}$, is the largest class of paths in $\mathscr{T}_{\mathrm{CCSS}}$ such that

- a finite Y-just path ends in a state that admits actions from Y only;
- a Y-just path of a process $P|Q$ can be decomposed into a path of P that is X-just and X'-signalling, and a path of Q that is Z-just and Z'-signalling, such that $Y \supseteq X \cup Z$, $X \cap \bar{Z}_{\mathscr{H}} = \emptyset$, $X \cap Z' = \emptyset$ and $X' \cap Z = \emptyset$—here $\bar{Z}_{\mathscr{H}} := \{\bar{a} \mid a \in Z \cap \mathscr{H}\}$;
- a Y-just path of $P\backslash L$ can be decomposed into a $Y \cup L \cup \bar{L}_{\mathscr{H}}$-just path of P;
- a Y-just path of $P[f]$ can be decomposed into an $f^{-1}(Y)$-just path of P;
- and each suffix of a Y-just path is Y-just.

As before, a path π is *just* if it is Y-just for some set of blocking actions and signals $Y \subseteq \mathscr{H} \cup \mathscr{S}$. A just path π is *a-enabled* for $a \in \mathscr{H} \cup \mathscr{S}$ if $a \in Y$ for all Y such that π is Y-just.

The condition on signals in the second item guarantees that a process $(\mathbf{0}\hat{}s \mid s.\mathbf{0})\backslash\{s\}$ makes progress.

The encoding in the proof of Theorem 2 does not preserve justness. In Ex. 2, for instance, applying the operational semantics of Table 2, the path ρ_R involving infinitely many read actions but no write action is not just, because its decomposition along x^{true} is finite and $asgn_x^{false}$-enabled, whereas its decomposition along W is $\overline{asgn_x^{false}}$-enabled; so by the second clause of Def. 3 ρ_R is not just: there are no Y, X and Z such that the condition $X \cap \bar{Z}_{\mathscr{H}} = \emptyset$ is satisfied. Yet, after applying the encoding in the proof of Theorem 2, the decomposition along x^{true} becomes infinite and $\emptyset$-just, and ρ_R becomes just. This is the main reason we did not present the semantics of CCSS in this form from the onset.

7 Peterson's Mutual Exclusion Protocol—Part II

We now present an implementation of Peterson's mutual exclusion algorithm in CCSS. We use the same notation as in Sect. 3, except that actions n_x^v and $\overline{n_x^v}$ are replaced with signals n_x^v, just as in Ex. 2. Only the variable processes change, such as $Turn^{\mathrm{A}} \stackrel{def}{=} (asgn_{turn}^{\mathrm{A}}.Turn^{\mathrm{A}} + asgn_{turn}^{\mathrm{B}}.Turn^{\mathrm{B}})\hat{}n_{turn}^{\mathrm{A}}$; Processes A and B are unchanged. The protocol rendering is still $(\mathrm{A} \mid \mathrm{B} \mid ReadyA^{false} \mid ReadyB^{false} \mid Turn^{A})\backslash L$, where L is the set of all names and signals except **noncritA**, **critA**, **noncritB**, and **critB**, as before.

In the remainder of this section we prove Peterson's protocol correct, i.e. safe and live. We include the proof of safety for completeness, but concentrate on liveness.

Theorem 3 *Peterson's protocol is safe. In terms of Fig. 1, there is no reachable state where* A *and* B *have already executed lines ℓ_4 and m_4 but have not yet executed ℓ_6 or m_6.*

Proof. We follow the proof by contradiction of Peterson [31]. Suppose both processes succeed the test at ℓ_4 and m_4. Let A be the first to pass this test. At that time either *readyB* was false (meaning that Process B was between m_6 and m_2) or $turn$ was set to A. In the first case, *readyA* will not be set to false before Process A leaves the critical section and *turn* is bound to be set to A by Process B before m_4 is executed. So the test at m_4 will fail. In the second case, since A is about to enter the critical section, *turn* cannot be set to B anymore and *readyA* is *true*, so again the test m_4 will fail for Process B. □

Peterson's protocol satisfies also the liveness property. As mentioned before, this result could not be proven for the formalisation of the protocol in CCS, assuming justness only.

Theorem 4 *Assuming justness, Peterson's protocol satisfies the liveness property: on each just path, each occurrence of* **noncritA** *is followed by* **critA** *(and similarly for* B*).*

Proof. Let π be a just path of the protocol. Since **noncritA** and **noncritB** are the only possible blocking actions, π must be $\{\mathbf{noncritA}, \mathbf{noncritB}\}$-just. If we get rid of all the restrictions we obtain a Y-just path

of (A | B | $ReadyA^{false}$ | $ReadyB^{false}$ | $Turn^{A}$) where $Y = \{\mathbf{noncritA}, \mathbf{noncritB}\} \cup L \cup \bar{L}_{\mathscr{H}}$. Suppose its decomposition π_A along Process A ends somewhere between instructions ℓ_1 and ℓ_4. Then the decomposition π_{ReadyA} along $ReadyA^{false}$ is also finite since only A can communicate with this process. Using the CCS rendering from Sect. 3 this statement would be incorrect, since there Process B can constantly interact with $ReadyA^{false}$, by reading its value; resulting in an infinite path $ReadyA^{false} \circlearrowleft n^{false}_{readyA}$.

By Def. 3, the path π_A must be X-just, and the path π_{ReadyA} Z-just, for sets $X, Z \subseteq Y$ with $\overline{X \cap \bar{Z}_{\mathscr{H}}} = \emptyset$. Furthermore, $asgn^{true}_{readyA} \in Z$, since this action is enabled in the last state of π_{ReadyA}. Hence $\overline{asgn^{true}_{readyA}} \notin X$. Therefore π_A cannot end right before instruction ℓ_2. As a result, Process A is stuck either right before ℓ_3, or right before ℓ_4. In both cases Process B would not be able to pass the test before the critical section more than once. Indeed, in either case *readyA* is already set to *true*, thus Process B must use *turn* = *B* to enter its critical section. But, if trying to enter a second time, it would be forced to set *turn* to A and will be stuck. When Processes A and B are both stuck, the path π is finite and an action τ or **noncritA** stemming from instruction ℓ_3 or ℓ_4 is enabled at the end, contradicting, through the first clause of Def. 3, the $\{\mathbf{noncritA}, \mathbf{noncritB}\}$-justness of π. □

8 Peterson's Algorithm for N Processes

In the previous section we presented an implementation in CCSS of Peterson's algorithm of mutual exclusion for two processes. In [31], Peterson also presents a generalisation of his mutual exclusion protocol to N processes. In this section we describe the algorithm and explain which assumptions should be made on the memory model in order for this protocol to be correct, for $N>2$. We claim that these assumptions are somewhat unrealistic.

A pseudocode rendering of Peterson's protocol is depicted in Fig. 2. In order to proceed to the critical section, each process must go through $N-1$ locks (*rooms*). The shared variable $room[i] = j$ indicates that process number i is currently in Room j. The shared variable $last[j] = i$ indicates that the last process to '*enter*' Room j is Process i. A process can go to the next room if and only if it is not the last one to have entered the room, or if all other processes are strictly behind it. This algorithm is also called the *filter lock* because it ensures that for all j, no more than $N+1-j$ processes are in rooms greater or equal than j. The critical section can be thought of as Room N.

A natural memory model, used in [25], stipulates that memory accesses from different components can overlap in time, and that a read action that overlaps with a write action of the same variable may yield *any* value. Extending this idea, we assume that when two concurrent write actions overlap, *any* possible value could end up in the memory. We argue that the algorithm fails to satisfy mutual exclusion when assuming such a model.

Process i $(i \in \{1, \ldots, N\})$

repeat forever

ℓ_1 **noncritical section**
ℓ_2 **for** j **in** $1 \ldots N-1$
ℓ_3 $\quad room[i] := j$
ℓ_4 $\quad last[j] := i$
ℓ_5 $\quad$ **await**$(last[j] \neq i \vee (\forall k \neq i,\ room[k] < j))$
ℓ_6 **critical section**
ℓ_7 $room[i] := 0$

Figure 2: Peterson's algorithm for N processes (pseudocode)

Suppose there are three processes, A, B and C, and Processes A and B execute ℓ_1–ℓ_4 more or less simultaneously. When their instructions ℓ_4 overlap, the value C ends up in the variable $last[1]$—or any other value different from A and B. Hence they both perform ℓ_5, as well as ℓ_3–ℓ_4 for j=2. Again, the value C ends up in $last[2]$. Subsequently, they both enter their critical section, and disaster strikes.

It follows that Peterson's algorithm for $N>2$ only works when running on a memory where write actions cannot overlap in time, or—if they do—their effect is the same as when one occurred before the other. Such a memory can be implemented by having a small hardware lock around a write action to the same variable. This entails that one write action would have to wait until the other one is completed. A memory model of this kind is implicitly assumed in process algebras like CCS(S).

We show that, under such a memory model, Peterson's algorithm for $N>2$ does not satisfy liveness, unless we enrich it with an additional fairness assumption.

To prove this statement, let $N = 3$ and call the processes A, B and C. We show that (without the additional assumption) Process A can be stuck at ℓ_4 for j=1. Suppose Process A is at this line. Then A is about to set $last[1]$ to A, but has not written yet. We can imagine the following scenario: Process B enters Room 1, and sets $last[1]$ to B; then Process C enters Room 1, and sets $last[1]$ to C. This allows B to proceed to Room 2, then to go in the critical section (because all other processes are still in room 1), and then to go back to Room 1, setting $last[1]$ to B. This allows C to go to Room 2, to the critical section, and back to room 1, setting $last[1]$ to C. Next B can enter the critical section again, etc. Hence Processes B and C can go alternately in the critical section without giving A a chance to set variable $last[1]$. (The variable is too busy being written by B and C.) This scenario cannot happen for $N = 2$ because after B sets $last[1]$ to B, B is blocked until A sets it to A; so ℓ_4 will eventually happen (with progress as a basic assumption).

As a consequence, in order for Peterson's algorithm to be live for more than two processes, we must adopt the additional fairness assumption that *if a process permanently tries to write to a variable, it will eventually do so*, even if other processes are competing for writing to the same variable. This property appears to be at odds with having a hardware lock around the shared variable. Moreover, it cannot be implemented in CCSS assuming only justness: when two competitive processes try to write the same variable, nothing guarantees that both will eventually succeed.[6] As a result any CCSS-rendering of Peterson's algorithm for N processes does not possess the liveness property, unless one makes a fairness assumption. The problem comes from the fact that the variables $last[\cdot]$ are written by several parallel processes. Signals only allow a writer to set a variable while it is being read but do not allow multiple writers at the same time.

We believe that the problem does not come from a lack of expressiveness of CCSS but from the protocol, which, while not being incorrect in itself, requires a memory model that assumes write actions to happen eventually, even though simultaneous interfering write actions are excluded; whether this is a realistic assumption on modern hardware requires further investigation.

9 Lamport's Bakery Algorithm

In this section we analyse Lamport's bakery algorithm [25], another mutual exclusion protocol for N processes. It has the property that processes write to separate variables; only the read actions are shared. We give a model for this algorithm in CCSS and prove its liveness property, assuming justness only.

[6] Let us consider a CCSS process $(x^{true}|W_1|W_2)\backslash L$ where processes W_1 and W_2 are infinite writers ($W_i \stackrel{def}{=} \overline{asgn_x^{false}}.W_i$) and L is the set of communication names. A path where W_1 always succeeds, meaning that the decomposition along W_2 is empty, is just because the latter decomposition is $\{asgn_x^{false}\}$-just and all the other decompositions $\emptyset$-just.

Process i $(i \in \{1,\dots,N\})$

repeat forever

ℓ_1 **noncritical section**
ℓ_2 $choosing[i] := true$
ℓ_3 $number[i] := 1 + \max(number[1],\dots,number[N])$;
ℓ_4 $choosing[i] := false$
ℓ_5 **for** j **in** $1 \dots N$
ℓ_6 **await**$(choosing[j] = false)$
ℓ_7 **await**$(number[j] = 0 \vee (number[i], i) \leq (number[j], j))$
ℓ_8 **critical section**
ℓ_9 $number[i] := 0$

Figure 3: Lamport's bakery algorithm for N processes (pseudocode)

A pseudocode rendering of Lamport's bakery algorithm is depicted in Fig. 3. Lines 2–4 are called the *doorway* and lines 5–7 are called the *bakery*. In the doorway each process 'takes a ticket' that has a number strictly greater than all the numbers from the other processes (at the time the process reads them). The variable $choosing[i]$ is a lock that makes line 3, which is usually implemented by a simple loop, more or less 'atomic'. To ensure that the holder of the lowest number is next in the critical section, each process goes through a number of locks in the bakery (Lines 5–7). When process i enters the critical section, the value it has read for $number[j]$, if not 0, is greater or equal than its own $number[i]$, for all j.

We now model this algorithm in CCSS. As usual, we define one agent for every pair (variable, value). The variables *choosing* can take values *true* or *false*, and *number* any non-negative value. The modelling of a Boolean variable is addressed in Ex. 1, and for the integer variables we define:

$$number[i]^k \stackrel{def}{=} \Big(\sum_{l\in\mathbb{N}} asgn^l_{number[i]} \,.\, number[i]^l\Big) \hat{}\, n^k_{number[i]} \,.$$

Each process i begins with a non-critical section before entering the doorway.

$$P_i \stackrel{def}{=} \mathbf{noncrit[}i\mathbf{]} \,.\, \overline{asgn^{true}_{choosing[i]}} \,.\, doorway[i]^1_0$$

Line 3 encodes several read actions, an arithmetic operation, and an assignment in a single step. In CCS(S) (and most programming languages) this command is modelled by several atomic steps, e.g. by the simple loop $m := 0$; **for** j **in** $1 \dots N\{m := \mathtt{max}(m, number[j])\}$; $number[i] := 1 + m$. We define processes $doorway[i]^j_m$ that represent the state of being in the doorway **for**-loop for a process i with loop index j and local variable m storing the current maximum.

$$doorway[i]^j_m \stackrel{def}{=} \Big(\sum_{k>m} n^k_{number[j]} \,.\, doorway[i]^{j+1}_k\Big) + \Big(\sum_{k\leq m} n^k_{number[j]} \,.\, doorway[i]^{j+1}_m\Big) \;,\; j \in \{1,\dots,N\}$$

We then define $doorway[i]^{N+1}_m$, which represents the termination of the **for**-loop by

$$doorway[i]^{N+1}_m \stackrel{def}{=} \overline{asgn^{m+1}_{number[i]}} \,.\, \overline{asgn^{false}_{choosing[i]}} \,.\, bakery[i]^1_{m+1} \,.$$

The process $bakery[i]^j_m$ represents the state of being in the bakery **for**-loop for process i with loop index j and $number[i] = m$. For $j \in \{1,\dots,N\}$:

$$bakery[i]^j_m \stackrel{def}{=} n^{false}_{choosing[j]} \,.\, \Big(n^0_{number[j]} + \sum_{k>m\vee(k=m\wedge j\geq i)} n^k_{number[j]}\Big) \,.\, bakery[i]^{j+1}_m \,.$$

Finally, $bakery[i]_m^{N+1}$ is the exit of the bakery **for**-loop, granting access to the critical section:

$$bakery[i]_m^{N+1} = \mathbf{crit[}i\mathbf{]} \,.\, \overline{asgn^0_{number[i]}} \,.\, P_i$$

Our bakery algorithm is the parallel composition of all processes P_i, in combination with the shared variables $choosing[i]$ and $number[i]$, restricting the communication actions:

$$\Big(\Big| _{i\in\{1,\ldots,N\}} (P_i \mid choosing[i]^{false} \mid number[i]^0) \Big) \backslash L \,,$$

where L is the set of all names and signals except **noncrit[**i**]** and **crit[**i**]**.

We now prove the liveness of (our rendering of) the algorithm, given that it is straightforward to adapt Lamport's proof of safety of the pseudocode [25] to CCSS. Since every process writes in its own variables, no process can be stuck because of concurrent writing. Therefore, the only possibility for a process (call it A) to be stuck is when trying to read a variable, so at ℓ_3, ℓ_6 or ℓ_7.

If Process A is stuck at ℓ_3, trying to read $number$[B] for some process B, B will get stuck at ℓ_6 for j=A, because $choosing$[A] remains *false*. So, Process B cannot be perpetually busy writing $number$[B], and A cannot be stuck at ℓ_3.

If A is stuck at ℓ_6, then from the point of view of A, some process B is all the time in the doorway. It follows from the argument above that B cannot be stuck in one visit to its doorway, so it must be a repeating series of visits. This is impossible because when A tries to read $choosing$[B] for the first time, the value of $number$[A] is set and will not change anymore, so if B goes back to the doorway, it is bound to set $number$[B] > $number$[A] and will not be able to enter the critical section anymore.

Suppose that Process A is stuck at ℓ_7. Any process B that enters the doorway will receive a $number$[B] strictly larger than $number$[A] and be stuck in the bakery. So if A is stuck, eventually all processes are stuck at ℓ_7, which is impossible since every finite lexicographically ordered set has a minimal element.

10 Conclusion, Related Work and Outlook

This paper presents a minimal extension of CCS in which Peterson's mutual exclusion protocol can be modelled correctly, using a justness assumption only. The signalling operator allows processes to emit signals that can be received by other processes. The signalling process is not blocked by the emission of the signal, which means that its actions are in no way postponed or affected by other processes reading the signal. This property is crucial to correctly model mutual exclusion.

Our process algebra, *CCS with signals*, is strongly inspired by, and can be regarded as a simplification of, Bergstra's *ACP with signals* [4]. The idea of a signal as a predicate on states, rather than a transition between states, stems from that paper. However, the non-blocking nature of signals was not explored by Bergstra, who writes "The relevance of signals is not so much that process algebra without signals lacks expressive power". This point is disputed in the current paper.

CCS with signals is not the first process algebra with explicitly non-blocking read actions. In [10] Corradini, Di Berardini & Vogler add a similar operator to PAFAS [12], a process algebra for modelling timed concurrent systems. The semantics of this extension is justified in [11]. They show [10] that this enables the liveness property of Dekker's mutual exclusion algorithm [13, 15], modelled in PAFAS, when assuming *fairness of actions*, and in [8] they establish the same for Peterson's algorithm, while showing that earlier mutual exclusion algorithms by Dijkstra [14] and Knuth [24] lack the liveness property under fairness of actions. Fairness of actions is similar to our notion of justness—although formalised in a

quite different way—except that all actions are treated as being non-blocking. The notion of time plays an important role in the formalisation of the results in [10, 8], even if it is not used quantitatively. Our process algebra can be regarded as a conceptual simplification of this approach, as it completely abstracts from the concept of time, and hence is closer to traditional process algebras like CCS and CSP.

The accuracy of our extension depends highly on which memory model is considered as realistic. It is well known that in weak or relaxed memory models, mutual exclusion protocols like Peterson's or the bakery algorithm do not behave correctly; when employing a weak memory model, mutual exclusion is handled on the hardware layer only—this is not covered here. An extremely plausible memory model allows parallel non-blocking writing, but admits any value being written when two parallel write actions overlap. This memory model is compatible with the bakery algorithm, and with Peterson's algorithm for two processes, but—as we show—not for Peterson's algorithm with $N \geq 3$ processes. Instead one needs a form of sequential consistency, assuming that parallel write actions, or a parallel read/write, behave as if they are executed in either order.

When postulating sequential consistency, it is plausible to assume some kind of mutual exclusion between write actions being implemented in hardware. This in turn allows the possibility of a write action being delayed in perpetuity because other processes are writing to the same variable. Similarly, read actions could be blocked by a consistent flow of write actions. A third type of blocking is that write actions can be obstructed by read actions. However, this kind of blocking is questionable; it could be that during a parallel read/write the write action wins, and only the read action gets postponed.

When assuming all three kinds of blocking, the CCS rendering of mutual exclusion protocols—illustrated in Sect. 3—is fully accurate, and by [20] we conclude that no such protocol can have the intended liveness property. When disallowing write actions being blocked by read actions, but allowing write/write blocking, we get the modelling in CCSS. Using CCSS, we verified the correctness of the bakery algorithm, and Peterson's algorithm for two processes, whereas Peterson's for $N > 2$ fails liveness. The latter protocol becomes correct if we assume sequential consistency without any kind of blocking. Whether this is a realistic memory model on modern hardware needs further investigation. Regardless, we conjecture that such a memory can be modelled in an extension of CCSS with broadcast communication, i.e. the combination of the process algebras presented here and in [21].

The liveness property of Dekker's algorithm, when assuming merely justness, or fairness of actions, requires not only non-blocking reading, but also that repeated assignments to a variable x of the same value cannot block the reading of x [10]. This assumption can be modelled in CCSS, by defining *readyA* of Ex. 1 by $x^{true} \stackrel{def}{=} (asgn_x^{false} \,.\, x^{false})\hat{}n_x^{true}$ and $x^{false} \stackrel{def}{=} (asgn_x^{true} \,.\, x^{true})\hat{}n_x^{false}$, and replacing write actions $\overline{asgn_x^v}$ by $(\overline{asgn_x^v} + n_x^v)$. Alternatively, a pseudocode assignment $x := v$ could be interpreted as **if** $x \neq v$ **then** $x := v$ **fi**.

Although mutual exclusion protocols cannot be modelled in standard Petri nets—when not assuming fairness—[22, 33, 20], it is possible in nets extended with read arcs [33]. This opens the possibility of interpreting CCSS in terms of nets with read arcs, whereas an accurate semantics of CCSS in terms of standard nets is impossible. A read arc from a place to a transition requires the place to be marked for the transition be enabled, but the token is not consumed when the transition is fired. This behaviour really looks like signalling, so we conjecture that a read-arc net semantics of CCSS is fairly straightforward.

Finally, the definition of justness appears complicated because it includes the decomposition of paths. In order to compute if a path (an object from the semantics) is just or not just, we investigate the syntactic shape of the states on that path. It could be that the semantic object—the labelled transition system—is not well adapted to the problem of justness. Giving a semantics to CCSS that inherently includes the decomposition of paths—inspired by [6, 7, 1, 27]—could be an interesting idea for future research.

References

[1] L. Aceto (1994): *A Static View of Localities*. Formal Aspects of Computing 6(2), pp. 201–222, doi:10.1007/BF01221099.

[2] L. Aceto, A. Ingólfsdóttir, K. G. Larsen & J. Srba (2007): *Modelling Mutual Exclusion Algorithms*. In: Reactive Systems: Modelling, Specification and Verification, Cambridge University Press, pp. 142–158, doi:10.1017/CBO9780511814105.008.

[3] J. C. M. Baeten & C. Verhoef (1993): *A Congruence Theorem for Structured Operational Semantics with Predicates*. In E. Best, editor: Proc. *CONCUR '93*, LNCS 715, Springer, pp. 477–492, doi:10.1007/3-540-57208-2_33.

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

[5] A. Bouali (1992): *Weak and Branching Bisimulation in Fctool*. Research Report RR-1575, Inria-Sophia Antipolis. Available at `https://hal.inria.fr/inria-00074985/document`.

[6] G. Boudol, I. Castellani, M. Hennessy & A. Kiehn (1993): *Observing Localities*. Theoretical Computer Science 114(1), pp. 31–61, doi:10.1016/0304-3975(93)90152-J.

[7] G. Boudol, I. Castellani, M. Hennessy & A. Kiehn (1994): *A Theory of Processes with Localities*. Formal Aspects of Computing 6(2), pp. 165–200, doi:10.1007/BF01221098.

[8] F. Buti, M. Callisto De Donato, F. Corradini, M. R. Di Berardini & W. Vogler (2011): *Automated Analysis of MUTEX Algorithms with FASE*. In G. D'Agostino & S. La Torre, editors: Proc. *GandALF '11*, EPTCS 54, Open Publishing Association, pp. 45–59, doi:10.4204/EPTCS.54.4.

[9] F. Corradini, M. R. Di Berardini & W. Vogler (2009): *Liveness of a Mutex Algorithm in a Fair Process Algebra*. Acta Informatica 46(3), pp. 209–235, doi:10.1007/s00236-009-0092-9.

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

[11] F. Corradini, M. R. Di Berardini & W. Vogler (2011): *Read Operators and their Expressiveness in Process Algebras*. In B. Luttik & F. Valencia, editors: Proc. *EXPRESS '11*, EPTCS 64, Open Publishing Association, pp. 31–43, doi:10.4204/EPTCS.64.3.

[12] F. Corradini, W. Vogler & L. Jenner (2002): *Comparing the worst-case efficiency of asynchronous systems with PAFAS*. Acta Informatica 38(11/12), pp. 735–792, doi:10.1007/s00236-002-0094-3.

[13] E. W. Dijkstra (1962 or 1963): *Over de Sequentialiteit van Procesbeschrijvingen*. Available at `http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD35.PDF`. Circulated privately.

[14] 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.

[15] E. W. Dijkstra (1968): *Cooperating Sequential Processes*. In F. Genuys, editor: Programming Languages: NATO Advanced Study Institute, Academic Press, pp. 43–112.

[16] J. Esparza & G. Bruns (1996): *Trapping Mutual Exclusion in the Box Calculus*. Theoretical Computer Science 153(1-2), pp. 95–128, doi:10.1016/0304-3975(95)00119-0.

[17] A. Fehnker, R. J. van Glabbeek, P Höfner, A. K. McIver, M. Portmann & W. L. Tan (2013): *A Process Algebra for Wireless Mesh Networks used for Modelling, Verifying and Analysing AODV*. Technical Report 5513, NICTA. Available at `http://arxiv.org/abs/1312.7645`.

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

[19] R. J. van Glabbeek (2011): *Bisimulation*. In D. Padua, editor: Encyclopedia of Parallel Computing, Springer, pp. 136–139, doi:10.1007/978-0-387-09766-4_149.

[20] 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.

[21] R. J. van Glabbeek & P. Höfner (2015): *Progress, Fairness and Justness in Process Algebra*. CoRR abs/1501.03268. Available at `http://arxiv.org/abs/1501.03268`.

[22] E. Kindler & R. Walter (1997): *Mutex Needs Fairness*. Information Processing Letters 62(1), pp. 31–39, doi:10.1016/S0020-0190(97)00033-1.

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

[24] 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.

[25] 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.

[26] R. Milner (1989): *Communication and Concurrency*. Prentice Hall.

[27] M. Mukund & M. Nielsen (1992): *CCS, Locations and Asynchronous Transition Systems*. In R. K. Shyamasundar, editor: Proc. *FSTTCS '92*, LNCS 652, Springer, pp. 328–341, doi:10.1007/3-540-56287-7_116.

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

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

[30] S. S. Owicki & L. Lamport (1982): *Proving Liveness Properties of Concurrent Programs*. ACM TOPLAS 4(3), pp. 455–495, doi:10.1145/357172.357178.

[31] 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.

[32] A. Valmari & M. Setälä (1996): *Visual Verification of Safety and Liveness*. In M.-C. Gaudel & J. Woodcock, editors: Industrial Benefit and Advances in Formal Methods (FME'96), LNCS 1051, Springer, pp. 228–247, doi:10.1007/3-540-60973-3_90.

[33] W. Vogler (2002): *Efficiency of asynchronous systems, read arcs, and the MUTEX-problem*. Theoretical Computer Science 275(1-2), pp. 589–631, doi:10.1016/S0304-3975(01)00300-0.

[34] D. J. Walker (1989): *Automated analysis of mutual exclusion algorithms using CCS*. Formal Aspects of Computing 1(1), pp. 273–292, doi:10.1007/BF01887209.

Bisimulation and Hennessy-Milner Logic for Generalized Synchronization Trees*

James Ferlez[1,2], Rance Cleaveland[1,3] and Steve Marcus[1,2]
[1] *The Institute for Systems Research, University of Maryland, College Park*
[2] *Department of Electrical and Computer Engineering, University of Maryland, College Park*
[3] *Department of Computer Science, University of Maryland, College Park*

In this work, we develop a generalization of Hennessy-Milner Logic (HML) for Generalized Synchronization Trees (GSTs) that we call Generalized Hennessy Milner Logic (GHML). Importantly, this logic suggests a strong relationship between (weak) bisimulation for GSTs and ordinary bisimulation for Synchronization Trees (STs). We demonstrate that this relationship can be used to define the GST analog for image-finiteness of STs. Furthermore, we demonstrate that certain maximal Hennessy-Milner classes of STs have counterparts in maximal Hennessy-Milner classes of GSTs with respect to GST weak bisimulation. We also exhibit some interesting characteristics of these maximal Hennessy-Milner classes of GSTs.

1 Introduction

In the context of discrete systems modeled as Synchronization Trees (STs), Hennessy and Milner first noticed a relationship between bisimulation and a simple modal logic, subsequently to be known as Hennessy-Milner logic (HML) [13]. In particular, they observed that HML characterizes bisimulation within the class of image-finite STs in the following sense: two image-finite STs are bisimilar if and only if they satisfy exactly the same HML formulas. Subsequent to Hennessy and Milner's original work, HML has likewise been shown to characterize bisimulation within other classes of STs (though not the class of *all* STs [16]). Indeed, any class of STs for which modal equivalence implies bisimulation is known as a *Hennessy-Milner class* (HM class), and a number of maximal ones have been exhibited [14].

Such characterizations of bisimulation have significant ramifications for the verification of system properties: if two systems belong to the same HM class, then they can be checked for bisimulation equivalence by checking HML formulas instead. In addition, if two such systems are *not* bisimilar, then an HML formula can bear witness to this lack of bisimilarity [2]. For a simple logic such as HML, this is a considerable advantage. Moreover, the existence of *maximal* HM classes is particularly important in the study of STs and process algebra, given the inherent compositionality of those objects. In particular, it is useful to know which operations preserve membership in a maximal HM class; this has been considered for some cases in [14].

Recently, the authors have proposed Generalized Synchronization Trees (GSTs) [9] as a flexible modeling framework for *non-discrete* systems such as continuous or hybrid systems. Despite their broader applicability, GSTs have many similarities to STs: elegant composition operators between GSTs are plentiful, and there are well defined notions of bisimulation (see [9]). Thus, GSTs are natural candidates for the treatment of both HML-like logics and HM classes, especially with a view to studying the composition of continuous and hybrid systems. This paper launches such a study and makes three crucial

*This work was supported by NSF grant CNS-1446665.

K. Peters and S. Tini (Eds.): Combined Workshop on Expressiveness in
Concurrency and Structural Operational Semantics (EXPRESS/SOS 2017).
EPTCS 255, 2017, pp. 35–50, doi:10.4204/EPTCS.255.3

$Sys_1 \sqsupseteq_X Sys_2$	:	*Sys_2 simulates Sys_1* (w.r.t. simulation notion X)
$Sys_1 \leftrightarrows_X Sys_2$	:	Sys_1 and Sys_2 are bisimilar (w.r.t. simulation notion X)
$s \leftrightarrows_X t$	:	*States (or worlds, nodes) s* and t are bisimilar (w.r.t. simulation notion X)
$Sys_1 \approx_Y Sys_2$	:	Sys_1 and Sys_2 satisfy the same formulas of logic Y
$s \approx_Y t$	:	*States (or worlds, nodes) s* and t satisfy the same formulas of logic Y

Table 1: Notation for simulation, bisimulation and modal equivalence.

contributions on the topic of modal logic for non-discrete systems: first, we define a novel generalization of HML that has semantic parity with trajectories in GSTs; second, we use this logic to define a notion of image finiteness for GSTs; and third, we exhibit a partial characterization of maximal HM classes in the context of our generalized HML. The third contribution is particularly novel, since there seem to be no results at all about *maximal* HM classes for generic hybrid system models, much less results in a framework as flexible and compositional as GSTs. Since GSTs can exhibit infinite – and even continuous – non-determinism, they offer a particularly rich setting in which to explore the structure of HM classes.

There are other results in the hybrid systems literature that relate bisimulation to modal logics (see e.g. [4]), but these results typically focus on establishing that *bisimulation preserves the satisfaction of formulas* from some modal logic. Almost none consider the problem of specifying when modal equivalence implies bisimilarity, i.e. the identification of Hennessy-Milner classes. The few papers that do consider the problem of identifying HM classes seem to be concerned with probabilistic systems: see [5, 6, 7, 17, 8, 3, 1] for example. However, in most of these papers, the following quote is emblematic of the source of these results: "... the probabilistic systems we are considering, without explicit nondeterminism, resemble deterministic systems quite closely, rather than nondeterministic systems" [5]. In other words, these papers typically end up with something like an image-finite assumption, and that forms the basis of their HM classes. On the other hand, the papers that do not have an image-finiteness assumption ([17, 8]) always consider more complicated logics than HML, and do not address questions of maximal HM classes.

2 Background

This section describes several foundational results that will be used subsequently. The first subsection contains background material on GSTs, including a review of the relevant notions of bisimulation for the same. The second subsection contains background material on HML and Hennessy and Milner's result for image-finite processes. The third subsection describes some maximal Hennessy-Milner classes over Kripke structures [14] and some necessary preliminaries on the canonical model [10]. Table 1 describes some common notation that will be used throughout this section and the rest of the paper.

2.1 Generalized Synchronization Trees

This section summarizes the theory of GSTs as presented in [9]; the interested reader is referred to that reference for more details.

GSTs extend Milner's Synchronization Trees (STs) via a generalized notion of tree. In particular, the essential element of a GST is the *tree partial order*, a well known structure in the mathematical literature (see [15], for example).

Definition 1 (Tree [15, 9]). *A* tree *is a triple $\langle P, \preceq, p_0 \rangle$, where $\langle P, \preceq \rangle$ is a partial order, $p_0 \in P$, and the following hold.*

1. *$p_0 \preceq p$ for all $p \in P$ (p_0 is called the* root *of the tree)*
2. *For any $p \in P$ the set $[p_0, p] \triangleq \{p' \in P : p_0 \preceq p' \preceq p\}$ is totally ordered by $\preceq$.*

In this definition of a tree, there is no inherent notion of edge, or "discrete" transition, so unlike STs external interactivity cannot be captured by labeling edges; such action labels need a different encoding. The definition of GSTs below suggests just such a scheme.

Definition 2 (Generalized Synchronization Tree [9]). *Let L be a set of labels. Then a **Generalized Synchronization Tree (GST)** is a tuple $\langle P, \preceq, p_0, \mathscr{L} \rangle$, where:*

1. *$\langle P, \preceq, p_0 \rangle$ is a tree in the sense of Definition 1; and*
2. *$\mathscr{L} \in P \backslash \{p_0\} \to L$ is the labeling function (may be partial).*

Intuitively, labels in a GST are affixed to nodes in the tree. If the tree is discrete, it can be converted into an ST by moving the node labels onto the incoming edge from the nodes parent (note that roots are not labeled in GSTs, so this operation is well-defined).

Another consequence of a lack of discrete transitions is that bisimulation must be defined differently than for STs. Specifically, bisimulation between GSTs is defined in *trajectories*, which are totally ordered sets of nodes in a GST that play roughly the same role as transitions (or sequences of transitions) in discrete bisimulation.

Definition 3 (Trajectory [9]). *Let $\langle P, \preceq, p_0, \mathscr{L} \rangle$ be a GST, and let $p \in P$. Then a **trajectory** from p is either:*

1. *the set $(p, p'] \triangleq \{p'' \in P : p \prec p'' \preceq p'\}$ for some $p' \succ p$, or*
2. *a (set-theoretic) maximal linear subset $P' \subseteq P$ with the property that for all $p' \in P'$, $p' \succ p$.*

*Trajectories of the first type are called **bounded**, while those of the second type are called **(potentially) unbounded**.*

To account for the labels on the nodes of a trajectory, we define a notion of order equivalence to parallel the notion of identically labeled transitions in a ST:

Definition 4 (Order Equivalence [9]). *Let $\langle P, \preceq_P, p_0, \mathscr{L}_P \rangle$ and $\langle Q, \preceq_Q, q_0, \mathscr{L}_Q \rangle$ be GSTs, let T_p, T_q be trajectories from $p \in P$ and $q \in Q$ respectively. Then T_p and T_q are **order-equivalent** if there exists a bijection $\lambda \in T_P \to T_Q$ such that:*

1. *$p_1 \preceq_P p_2$ if and only if $\lambda(p_1) \preceq_Q \lambda(p_2)$ for all $p_1, p_2 \in T_P$, and*
2. *$\mathscr{L}_P(p) = \mathscr{L}_Q(\lambda(p))$ for all $p \in T_P$*

When λ has this property, we say that λ is an order equivalence *from T_P to T_Q.*

We now recall the two notions of simulation from [9]; corresponding notions of bisimulation can be defined in the obvious way [9].

Definition 5 (Weak Simulation for GSTs[1] [9]). *Let $G_1 = \langle P, \preceq_P, p_0, \mathscr{L}_P \rangle$ and $G_2 = \langle Q, \preceq_Q, q_0, \mathscr{L}_Q \rangle$ be GSTs. Then $R \subseteq P \times Q$ is a **weak simulation from G_1 to G_2** if, whenever $\langle p, q \rangle \in R$ and $p' \succeq p$, then there is a $q' \succeq q$ such that:*

1. *$\langle p', q' \rangle \in R$, and*

[1] "Weak" is used here only as a relative term; it does not refer to the inclusion of τ transitions.

2. *Trajectories $(p,p']$ and $(q,q']$ are order-equivalent.*

We say $G_1 \trianglerighteq_w G_2$ if there is a weak simulation R from G_1 to G_2 with $\langle p_0,q_0\rangle \in R$.

Definition 6 (Strong Simulation for GSTs [9])**.** *Let $G_1 = \langle P, \preceq_P, p_0, \mathscr{L}_P\rangle$ and $G_2 = \langle Q, \preceq_Q, q_0, \mathscr{L}_Q\rangle$ be GSTs. Then $R \subseteq P \times Q$ is a **strong simulation from** G_1 **to** G_2 if, whenever $\langle p,q\rangle \in R$ and T_p is a trajectory from p, there is a trajectory T_q from q and bijection $\lambda \in T_p \to T_q$ such that:*

1. *λ is an order equivalence from T_p to T_q, and*
2. *$\langle p', \lambda(p')\rangle \in R$ for all $p' \in T_p$.*

We write $G_1 \trianglerighteq_s G_2$ if there is a strong simulation R from G_1 to G_2 with $\langle p_0,q_0\rangle \in R$.

2.2 Hennessy-Milner Logic and a Hennessy-Milner Class

2.2.1 Hennessy-Milner Logic

Hennessy-Milner Logic is defined as follows; note the lack of atomic propositions.

Definition 7 (Hennessy-Milner Logic (HML) [13])**.** *Given a set of labels, L, **Hennessy-Milner Logic (HML)** is the set of formulas $\Phi_{HML}(L)$ specified as follows, where $\ell \in L$:*

$$\varphi := \top \quad | \quad \neg\varphi \quad | \quad \varphi_1 \wedge \varphi_2 \quad | \quad \langle\ell\rangle\varphi. \tag{1}$$

In [13], the semantics of this logic are defined in following familiar way over STs.

Definition 8 (Semantics of HML [13])**.** *A satisfaction relation of HML for a set of STs $\mathscr{P}$ is a set $\models \subseteq \mathscr{P} \times \Phi_{HML}(L)$ such that:*

1. *$p \models \top$ for all $p \in \mathscr{P}$;*
2. *$p \models \neg\varphi$ if and only if $p \not\models \varphi$;*
3. *$p \models \varphi_1 \wedge \varphi_2$ if and only if $p \models \varphi_1$ and $p \models \varphi_2$; and*
4. *$p \models \langle l\rangle\varphi$ if and only if there exists a p' such that $p \xrightarrow{l} p'$ and $p' \models \varphi$.*

Here $p \xrightarrow{l} p'$ means p' is a subtree of p whose root is a child of the root of p, with the edge connecting the roots labeled by l.

Remark 1. *We will freely avail ourselves of usual derived operators such as $\bot$, $\vee$, $\to$ and $[\ell]$.*

2.2.2 Bisimulation and a Hennessy-Milner Class

Hennessy and Milner noticed that STs satisfying the same HML formulas need not be bisimilar; i.e. $p \approx_{\text{HML}} q \not\Rightarrow p \leftrightarrows q$ in general [13]. Nevertheless, they exhibited a class of STs for which HML modal equivalence *does* imply bisimilarity: that is the class of *image-finite STs*.

Definition 9 (Image-Finite Process [13])**.** *A ST $p \in \mathscr{P}$ is said to be image-finite if for each subtree q of p (including p itself) and each label $\ell \in \mathscr{L}$, the set $\{q' : q \xrightarrow{\ell} q'\}$ is finite.*

Hennessy and Milner proved the following theorem.

Theorem 1 (Image-Finite Hennessy-Milner Theorem [13])**.** *Let p and q be any two image-finite STs. Then*

$$p \leftrightarrows q \iff p \approx_{HML} q. \tag{2}$$

Remark 2. *Any theorem with a conclusion of the form* (2) *is called a **Hennessy-Milner Theorem**. Likewise, any class of STs (or systems, Kripke structures, etc.) for which a Hennessy-Milner Theorem can be exhibited is called a **Hennessy-Milner class**.*

2.3 The Canonical Model and (Maximal) Hennessy-Milner Classes

Since Hennessy and Milner's work [13], other *Hennessy-Milner classes* have been exhibited. In particular, Visser, via Hollenberg [14], has generalized the idea of a Hennessy-Milner class to Kripke structures, and exhibited certain *maximal* Hennessy-Milner classes of Kripke structures. This section describes the characterization of these classes.

As a prelude, we introduce the following familiar definitions of modal logic, Kripke structure and bisimulation between Kripke structures.

Definition 10 (Modal Logic [10])**.** *By a **modal logic**, we mean formulas constructed as in Definition 7 but with the addition of propositional variables (atomic propositions). The set of formulas with a set of propositional variables Θ and a set of labels L is denoted by $\Phi_\Theta(L)$.*

Definition 11 (Kripke structure [10, 14])**.** *A Kripke structure of a set of labels L and a set of propositional variables Θ is a tuple $\mathbf{S} = (S, \{R_\ell \subseteq S \times S : \ell \in L\}, V)$ where*

- *S is the set of states (or worlds);*
- *for each $\ell \in L$, $R_\ell \subseteq S \times S$ is a transition relation for label ℓ; and*
- *$V : \Theta \to 2^S$ is a function that maps propositional variables to sets of states.*

Definition 12 (Satisfaction relation for a Kripke structure)**.** *Given a Kripke structure $\mathbf{S} = (S, \{R_\ell \subseteq S \times S : \ell \in L\}, V)$, a satisfaction relation $\models \subseteq S \times \Phi_\Theta(L)$ is defined as in Definition 7 with the addition that for any $\theta \in \Theta$, $s \models \theta$ if and only if $s \in V(\theta)$.*

Definition 13 (Bisimulation between Kripke structures)**.** *Given two Kripke structures $\mathbf{S}$ and $\mathbf{T}$, we say that $s \in S$ and $t \in T$ are bisimilar or $s \leftrightarrows_K t$ if there is a bisimulation relation $\sim$ such that $s \sim t$, and for all $s' \in S, t' \in T$, and $\theta \in \Theta$, $s' \sim t'$ implies $s' \models \theta \Leftrightarrow t' \models \theta$. Bisimulation between Kripke structures is defined in the obvious way.*

2.3.1 The Canonical Model for a Modal Logic

The so-called *canonical model* – called the *Henkin model* in [14] – is a special Kripke structure that is one of the most important tools in the study of (normal) modal logic(s). In the canonical model states (or worlds) are defined in terms of the modal formulas that they satisfy. In our context it has an important connection to the maximal Hennessy-Milner classes we will consider; see Subsection 2.3.2. This subsection is meant to be a summary of the relevant material in [10]; the full details can be found therein.

In order to define the canonical model, we must first establish certain consistency criteria that we will enforce on any "reasonable" set of formulas. Any such "reasonable" set of formulas will (somewhat confusingly) be called a *logic*.

Definition 14 (Logic [10])**.** *A set of modal formulas $\Lambda \subseteq \Phi_\Theta(L)$ is called a **logic** if it satisfies:*

1. *Λ contains all of the **tautologies**: that is all of the formulas which are true irrespective of how we assign truth values to modal sub-formulas and propositional variables; and*
2. *Λ is closed under modus ponens: if $\varphi_1 \in \Lambda$ and $\varphi_1 \to \varphi_2 \in \Lambda$[2], then $\varphi_2 \in \Lambda$.*

Definition 15 (Λ-Consistent Set of Modal Formulas [10])**.** *Given a logic $\Lambda \subseteq \Phi_\Theta(L)$, a set of modal formulas $\Gamma \subseteq \Phi_\Theta(L)$ is said to be **Λ-consistent** if there is no formula of the form $\varphi_0 \to (\varphi_1 \to (\cdots \to (\varphi_n \to \bot)\ldots)$ in Λ, where $\varphi_0, \ldots, \varphi_n \in \Gamma$.*

[2] In terms of HML, $\varphi_1 \to \varphi_2$ is shorthand for $\neg(\varphi_1 \wedge \neg\varphi_2)$.

Definition 16 (Λ-maximal set of formulas [10])**.** *Given a logic Λ, a set of formulas $\Gamma \subseteq \Phi_\Theta(L)$ is said to be Λ-**maximal** if it satisfies the following two properties:*

1. *Γ is Λ-consistent; and*
2. *for all $\varphi \in \Phi_\Theta(L)$, either $\varphi \in \Gamma$ or $\neg\varphi \in \Gamma$.*

Lemma 1 (Lindenbaum's Lemma [10])**.** *Given a logic Λ and a Λ-consistent set of formulas Γ, there exists a Λ-maximal set of formulas $\Gamma_0 \subseteq \Phi_\Theta(L)$ such that $\Gamma \subseteq \Gamma_0$.*

Corollary 1 (The set of maximal Λ-consistent sets of formulas is non-empty [10])**.** *Given a logic Λ, let S^Λ denote the set of Λ-maximal sets of formulas. Then S^Λ is non-empty.*

For our purposes, Corollary 1 tells us that the set of maximally consistent sets of formulas is non-empty, and hence, can be used as the set of worlds for the canonical model. The next step in the construction of the canonical model is to define the *transitions* between these states; these must ensure that a state – a Λ-maximal set of formulas – satisfies a modal formula if an only if that formula is an element of the set. This desirable property – called the "Henkin property" in [14] – is the essence of the value of the canonical model. It turns out that some additional conditions must be imposed on a logic Λ before transitions can be defined between Λ-maximal sets of formulas in such a way that the Henkin property holds.

Definition 17 (Normal Logic [10])**.** *A logic Λ is **normal** if it satisfies:*

1. *for all $\varphi_1, \varphi_2 \in \Phi_\Theta(L)$ and $\ell \in L$, the formula $[\ell](\varphi_1 \to \varphi_2) \to ([\ell]\varphi_1 \to [\ell]\varphi_2)$ is in Λ;*[3]
2. *for all $\varphi \in \Lambda$ and $\ell \in L$, $[\ell]\varphi \in \Lambda$.*

With this definition in hand, we can define the *canonical model*.

Definition 18 (Canonical (Henkin) Model [10, 14])**.** *Let $\Lambda \subseteq \Phi_\Theta(L)$ be a normal logic. Then the **canonical model** is the Kripke structure $\mathbf{C}^\Lambda = (S^\Lambda, \{R^\Lambda_\ell : \ell \in L\}, V^\Lambda)$ defined as follows:*

- *S^Λ is the set of states (worlds);*
- *for each $\ell \in L$, the transition relation $R^\Lambda_\ell \subseteq S^\Lambda \times S^\Lambda$ is defined such that $sR_\ell t$ if and only if $\varphi \in t$ implies that $\langle\ell\rangle\varphi \in s$.*
- *the valuation $V^\Lambda : \Theta \to S^\Lambda$ is defined such that $V(p) = \{s \in S^\Lambda : p \in s\}$.*

Theorem 2 (The Canonical Model Satisfies the Henkin Property [10, 14])**.** *For any state s in the canonical model $\mathbf{C}^\Lambda$ and any formula $\varphi \in \Phi_\Theta(L)$: $s \models \varphi \iff \varphi \in s$ (the aforementioned **Henkin property**).*

2.3.2 Hennessy-Milner Classes for Kripke Structures

It is important to note that in Hennessy and Milner's definition of image-finite STs, all subtrees must be image-finite: in other words, the set of image-finite STs is *closed under subtrees*. Thus, one could think about generalizing Theorem 1 by examining when modal equivalence implies bisimulation for a larger class of STs that is closed under subtrees. The following definition captures that spirit in the context of Kripke structures, but it does so without insisting on image-finiteness.

Definition 19 (Visser/Hollenberg Hennessy-Milner Property [14])**.** *Let $\mathfrak{H}$ be a set of Kripke structures. $\mathfrak{H}$ is said to satisfy the **Visser/Hollenberg Hennessy-Milner Property (VHHM property) with respect to** $\Phi_\Theta(L)$ if for any two Kripke structures $\mathbf{S}, \mathbf{T} \in \mathfrak{H}$ and **any two states** $s' \in S$ and $t' \in T$*

$$s' \leftrightarrows t' \quad \Leftrightarrow \quad s' \approx_{\Phi_\Theta(L)} t'. \tag{3}$$

[3] Recall that $[\ell]\varphi = \neg\langle\ell\rangle\neg\varphi$.

Remark 3. *We use the terminology VHHM property to distinguish this property from another definition of* ***Hennessy-Milner Property*** *in the literature, which considers only modal equivalence and bisimulation between initial states (the points of pointed Kripke structures). For example, this definition is used in [12]. However, we note that a number of other sources use what we call the VHHM property; see [11] for example.*

Definition 20 (Visser-Hollenberg Hennessy-Milner Class)**.** *We say that any set of Kripke structures that satisfies the Visser/Hollenberg Hennessy-Milner Property is a* ***Visser-Hollenberg Hennessy-Milner class (VHHM class)****.*

Definition 19 seems innocuous, but in fact the VHHM property is a nontrivial strengthening of the HM property described in Remark 3. To the best of our knowledge, there are no results in the literature that compare VHHM classes with this alternate definition of HM classes. We will revisit this in Section 5.2 where we exhibit a Kripke structure that fails to be a member of any VHHM class because it fails to satisfy the conditions of Definition 19.

Importantly, though, there is an elegant characterization of *maximal* VHHM classes due to Visser and reported in [14]. We first define the notion of a "Henkin-like" model [14].

Definition 21 (Henkin-like model [14])**.** *Let* $\mathbf{C}^K$ *be the canonical model associated with the smallest normal logic K. Then a* ***Henkin-like*** *model is any Kripke structure* $\mathbf{HC}^K = (S^K, \{R_\ell^{\mathbf{H}K} \subseteq R_\ell^K : \ell \in L\}, V^K)$ *that satisfies the Henkin property (see Theorem 2 and the discussion preceding it).*

Thus, a Henkin-like model is simply the canonical model with transitions removed in such a way that a state satisfies a formula if and only that formula is an element of the state (recall that the states in $\mathbf{C}^K$ are sets of formulas). Henkin-like models form the basis for maximal VHHM classes in the following sense.

Theorem 3 (Maximal VHHM Classes [14])**.** *Let* $\mathbf{HC}^K$ *be any Henkin-like model, and let* $\mathsf{S}(\mathbf{HC}^K)$ *be the set of generated sub-models of* $\mathbf{HC}^K$*. Then*

1. *The set of all Kripke structures that are bisimilar to a model in* $\mathsf{S}(\mathbf{HC}^K)$ *is a maximal VHHM class; that is it is maximal in a set-theoretic sense. We denote such a class by* $\mathsf{BS}(\mathbf{HC}^K)$ *.*
2. *Let* $\mathfrak{H}$ *be any set of Kripke structures that satisfies the VHHM property. Then* $\mathfrak{H} \subseteq \mathsf{BS}(\mathbf{HC}^K)$ *for at least one Henkin-like model* $\mathbf{HC}^K$*.*

The basic idea behind Theorem 3 is this: a set of models $\mathsf{BS}(\mathbf{HC}^K)$ is necessarily a VHHM class because modal equivalence is a bisimulation relation over a single Henkin-like model (each maximal set of formulas is satisfied only by its own unique state in the model). Thus, Henkin-like models effectively "canonicalize" different VHHM classes because a given Henkin-like model associates a particular transition structure with each and every (maximal) set of formulas that can be satisfied in any Kripke structure (K is sound and complete with respect to Kripke structures).

Maximal VHHM classes are related to the VHHM class of image finite models in the following way.

Theorem 4 (Image Finite Kripke structures [14])**.** *Each maximal VHHM class of Theorem 3 contains every Kripke structure that is bisimilar to an image finite Kripke structure. Hence, each maximal VHHM class contains all image finite Kripke structures, and the class of image finite Kripke structures is itself a VHHM class.*

3 Generalized Hennessy-Milner Logic

In this section, our aim is to define a logic akin to HML but with GSTs as the intended models. We proceed by first defining the syntax and then the semantics of our logic.

3.1 HML for GSTs: Syntax

Our generalization of HML will be mostly recognizable, but the $\langle\,\rangle$ modality requires some significant modifications. In particular, recall that in weak bisimulation for GSTs, transitions are replaced by *trajectories* (see Definition 3) and labels by functions over trajectories. To capture this notion, we generalize the way we *label* the $\langle\,\rangle$ modality.

Definition 22 (Domain of modalities)**.** *A domain of modalities is a totally ordered set, $(\mathscr{I}, \preceq_{\mathscr{I}})$, together with a set of labels L.*

Intuitively, a domain of modalities will be used to define the trajectory-like structures appearing in our modalities. However, we eventually need such a domain of modalities to satisfy some additional properties to ensure certain formulas exist. Hence, we provide the following definitions.

Definition 23 (Spanned by an interval)**.** *Given a totally ordered set $\mathscr{I}$, we say a subset $I \subseteq \mathscr{I}$ is **spanned by an interval**, if there exists an interval $[i_0, i_1] = \{i \in \mathscr{I} : i_0 \preceq_{\mathscr{I}} i \preceq_{\mathscr{I}} i_1\}$ such that $I \subseteq [i_0, i_1]$ and $\{i_0, i_1\} \subset I$; this is equivalent to saying that I contains its least upper bound (LUB) and greatest lower bound (GLB). We say that i_0 and i_1 are the **left and right endpoints of** I, respectively, and they will be denoted by $\overleftarrow{I}$ and $\overrightarrow{I}$, respectively.*

Definition 24 (Left-open subset)**.** *A subset I of a totally ordered set $\mathscr{I}$ is **left open** if there exists a set $I' \subseteq \mathscr{I}$ spanned by an interval such that $I = I' \backslash \{\overleftarrow{I'}\}$*

Definition 25 (Closed under left-open concatenation)**.** *We say that a totally ordered set $\mathscr{I}$ is **closed under left-open concatenation** if for any two left-open subsets $I_1, I_2 \subseteq \mathscr{I}$, there exists another left-open set I_3 such that there is an order preserving bijection from I_3 to the totally ordered set $I_1; I_2 = (\{1\} \times I_1) \cup (\{2\} \times I_2)$ under the lexicographic ordering. A totally ordered set $\mathscr{I}$ that is closed under left-open concatenation will be denoted $\bar{\mathscr{I}}$.*

Example 1. *Any totally ordered set that can be embedded in an order-preserving additive group structure is closed under left-open concatenation. $\mathbb{N}$, $\mathbb{R}$ and $\mathbb{R} \times \mathbb{N}$ are examples.*

Remark 4. *Henceforth, we will work exclusively with total orders that are closed under left-open concatenation when we construct a domain of modalities.*

Definition 26 (Modal execution)**.** *Let $(\bar{\mathscr{I}}, L)$ be a domain of modalities. A **modal execution** is a map from a left-open subset of $\bar{\mathscr{I}}$ to the set of labels, L. The set of modal executions over $(\bar{\mathscr{I}}, L)$ will be denoted $\mathscr{M}(\bar{\mathscr{I}}, L)$.*

The notion of a modal execution is almost usable as a label for our generalized diamond modalities, but it is too tied to the specific *domain* of the function in question. This will prove cumbersome in the future, so we restrict ourselves to equivalence classes of such modalities.

Definition 27 (Order Equivalent Modal Executions)**.** *Let $E_1 : I_1 \to L$ and $E_2 : I_2 \to L$ be two modal executions from a domain of modalities $(\bar{\mathscr{I}}, L)$. We say that E_1 is **order equivalent** to E_2 if there exists an order preserving bijection $\lambda : I_1 \to I_2$ such that $E_1(i) = E_2(\lambda(i))$ for all $i \in I_1$. If E_1 is order equivalent to E_2, then we write $E_1 \overset{o.e}{\sim} E_2$. This definition parallels Definition 4 for GST trajectories.*

Theorem 5 (Order Equivalence is an equivalence relation)**.** *$\overset{o.e.}{\sim}$ is an equivalence relation between modal executions. We denote by the equivalence class $\{E' \in \mathscr{M}(\bar{\mathscr{I}}, L) : E \overset{o.e.}{\sim} E'\}$ by $|E|$, and the set of such equivalence classes by $|\mathscr{M}(\bar{\mathscr{I}}, L)|$.*

Definition 28 (Set of Generalized HML (GHML) formulas)**.** *Given a domain of modalities $(\bar{\mathscr{I}}, L)$, the set of* ***Generalized HML (GHML)*** *formulas is the set of formulas, $\Phi_{GHML}(\bar{\mathscr{I}}, L)$, inductively defined according to the following rules:*

$$\varphi := \top \quad | \quad \neg\varphi \quad | \quad \varphi_1 \wedge \varphi_2 \quad | \quad \langle\langle |E| \rangle\rangle \varphi \tag{4}$$

where $|E|$ is an equivalence class of modal executions over the domain of modalities $(\bar{\mathscr{I}}, L)$.

The formal semantics of this logic will be presented in next subsection within Definition 31.

We have chosen to define our logic without propositional variables in order to mirror Hennessy and Milner's original work. However, in Section 5 we will consider a modal logic with a syntax based on Definition 28, and so we describe here such a modal logic.

Definition 29 (GHML Modal Logic)**.** *A GHML modal logic is a modal logic with all of the connectives from Definition 28 plus propositional variables. If Θ is the set of propositional variables, then we denote the set of these formulas by $\Phi_{GHML\text{-}\Theta}(\bar{\mathscr{I}}, L)$.*

A number of the proof theoretic results from Section 2.3.1 apply equally well to a GHML modal logic: the definition of a logic (Definition 14), the definition of Λ-consistency and the definition of Λ-maximality all apply directly to a GHML modal logic. On the other hand, Lindenbaum's lemma (Lemma 1) requires a different proof because of the multiplicity of modalities. Nevertheless, it is still true, as the following theorem asserts.

Theorem 6 (Λ-maximal sets of GHML formulas)**.** *Let $\Lambda \subseteq \Phi_{GHML\text{-}\Theta}(\bar{\mathscr{I}}, L)$ be a logic, and let $\Gamma \subseteq \Phi_{GHML\text{-}\Theta}(\bar{\mathscr{I}}, L)$ be a Λ-consistent set of formulas. Then there exists a Λ-maximal set $\Gamma_0 \subseteq \Phi_{GHML\text{-}\Theta}(\bar{\mathscr{I}}, L)$ such that $\Gamma \subseteq \Gamma_0$.*

Proof. Because the collection of GHML modal logic formulas is a set, this is a straightforward application of Zorn's lemma. □

3.2 HML for GSTs: Semantics

We define the semantics of GHML for a GST model G in terms of the set of the sub-GSTs of G; because each GST is itself defined in terms of sets, we may soundly define the following notion of a sub-GST rooted at a node.

Definition 30 (Sub-GST rooted at a node)**.** *Let $G = (P, \preceq, p_0, \mathscr{L})$ be a GST. We let $G|_p$ denote the sub-GST of G rooted at p, i.e. $G|_p \triangleq (\{p' \in P | p' \succeq p\}, \preceq, p, \mathscr{L})$.*

Now we can formally define the semantics of the generalized HML formulas defined above.

Definition 31 (Satisfaction relation over GHML formulas)**.** *Let $G = (P, \preceq, p_0, \mathscr{L})$ be a GST, and let $\mathscr{G}_{sub} := \{G|_p : p \in P\}$. A satisfaction relation, $\models$, is a relation $\models \subseteq \mathscr{G}_{sub} \times \Phi_{GHML}(\bar{\mathscr{I}}, L)$ that is defined inductively over GHML formulas. Satisfaction of the formula $\langle\langle |E| \rangle\rangle \varphi$ is defined in the following way: $G \models \langle\langle |E| \rangle\rangle \varphi$ if and only if there exists an interval $(p_0, p]$; a left-open set $I \subset \bar{\mathscr{I}}$; and an order preserving bijection $\lambda : I \to (p_0, p]$ such that*

1. *$\mathscr{L} \circ \lambda \in |E|$*
2. *$G|_p \models \varphi$.*

The satisfaction relation is defined for other formulas in the usual way.

Intuitively, a GST satisfies the formula $\langle\langle |E| \rangle\rangle \top$ when it has a trajectory emanating from its root that is order equivalent to every $E \in |E|$ (recall that all elements of $|E|$ are order equivalent to each other). Importantly, this logic also yields formulas that are analogous to HML formulas on discrete GSTs when there are at least two points in $\bar{\mathscr{I}}$. In particular, if $i_0 \preceq i_1$, then $\{i_0, i_1\}$ is spanned by the interval $[i_0, i_1]$, and the singleton point $\{i_1\}$ is a left-open set. Thus, $\mathscr{M}(\bar{\mathscr{I}}, L)$ contains modal executions that are order-equivalent to discrete transitions in a GST. Of course, discrete transitions are the essence of the semantics for the labeled modalities in HML.

4 A First Hennessy-Milner Theorem: "Image-finite" GSTs

In this section, our objective is to define something like a class of image-finite GSTs with the ultimate intention of defining a Hennessy-Milner class of GSTs. We introduce this section with an example to show that the most straightforward definition of image-finiteness is too exclusive to be of much interest.

Example 2. *Consider the following GST defined on the unit interval* $[0,1] \subset \mathbb{R}$*:* $G_{[0,1]} := ([0,1], \leq_{\mathbb{R}}, 0, (0,1] \to \{\alpha\})$.

The point of Example 2 is that $G_{[0,1]}$ has uncountably many nodes that are accessible from the root, 0, with a single trajectory: that is for any $x, y \in (0,1]$ there is an order preserving bijection between $(0,x]$ and $(0,y]$. Since these trajectories' nodes are labeled by a single label, α, they are thus order equivalent in the sense of Definition 4. Nevertheless, this GST appears to be about as simple as one could wish for in terms of nondeterminism: there is essentially no branching behavior at all.

4.1 GSTs as Discrete Structures

The discussion following Example 2 suggests a way of looking at GSTs that will be profitable, especially when it comes to examining GHML formulas and constructing Hennessy-Milner classes. In particular, we use equivalence classes of modal executions to label discrete transitions on a Kripke structure; we show that such a construction captures the relevant structure of a given class of GSTs with respect to bisimulation and GHML satisfaction.

Definition 32 (Captured by a Domain of modalities)**.** *Let $\mathscr{U}$ be a set of GSTs. We say that $\mathscr{U}$ is* ***captured*** *by a domain of modalities $(\bar{\mathscr{I}}, L)$ if every trajectory from every GST in $\mathscr{U}$ is order equivalent to some modal execution over $(\bar{\mathscr{I}}, L)$.*

Definition 33 (Surrogate Kripke Structure)**.** *Let $\mathscr{U}$ be a set of GSTs that is captured by a domain of modalities $(\bar{\mathscr{I}}, L)$. For any GST $G = (P, \preceq_P, p_0, \mathscr{L})$ in $\mathscr{U}$, we define a* ***surrogate Kripke structure****, $\mathbf{G} = (P, \{R^G_{|E|} \subseteq P \times P : |E| \in \mathscr{M}(\bar{\mathscr{I}}, L)\}, V)$, as follows:*

- *the set of states is P; and*
- *$p_1 \overset{|E|}{\to} p_2$ – i.e. $p_1 R^G_{|E|} p_2$ – if and only if $p_1 \preceq_P p_2$ and $(p_1, p_2]$ is order equivalent to an element of $|E|$; and*
- *$V : \Theta \to \{P\}$ indicates all propositional variables are true in all states in P.*

Remark 5. *We will not consider valuations in this section, but they will be used in the next section. Thus, for the purposes of this section, we may regard surrogate Kripke structures as labeled transition systems.*

Example 3 (Surrogate Kripke Structure for $G_{[0,1]}$)**.** *If we let $\bar{\mathscr{I}} = (\mathbb{R}, \leq_{\mathbb{R}})$ and $L = \{\alpha\}$, then Figure 1 shows some of the transitions that appear in the surrogate Kripke structure for the GST $G_{[0,1]}$ from Example 2.*

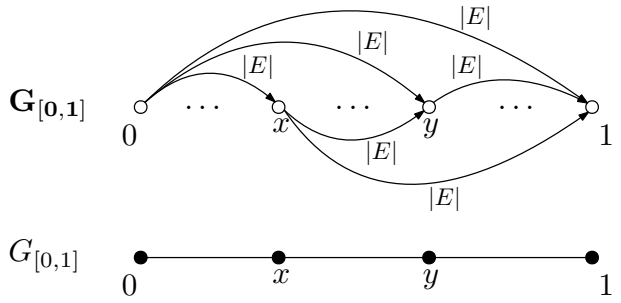


Figure 1: GST and surrogate Kripke structure from Example 3; we define $E : (0,1] \to \{\alpha\}$.

The idea of a surrogate Kripke structure seems simple enough, but its importance is indicated by the following two theorems: one relates GHML formulas to HML formulas, and the other relates ordinary bisimulation to weak bisimulation for GSTs.

Theorem 7 (Relating GHML formulas on G to HML formulas on $\mathbf{G}$)**.** *Let $(\bar{\mathscr{I}}, L)$ be a domain of modalities, and let $\mathscr{U}$ be a set of GSTs captured by $(\bar{\mathscr{I}}, L)$. Furthermore, consider HML over the set of labels given by $|\mathscr{M}(\bar{\mathscr{I}}, L)|$. Then for every $G = (P, \preceq_P, p_0, \mathscr{L}) \in \mathscr{U}$,*

1. *for all $\varphi \in \Phi_{GHML}(\bar{\mathscr{I}}, L)$, $G \models \varphi \;\Rightarrow\; p_0 \models \varphi_{\langle\rangle}$ and*
2. *for all $\phi \in \Phi_{HML}(|\mathscr{M}(\bar{\mathscr{I}}, L)|)$, $p_0 \models \phi \;\Rightarrow\; G \models \phi_{\langle\langle\rangle\rangle}$.*

The notation $\varphi_{\langle\rangle}$ indicates that the GHML formula φ is converted to an HML formula by replacing each $\langle\langle |E| \rangle\rangle$ modality with the corresponding HML modality $\langle |E| \rangle$. $\phi_{\langle\langle\rangle\rangle}$ indicates an analogous conversion from an HML formula to a GHML formula.

Proof. This is a straightforward proof by induction on formula structure (the base case is $\top$, which has an identical meaning in HML and GHML). The ability to match HML modalities to GHML modalities (and conversely) is assured by the way we have constructed the surrogate Kripke structure, and in particular, the fact that we have labeled trajectories by *equivalence classes* of modal executions. □

Theorem 8 (Weak bisimulation between GSTs and bisimulation between surrogates)**.** *Let $\mathscr{U}$ and $(\bar{\mathscr{I}}, L)$ be as in Theorem 7. Furthermore, let $G_1 = (P, \preceq_P, p_0, \mathscr{L}_P)$ and $G_2 = (Q, \preceq_Q, q_0, \mathscr{L}_Q)$ be two GSTs in $\mathscr{U}$. Then*

$$G_1 \leftrightarrows_w G_2 \quad \Longleftrightarrow \quad p_0 \leftrightarrows q_0. \tag{5}$$

where the bisimulation $p_0 \leftrightarrows q_0$ is taken in the context of the surrogate Kripke structures $\mathbf{G_1}$ and $\mathbf{G_2}$.

Proof. This theorem, like Theorem 7, is a consequence of the way that we defined the surrogate Kripke structure: in particular, any weak bisimulation relation between G_1 and G_2 is a bisimulation relation between $\mathbf{G_1}$ and $\mathbf{G_2}$ and conversely. □

Theorems 7 and 8 together reinforce that weak bisimulation is very much a discrete notion. In the context of GHML formulas and the construction of Hennessy-Milner classes, though, this will prove to be an advantage.

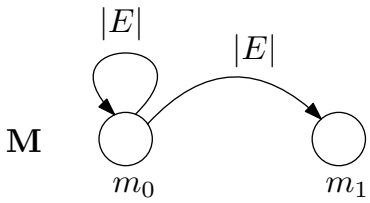


Figure 2: A Kripke structure **M** that is bisimilar to $\mathbf{G}_{[\mathbf{0},\mathbf{1}]}$; again, we define $E : (0,1] \to \{\alpha\}$.

4.2 "Image-Finite" GSTs

If we reconsider Example 2 in the context of Theorems 7 and 8, then a natural means of defining "image-finite" GSTs emerges. In particular, it is evident that the surrogate Kripke structure $\mathbf{G}_{[0,1]}$ (see Figure 1) is bisimilar to the two state Kripke structure **M** depicted in Figure 2. Of course **M** *is* image-finite, and so we have just exemplified a serviceable means by which we can define image-finiteness for GSTs.

Definition 34 (Image-finite GST)**.** *Let $\mathscr{U}$ and $(\bar{\mathscr{I}}, L)$ be as before. Then a GST $G \in \mathscr{U}$ is **image finite** if its surrogate Kripke structure is $\leftrightarrows_K$ to an image-finite Kripke structure.*

In Definition 34, we really mean bisimulation according to $\leftrightarrows_K$ of Definition 13, so that the surrogate Kripke structure ends up being bisimilar to a Kripke structure that also has a "universal" valuation. Hence, the surrogate Kripke structure and its image-finite pair can be regarded simply as labeled transition systems with the usual notion of bisimulation $\leftrightarrows$. We introduce this requirement in preparation for the treatment of maximal classes to come.

Of course because of Theorem 7 and 8, this definition of image-finiteness implies a Hennessy-Milner class of GSTs through the use of Hennessy and Milner's original theorem.

Theorem 9 (Image-finite GSTs form a Hennessy-Milner class)**.** *Let $\mathscr{U}$ and $(\bar{\mathscr{I}}, L)$ be as before. Then the set of image-finite GSTs in $\mathscr{U}$ forms a Hennessy-Milner class according to weak bisimulation. That is any two image finite GSTs from $\mathscr{U}$ are weakly bisimilar if and only if they satisfy the same GHML formulas.*

5 Maximal Hennessy-Milner Classes for GSTs

The construction of surrogate Kripke structures in Definition 33 combined with Theorems 7 and 8 suggests that the maximal VHHM classes of Section 2.3.2 have analogs as maximal HM classes of GSTs with respect to weak bisimulation. In this section we demonstrate that this is indeed the case, although the translation is not exact. We also exhibit some interesting GST-specific properties that these classes possess.

5.1 Characterizing Maximal VHHM Classes of GSTs

The essential assumption required for the proof of Theorem 3 is the VHHM property: that is a VHHM class of Kripke structures cannot contain two states that satisfy the same formulas yet are not bisimilar. Since we are interested in weak bisimulation and GHML formulas, we can straightforwardly define a VHHM property for GSTs as follows.

Definition 35 (VHHM class for GSTs)**.** *Let $\mathscr{U}$ be a set of GSTs, and let $(\bar{\mathscr{I}}, L)$ be a domain of modalities that captures $\mathscr{U}$, so that $\approx_{GHML}$ is interpreted with respect to $\Phi_{GHML-\Theta}(\bar{\mathscr{I}}, L)$. Then we say that a subset*

$\mathfrak{h} \subseteq \mathscr{U}$ ***satisfies the VHHM property for GSTs*** *if for any two sub-GSTs $G_1|_p$ and $G_2|_q$ from the set $\mathfrak{h}$ (possibly with $G_1 = G_2$),*

$$G_1|_p \leftrightarrows_w G_2|_q \iff G_1|_p \approx_{GHML} G_2|_q. \tag{6}$$

Of course this definition will help us define maximal VHHM classes of GSTs because of Theorem 7 and 8, which relate GHML formulas and weak bisimulation for GSTs to HML formulas and bisimulation for Kripke structures. Hence, we have the following theorem.

Theorem 10 (Maximal VHHM classes for GSTs are restrained by maximal VHHM classes for their surrogates)**.** *Let $\mathscr{U}$ and $(\bar{\mathscr{I}}, L)$ be as in Definition 35. If $\mathfrak{h} \subseteq \mathscr{U}$ is a VHHM class of GSTs, then the set of surrogate Kripke structures $\{\mathbf{G} : G \in \mathfrak{h}\}$ satisfies the VHHM property of Definition 20 with respect to $\Phi_{HML}(|\mathscr{M}(\bar{\mathscr{I}}, L)|)$.*

Proof. This is a direct consequence of Theorems 7 and 8. First, check that for any node p in a surrogate Kripke structure $\mathbf{G_1}$ and node q in surrogate Kripke structure $\mathbf{G_2}$, $p \approx_{\text{HML}} q$ implies $p \leftrightarrows_K q$. Because of Theorem 7, we know that $p \approx_{\text{HML}} q$ implies $G_1|_p \approx_{\text{GHML}} G_2|_q$. But $\mathfrak{h}$ is a VHMM class of GSTs, so the preceding implies that $G_1|_p \leftrightarrows_w G_2|_q$, and Theorem 8 then implies that $p \leftrightarrows_K q$ as required. The converse follows by using first Theorem 8 and then Theorem 7. □

The essential intuition here is that *any* set of GSTs that satisfies the VHHM property will yield a set of surrogate Kripke structures that satisfies the VHHM property; then by part 2 of Theorem 3, these surrogate Kripke structures will be contained in a maximal VHHM class of Kripke structures. Thus, a VHHM class of GSTs can only be enlarged so long as its surrogate Kripke structures do not escape a maximal VHHM class of Kripke structures, so *every maximal VHHM class of GSTs can be matched to at least one maximal VHHM class of Kripke structures*. This is expressed in the following corollary.

Corollary 2. *Let $\mathscr{U}$ and $(\bar{\mathscr{I}}, L)$ be as in Definition 35. If $\mathfrak{h} \subseteq \mathscr{U}$ is a VHHM class of GSTs, then there exists a Henkin-like model $\mathbf{HC}^K$ such that $\{\mathbf{G} : G \in \mathfrak{h}\} \subseteq \mathsf{BS}(\mathbf{HC}^K)$. Furthermore, if there is a set $\mathfrak{h}' \subseteq \mathscr{U}$ such that $\mathfrak{h} \subseteq \mathfrak{h}'$ and $\{\mathbf{G} : G \in \mathfrak{h}'\} \subseteq \mathsf{BS}(\mathbf{HC}^K)$, then $\mathfrak{h}'$ is a VHHM class of GSTs.*

However, we have not yet established that every maximal VHHM class of Kripke structures corresponds to a maximal VHHM class of GSTs. Indeed, the fact that Theorem 3 makes no assumptions about valuations immediately suggests that several maximal VHHM classes of the form $\mathsf{BS}(\mathbf{HC}^K)$ will correspond to the same maximal VHHM class of GSTs. As it turns out, there are other yet more profound redundancies in the Henkin-like models derived from the canonical model over the smallest normal logic, K. These differences are described in the following theorem, though it too falls short of an absolute characterization of maximal VHHM classes for GSTs.

Theorem 11 (Maximal VHHM classes for GSTs and refined Henkin-like models)**.** *Let $\mathscr{U}$ and $(\bar{\mathscr{I}}, L)$ be as in Definition 35, and let $\mathfrak{h} \subseteq \mathscr{U}$ be a VHHM class of GSTs. Furthermore, let Δ be the smallest normal logic that contains all of the following:*

- *the propositional variables Θ;*
- *$\forall |E_1|, |E_2| \in |\mathscr{M}(\bar{\mathscr{I}}, L)|$, the schema $\langle |E_1| \rangle \langle |E_2| \rangle \varphi \to \langle |E_{1;2}| \rangle \varphi$; and*
- *$\forall |E|, |E_1|, |E_2| \in |\mathscr{M}(\bar{\mathscr{I}}, L)|$ such that there is an order equivalence $\lambda : I_1; I_2 \to dom(E)$ with $E \circ \lambda(1, \cdot) \in |E_1|$ and $E \circ \lambda(2, \cdot) \in |E_2|$, the schema $\langle |E| \rangle \varphi \to \langle |E_1| \rangle \langle |E_2| \rangle \varphi$.*

Then $\{\mathbf{G} : G \in \mathfrak{h}\} \subseteq \mathsf{BS}(\mathbf{HC}^\Delta)$ for some Henkin-like model $\mathbf{HC}^\Delta$ that preserves the first-order transition-relation properties imposed on $\mathbf{C}^\Delta$ by the schemata above. Furthermore, if there is a set $\mathfrak{h}' \subseteq \mathscr{U}$ such that $\mathfrak{h} \subseteq \mathfrak{h}'$ and $\{\mathbf{G} : G \in \mathfrak{h}'\} \subseteq \mathsf{BS}(\mathbf{HC}^\Delta)$, then $\mathfrak{h}'$ is a VHHM class of GSTs.

Proof. (Theorem 11.) All of the additions to the logic Δ reflect structure in surrogate Kripke structures (including in the valuations used), and Theorem 3 remains applicable when confined to such restricted Kripke structures. □

The reader will recognize in the formulas $\langle|E_1|\rangle\langle|E_2|\rangle\varphi \to \langle|E_{1;2}|\rangle\varphi$ and $\langle|E|\rangle\varphi \to \langle|E_1|\rangle\langle|E_2|\rangle\varphi$ the schemata for something like *transitivity* and *weak density*, respectively [10]. That the surrogate Kripke structures satisfy these conditions is a reflection of the unique semantics we have specified for GHML: in particular, following one trajectory in a GST followed by another implies the existence of a third, "longer" trajectory (transitivity), and following a non-trivial trajectory implies the existence of "smaller" trajectories ending and beginning from some intermediary point (weak density). On the other hand, the additional constraint on the Henkin-like model in Theorem 11 is necessary because just satisfying the relevant schemata under one valuation is not enough to impose the first-order transition relation properties that surrogate Kripke structures possess (see [10]).

It is also worth noting that our choice of *equivalence classes* of modal executions is relevant here. Had we not chosen to label transitions in the surrogate Kripke structure with such *equivalence classes*, there would be *multiple* order-equivalent transitions between any two nodes. This would lead to additional Henkin-like models that fail to respect the semantics of weak bisimulation: i.e. among a collection of order-equivalent transitions, some could be present in the Henkin-like model while some could be absent.

Finally, it is important to note that neither Corollary 2 nor Theorem 11 imply that *every* maximal VHHM class of Kripke structures corresponds to a maximal VHHM class of GSTs in $\mathscr{U}$. For one, the set $\mathscr{U}$ may be deficient. For another, it remains as future work to show that every Henkin-like model over the canonical model for logic Δ reflects the surrogate Kripke structure of some GST.

5.2 Properties of Maximal VHHM Classes of GSTs

In this subsection we make two small remarks that identify some properties of interest with regard to maximal VHHM classes of GSTs.

First, we note that maximal VHHM classes are not so small that modal equivalence within such a class implies *strongly bisimulation* (Definition 6). That is to say there is a maximal VHHM class which contains two GSTs that satisfy the same formulas yet are not strongly bisimilar. Such a situation is illustrated in the following example.

Example 4. *Consider the domain of modalities given by the set $\bar{\mathscr{I}} = (\mathbb{R}, \leq_{\mathbb{R}})$ and the set $L = \{\alpha, \beta\}$. Furthermore, for a subset A of $[0,1] \subset \mathbb{R}$, define the GST G_A as $G_A = ([0,1] \cup (\{1\} \times A), \preceq_A, 0, \mathscr{L}_A)$ where $\preceq_A = \cup_{a \in A}\{(x,(1,a)) : x \leq a\} \cup \leq_{\mathbb{R}\cap[0,1]}$ and $\mathscr{L}_A : x \mapsto \alpha\,; (1,a) \mapsto \beta$. We claim that the GSTs $G_{\mathbb{Q}\cap(0,1)}$ and $G_{(0,1)\setminus\mathbb{Q}}$ together satisfy the VHHM property: in fact their surrogate Kripke structures are both bisimilar to the same image-finite Kripke structure. Nevertheless, they are clearly not strongly bisimilar, since there is no order preserving way of matching $\mathbb{Q}\cap(0,1)$ with $(0,1)\setminus\mathbb{Q}$.*

Second, we note that there are GSTs that don't belong to any VHHM class. This is ultimately because there are Kripke structures that don't belong to any VHHM class of Kripke structures: the following example describes just such a Kripke structure.

Example 5. *Consider the Kripke structure depicted in Figure 3 with a valuation that assigns all propositional variables to be true in all states. We claim that the shaded states satisfy the same formulas, yet they are clearly not bisimilar. Hence, this Kripke structure doesn't belong to any VHHM class of Kripke structures.*

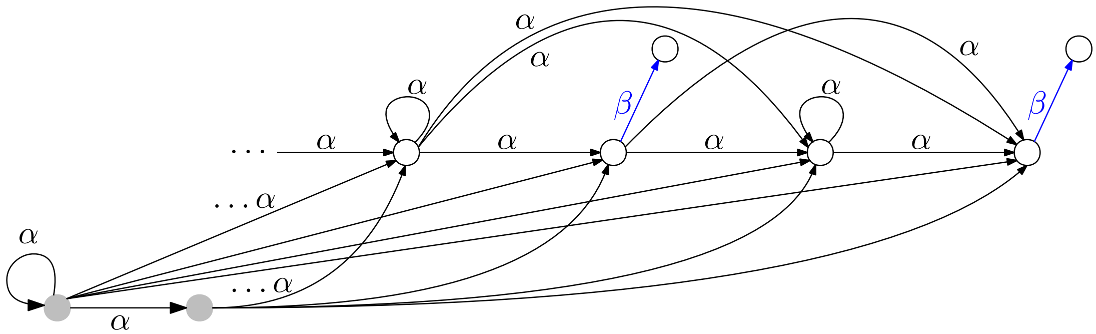


Figure 3: Kripke structure for Theorem 5. $L = \{\alpha, \beta, \alpha.\beta\}$; all black arrows have label α; $\alpha.\beta$ transitions are not shown but span each concatenated α transition and β transition.

The proof of the claim in Example 5 is nontrivial, and as far as we know, there are no results even suggesting that such Kripke structures exist. Importantly, Example 5 implies a similar example for GSTs because it contains a Kripke structure that also satisfies the schemata in Theorem 11. The following example makes this explicit.

Example 6. *Recall the definition of G_A from Example 4, and consider the GST G_X for $X = \{1/2 + 1/(n+2) : n \in \mathbb{N}\} \subset (0,1)$. Then G_X doesn't belong to any VHHM class of GSTs because its surrogate Kripke structure is bisimilar to the Kripke structure in Example 5 (when it is suitably relabeled).*

6 Conclusions and Future Work

In this paper we have proposed a generalization of Hennessy-Milner logic that is suitable for GSTs, and we have used this logic to exhibit some results regarding Hennessy-Milner classes with respect to weak bisimulation. Nevertheless, there is a great deal of work still to be done. One key avenue of future research lies in deciding whether the characterization in Theorem 11 really describes all maximal VHHM classes of GSTs (given a sufficiently large set of GSTs to begin with). Another important avenue of future work is to investigate what implications these VHHM classes have for common hybrid system models, such as the behavioral modeling framework of [18].

References

[1] Carlos E. Budde, Pedro R. D'Argenio, Pedro Sánchez Terraf & Nicolás Wolovick (2014): *A Theory for the Semantics of Stochastic and Non-deterministic Continuous Systems*. In: *Stochastic Model Checking. Rigorous Dependability Analysis Using Model Checking Techniques for Stochastic Systems*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 67–86, doi:10.1007/978-3-662-45489-3_3.

[2] Rance Cleaveland (1990): *On automatically explaining bisimulation inequivalence*. In: *International Conference on Computer Aided Verification*, Springer, pp. 364–372, doi:10.1007/BFb0023750.

[3] Pedro R. D'Argenio & Pedro Sánchez Terraf (2012): *Bisimulations for non-deterministic labelled Markov processes*. *Mathematical Structures in Computer Science* 22(1), pp. 43–68, doi:10.1017/S0960129511000454.

[4] J M Davoren & Paulo Tabuada (2007): *On Simulations and Bisimulations of General Flow Systems*. In: *Hybrid Systems: Computation and Control*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 145–158, doi:10.1007/978-3-540-71493-4_14.

[5] Josée Desharnais, Abbas Edalat & Prakash Panangaden (2002): *Bisimulation for Labelled Markov Processes*. *Information and Computation* 179(2), pp. 163–193, doi:10.1006/inco.2001.2962.

[6] E. Doberkat (2005): *Stochastic Relations: Congruences, Bisimulations and the Hennessy–Milner Theorem*. *SIAM Journal on Computing* 35(3), pp. 590–626, doi:10.1137/S009753970444346X.

[7] Ernst-Erich Doberkat (2007): *The Hennessy–Milner equivalence for continuous time stochastic logic with mu-operator*. *Journal of Applied Logic* 5(3), pp. 519–544, doi:10.1016/j.jal.2006.05.001.

[8] Ernst-Erich Doberkat & Pedro Sánchez Terraf (2017): *Stochastic non-determinism and effectivity functions*. *Journal of Logic and Computation* 27(1), pp. 357–394, doi:10.1093/logcom/exv049.

[9] James Ferlez, Rance Cleaveland & Steve Marcus (2014): *Generalized Synchronization Trees*. In: *Foundations of Software Science and Computation Structures (FoSSaCS)*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 304–319, doi:10.1007/978-3-642-54830-7_20.

[10] Robert Goldblatt (1992): *Logics of Time and Computation*, Second edition. CSLI Lecture Notes, Stanford University, Center for the Study of Language and Information. Available at `http://www.press.uchicago.edu/ucp/books/book/distributed/L/bo3615704.html`.

[11] Robert Goldblatt (1995): *Saturation and the Hennessy-Milner Property*. In Alban Ponse, Maarten de Rijke & Yde Venema, editors: *Modal Logic and Process Algebra: A Bisimulation Perspective*, Center for the Study of Language and Information Publication Lecture Notes, Cambridge University Press, pp. 107–129.

[12] Valentin Goranko & Martin Otto (2007): *Model theory of modal logic*. In Johan Van Benthem and Frank Wolter Patrick Blackburn, editor: *Studies in Logic and Practical Reasoning*, *Handbook of Modal Logic* 3, Elsevier, pp. 249–329, doi:10.1016/S1570-2464(07)80008-5.

[13] Matthew Hennessy & Robin Milner (1985): *Algebraic laws for nondeterminism and concurrency*. *Journal of the ACM*, doi:10.1145/2455.2460.

[14] Marco Hollenberg (1995): *Hennessy-Milner Classes and Process Algebra*. In Alban Ponse, Maarten de Rijke & Yde Venema, editors: *Modal Logic and Process Algebra: A Bisimulation Perspective*, Center for the Study of Language and Information Publication Lecture Notes, Cambridge University Press, pp. 187–216.

[15] Thomas Jech (2003): *Set Theory*, Third Millenium edition. Springer Monographs in Mathematics, Springer-Verlag Berlin Heidelberg, doi:10.1007/3-540-44761-X.

[16] Robin Milner (1990): *Operational and Algebraic Semantics of Concurrent Processes*. In: *Handbook of Theoretical Computer Science (Vol. B)*, MIT Press, Cambridge, MA, USA, pp. 1201–1242, doi:10.1016/B978-0-444-88074-1.50024-X.

[17] Pedro Sánchez Terraf (2015): *Bisimilarity is not Borel*. *Mathematical Structures in Computer Science*, pp. 1–20, doi:10.1017/S0960129515000535.

[18] Jan Willems (2007): *The Behavioral Approach to Open and Interconnected Systems*. *IEEE Control Systems Magazine* 27(6), pp. 46–99, doi:10.1109/MCS.2007.906923.

Reversing Imperative Parallel Programs

James Hoey Irek Ulidowski
Department of Informatics
University of Leicester, UK
jbh11@leicester.ac.uk iu3@leicester.ac.uk

Shoji Yuen
Graduate School of Information Science
Nagoya University, Japan
yuen@is.nagoya-u.ac.jp

We propose an approach and a subsequent extension for reversing imperative programs. Firstly, we produce both an augmented version and a corresponding inverted version of the original program. Augmentation saves reversal information into an auxiliary data store, maintaining segregation between this and the program state, while never altering the data store in any other way than that of the original program. Inversion uses this information to revert the final program state to the state as it was before execution. We prove that augmentation and inversion work as intended, and illustrate our approach with several examples. We also suggest a modification to our first approach to support non-communicating parallelism. Execution interleaving introduces a number of challenges, each of which our extended approach considers. We define annotation and redefine inversion to use a sequence of statement identifiers, making the interleaving order deterministic in reverse.

1 Introduction

Reverse computation has been an active research area for a number of years. The ability to reverse execute, or invert, a program is desirable due to its potential applications. The relationship with the Landauer principle shows reverse computation to be a feasible solution for producing low power, energy efficient computation [10]. In this paper, we consider reverse computation within the setting of imperative programs. We first propose a state-saving approach for reversing such programs consisting of assignments, conditional statements and while loops. We display an example of this approach showing the execution can now be reversed, and we verify that this reversal is correct. Secondly, we discuss the challenges faced when introducing parallelism, as well as the required modifications to our first approach in order to support it. The formal definition and accompanying example demonstrate the reversal of a parallel program. Finally, we present correctness results for this modified approach.

The most obvious approach to implementing program reversal is to record the entire program state before executing the program. Recording all of the initial variable values does allow immediate reversal to the original state, however suffers several setbacks, including not re-creating the intermediate program states, and the production of garbage data. We propose an approach that records the necessary information to reverse an execution step-by-step, re-creating intermediate steps faithfully allowing movement in both directions at any point. Any information we save as a result of this is used during inversion, meaning no garbage data is produced.

Inspired by the Reverse C Compiler (RCC) [12, 3], our initial approach takes an original program and produces two versions. The first is the *augmented version*, which becomes the program used for forward execution, and has the capability to save all information necessary for inversion, termed *reversal information*. This is implemented via the function *aug* that analyses the original program statement by statement, producing the augmented version. The execution of this version populates a collection of initially empty stacks, termed an auxiliary store δ, with this reversal information. Consider the program shown in Figure 1, producing the `Nth` element of a Fibonacci-like sequence beginning with the values

K. Peters and S. Tini (Eds.): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics (EXPRESS/SOS 2017).
EPTCS 255, 2017, pp. 51–66, doi:10.4204/EPTCS.255.4

```
1   if X > Y then
2     Z = Y;
3     Y = X;
4     X = Z;
5   else
6     skip
7   end
8
9   while N−2 > 0 do
10    Z = X;
11    X = Y;
12    Y += Z;
13    N −= 1;
14  end
```

Figure 1: Original program

```
1   while pop(δ(W)) do
2     N +=1;
3     Y −= Z;
4     X = pop(δ(X));
5     Z = pop(δ(Z));
6   end
7
8   if pop(δ(B)) then
9     X = pop(δ(X));
10    Y = pop(δ(Y));
11    Z = pop(δ(Z));
12  else
13    skip
14  end
```

Figure 2: Inverted program

of X and Y. Let the initial state σ consist of X=4, Y=3, Z=0, N=5 and the initial auxiliary store δ consist of empty stacks. The execution of the augmented version (displayed later in Section 4, Figure 3) under these stores results in the state σ' where X=11, Y=18, Z=7, N=2 and auxiliary store δ' containing reversal information detailed in Section 4. With this version now being used for forwards execution, it is crucial that the behaviour with respect to the program state is unchanged. Our first result ensures that if the state σ' is produced via the original execution, then it must also be produced via the augmented execution.

The second version, termed the *inverted version*, is generated via the function *inv*. This version follows the inverted execution order of the original, containing a statement corresponding to each of those of the original. Each inverted statement will typically use information from the auxiliary store to revert all of the effects caused via execution of the original. Consider again the example in Figure 1. Application of *inv* to this program produces the inverted version, shown in Figure 2. Execution of this program under the stores σ' and δ' produces the state σ'' where X=4, Y=3, Z=0, N=5, and the auxiliary store δ'' containing only empty stacks. Our second result validates that $\sigma'' = \sigma$ and $\delta'' = \delta$, meaning the inversion has happened correctly and the initial program state has been restored. Doing so proves that the augmented program saves the required information, that the inverted program is capable of using this to restore the program state to exactly as it was before, and that augmentation produces no garbage data.

In the second part of the paper, we define a modified approach, this time capable of supporting non-communicating parallelism [9]. Issues introduced such as a non-deterministic execution order make our previous approach insufficient without further state-saving. The *interleaving order*, or order in which the statements are executed, now forms part of the reversal information, captured and stored at runtime. Storing this interleaving order makes the program deterministic in reverse, guaranteeing the execution of the inverted program always follows exactly the inverse execution order of the original. Modifications are made to the process of augmentation from our first approach, with all state-saving implemented at runtime via a set of modified operational semantics for forward execution. A similar reasoning is applied to the process of inversion, resulting in a modified set of operational semantics for reverse execution. These semantics are responsible for using the reversal information, including the interleaving order, to reverse the statements of the original program in exactly the opposite order. Finally the correctness results of our second approach are presented.

The paper is organised as follows. Section 2 introduces the programming language and its notion

of program state, with the operational semantics given in Section 3. Section 4 describes the process of augmentation and the information that must be saved, as well as proving the correctness of this augmentation. Section 5 defines the process of inversion and again proves the correctness of this process. Section 6 introduces an updated approach capable of supporting parallel composition, as well as presenting the correctness results.

1.1 Related Work

Program inversion has been discussed for many years, including the work by Gries [8] and by Glück and Kawabe [6, 7]. The Reverse C Compiler as described by Perumalla et al. [12, 3] is one example of a state-saving approach for the reversal of C programs. We relate very closely to this approach, but with differences including that we currently support a smaller language, and we record a while loop sequence in order to avoid modifying the behaviour of the original program (see Section 4). To the best of our knowledge, there is no formal proof of correctness of RCC, and so this is a major focus of our work. Our approach proposes the foundation from which a formally proved approach for a more complex language could emerge. Other work has been produced on reverse computation used within Parallel Discrete Event Simulation (PDES), a simulation methodology capable of executing events speculatively [5, 4]. The backstroke framework [17] and subsequent work on it by Schordan et al. [15, 16] relates slightly less closely to our work as it focuses on this application to PDES. Backstroke is capable of both a state-saving approach and a more advanced, path regeneration method for reverse computation. Other applications include to debugging, with examples being [2, 1]. Similarly to program inversion, the reversible programming language Janus, originally proposed in [11] requires additional information within the source code. Any program written in Janus is fully reversible, without the requirement for any control information to be recorded, but with a requirement for additional assertions that make the program deterministic in both directions [19, 18].

2 Programming Language and Program State

The programming language used for our first approach is similar to any *while language*, particularly that of Hüttel [9]. This consists of destructive and constructive assignments, with the expression not containing the variable in question, or any side effects. Conditional statements and loops are also supported, implemented using both arithmetic and Boolean expressions. Let the set of variables $\mathbb{V}$ be ranged over by `X`, `Y`, `Z` ..., the set of integers $\mathbb{Z}$ be ranged over by `l`, `m` and `n` and the set of Boolean values $\mathbb{B}$ be $\{$`T`,`F`$\}$. Also let `Cop` be the set of constructive assignment operators $\{$`+=`,`-=`$\}$ with `cop` $\in$ `Cop`, and `Op` be the set of arithmetic operators $\{$`+`,`-`$\}$ with `op` $\in$ `Op`.

```
P   ::= ε | S; P
S   ::= skip | X = Exp | X Cop Exp | if B then P else P end |
        while B do P end
B   ::= T | F | ¬B | (B) | Exp == Exp | Exp > Exp | B ∧ B
Exp ::= X | n | (Exp) | Exp Op Exp
```

$\mathbb{P}$ is the set of programs, ranged over by `P`, `Q` and `R`. $\mathbb{S}$ is the set of statements, ranged over by `S`. Expressions `Exp` are ranged over by `a`, $\mathtt{a}_0$, $\mathtt{a}'_0$, $\mathtt{a}_1$, $\mathtt{a}'_1$, Boolean expressions `B` are ranged over by `b`, $\mathtt{b}'$, $\mathtt{b}_0$, $\mathtt{b}_1$ and expressions that can be either are ranged over by `ba`, $\mathtt{ba}_0$, $\mathtt{ba}'_0$, $\mathtt{ba}_1$, $\mathtt{ba}'_1$.

The program state is represented via a data store σ, responsible for mapping each variable to the value it currently holds. A data store is represented as a set of pairs, with the first element of the pair being the variable name, and the second being its current value. A data store is represented formally as the partial function $\sigma : \mathbb{V} \to \mathbb{Z}$.

Such stores are manipulated using the following notation. Assuming $\mathtt{v} \in \mathbb{Z}$, $\sigma(\mathtt{X})$ returns the value currently associated to the variable X, while $\sigma[\mathtt{X} \mapsto \mathtt{v}]$ produces a store identical to σ, but with the variable X now holding the value v.

Consider the store σ, consisting of two variables X and Y, with values 3 and 5 respectively, described as $\sigma = \{(\mathtt{X}, 3), (\mathtt{Y},5)\}$. The statement $\sigma(\mathtt{X})$ returns 3, and $\sigma[\mathtt{X} \mapsto 10]$ results in the store $\{(\mathtt{X},10), (\mathtt{Y},5)\}$.

3 Structured Operational Semantics

This section defines the Structured Operational Semantics (SOS) of the programming language described above. These are defined in the traditional way, following closely with those of Hüttel [9]. The parameter δ, representing the auxiliary store, is not strictly necessary at this point, but is required later and included here for consistency.

3.1 Arithmetic Statements

Let $\mathtt{v} \in \mathbb{Z}$ and recall $\mathtt{op} \in \mathtt{Op}$.

$$\frac{}{(\mathtt{X},\sigma,\delta) \to (\sigma(\mathtt{X}),\sigma,\delta)} \quad \frac{\mathtt{v} \;=\; \mathtt{n}\ \mathtt{op}\ \mathtt{m}}{(\mathtt{n}\ \mathtt{op}\ \mathtt{m},\sigma,\delta) \to (\mathtt{v},\sigma,\delta)} \quad \frac{}{((\mathtt{v}),\sigma,\delta) \to (\mathtt{v},\sigma,\delta)} \quad \frac{(\mathtt{a}_0,\sigma,\delta) \to (\mathtt{a}_0',\sigma',\delta')}{((\mathtt{a}_0),\sigma,\delta) \to ((\mathtt{a}_0'),\sigma',\delta')}$$

$$\frac{(\mathtt{a}_0,\sigma,\delta) \to (\mathtt{a}_0',\sigma',\delta')}{(\mathtt{a}_0\ \mathtt{op}\ \mathtt{a}_1,\sigma,\delta) \to (\mathtt{a}_0'\ \mathtt{op}\ \mathtt{a}_1,\sigma',\delta')} \quad \frac{(\mathtt{a}_1,\sigma,\delta) \to (\mathtt{a}_1',\sigma',\delta')}{(\mathtt{a}_0\ \mathtt{op}\ \mathtt{a}_1,\sigma,\delta) \to (\mathtt{a}_0\ \mathtt{op}\ \mathtt{a}_1',\sigma',\delta')}$$

3.2 Boolean Expressions

Let $\mathtt{bop} \in \{\mathtt{>},\mathtt{==}\}$ if used between two arithmetic expressions or $\mathtt{bop} \in \{\wedge,\mathtt{==}\}$ if used between two Boolean expressions.

$$\frac{}{(\neg\mathtt{T},\sigma,\delta) \to (\mathtt{F},\sigma,\delta)} \quad \frac{}{(\neg\mathtt{F},\sigma,\delta) \to (\mathtt{T},\sigma,\delta)} \quad \frac{(\mathtt{b},\sigma,\delta) \to (\mathtt{b}',\sigma',\delta')}{(\neg\mathtt{b},\sigma,\delta) \to (\neg\mathtt{b}',\sigma',\delta')} \quad \frac{\mathtt{ba}_2 = \mathtt{ba}_0\ \ \mathtt{bop}\ \ \mathtt{ba}_1}{(\mathtt{ba}_0\ \ \mathtt{bop}\ \ \mathtt{ba}_1,\sigma,\delta) \to (\mathtt{ba}_2,\sigma,\delta)}$$

$$\frac{(\mathtt{ba}_0,\sigma,\delta) \to (\mathtt{ba}_0',\sigma',\delta')}{(\mathtt{ba}_0\ \mathtt{bop}\ \mathtt{ba}_1,\sigma,\delta) \to (\mathtt{ba}_0'\ \mathtt{bop}\ \mathtt{ba}_1,\sigma',\delta')} \quad \frac{(\mathtt{ba}_1,\sigma,\delta) \to (\mathtt{ba}_1',\sigma',\delta')}{(\mathtt{ba}_0\ \mathtt{bop}\ \mathtt{ba}_1,\sigma,\delta) \to (\mathtt{ba}_0\ \mathtt{bop}\ \mathtt{ba}_1',\sigma',\delta')}$$

3.3 Program Statements

Let $\mathtt{v} \in \mathbb{Z}$ and recall $\mathtt{cop} \in \mathtt{Cop}$. Let op be + if cop = +=, otherwise let op be −.

[Skip] $\dfrac{}{(\mathtt{skip;P},\sigma,\delta) \to (\mathtt{P},\sigma,\delta)}$ [Seq] $\dfrac{(\mathtt{S},\sigma,\delta) \to (\mathtt{S}',\sigma',\delta')}{(\mathtt{S;P},\sigma,\delta) \to (\mathtt{S}';\mathtt{P},\sigma',\delta')}$

[DA1] $\dfrac{}{(\mathtt{X}\ =\ \mathtt{v},\sigma,\delta) \to (\mathtt{skip},\sigma[\mathtt{X} \mapsto \mathtt{v}],\delta)}$ [DA2] $\dfrac{(\mathtt{a},\sigma,\delta) \to (\mathtt{a}',\sigma',\delta')}{(\mathtt{X}\ =\ \mathtt{a},\sigma,\delta) \to (\mathtt{X}\ =\ \mathtt{a}',\sigma',\delta')}$

[CA1] $\dfrac{}{(\mathtt{X}\ \mathtt{cop}\ \mathtt{v},\sigma,\delta) \to (\mathtt{skip},\sigma[\mathtt{X} \mapsto \sigma(\mathtt{X})\ \mathtt{op}\ \mathtt{v}],\delta)}$ [CA2] $\dfrac{(\mathtt{a},\sigma,\delta) \to (\mathtt{a}',\sigma',\delta')}{(\mathtt{X}\ \mathtt{cop}\ \mathtt{a},\sigma,\delta) \to (\mathtt{X}\ \mathtt{cop}\ \mathtt{a}',\sigma',\delta')}$

$$\text{[C1]}\ \frac{}{(\texttt{if T then P else Q end}, \sigma, \delta) \to (\texttt{P}, \sigma, \delta)} \qquad \text{[C2]}\ \frac{}{(\texttt{if F then P else Q end}, \sigma, \delta) \to (\texttt{Q}, \sigma, \delta)}$$

$$\text{[C3]}\ \frac{(\texttt{b}, \sigma, \delta) \to (\texttt{b}', \sigma', \delta')}{(\texttt{if b then P else Q end}, \sigma, \delta) \to (\texttt{if b}' \texttt{ then P else Q end}, \sigma', \delta')}$$

$$\text{[Wh]}\ \frac{}{(\texttt{P}, \sigma, \delta) \to (\texttt{if b then Q;P else skip end}, \sigma, \delta)} \quad \text{where } \texttt{P} = \texttt{while b do Q end}$$

4 Augmentation

The first step of our first approach is to generate the *augmented version* through a process termed augmentation. This process takes each statement of the original program in succession, and returns a semantically equivalent (with respect to the data store) code fragment containing any required state-saving operations. These fragments are then combined to produce the augmented version.

The information required to be saved depends on the type of statement. Destructive assignments discard the old value of a variable, meaning it must be saved. Constructive assignments do not suffer this problem meaning they are reversible without state-saving. Due to no guarantee that a condition is invariant, conditional statements must save control information indicating which branch was executed. While loops not having a fixed number of iterations means the number of times the loop should be inverted is unknown. Therefore a sequence of Booleans representing the while loop is saved.

Saving the result of evaluating conditional statements and while loops removes the burden of re-evaluating these expressions during inversion, unlike the reversible programming language Janus that does require this. In an effort to ensure that the state-saving does not affect the behaviour of the program (w.r.t. the data store), all reversal information is stored separately in an auxiliary data store.

4.1 Auxiliary Data Store

Recall that $\mathbb{V}$ is the set of program variable names, and now let both `B` and `W` be reserved keywords that cannot appear within this set. An auxiliary data store δ is a set of stacks, consisting of one self-named stack for each program variable within $\mathbb{V}$, one stack `B` for all conditional statements and one stack `W` for all while loops. More formally, $\delta : (\mathbb{V} \to \mathbb{X}) \cup (\{\texttt{B},\texttt{W}\} \cup \mathbb{B}')$, where $\mathbb{X}$ is the set of stacks of integers and $\mathbb{B}'$ is the set of stacks of Booleans. Auxiliary stores will be represented as a set of pairs. Each pair represents a stack, with the first element being the stack name and the second element being the sequence of its elements. The order of this sequence reflects that of the stack, with the left-most element being the head of the stack. Consider a program consisting of one variable `X` (initially `1`) destructively assigned twice (to `3` and `5`), one conditional statement that evaluates to `T` and a while loop with one iteration. The final auxiliary store would be $\{(\texttt{X},\{\texttt{3},\texttt{1}\}), (\texttt{B},\{\texttt{T}\}), (\texttt{W},\{\texttt{T},\texttt{F}\})\}$.

The stacks on δ are manipulated in the traditional manner [9], using `push` and `pop` operations introduced via augmentation. The notation $\delta[\texttt{v} \mapsto \texttt{X}]$ and `push(v,X)` both represent pushing the value `v` to the stack `X`, while $\delta[\texttt{X}]$ and `pop(X)` represent popping the stack `X`. Further notation includes $\delta(\texttt{S})$ that returns the stack named `S`, `v:S` that indicates a stack with head `v` and tail `S`, and $\delta[\texttt{X}/\texttt{X}']$ that states the stack `X` is replaced by $\texttt{X}'$. The SOS rules are defined, where $\texttt{v} \in \mathbb{Z} \cup \mathbb{B}$.

$$\text{[Pop]}\ \frac{\delta(\texttt{X}) = \texttt{v:X}'}{(\texttt{pop}(\delta(\texttt{X})), \sigma, \delta) \to (\texttt{v}, \sigma, \delta[\texttt{X}/\texttt{X}'])} \qquad \text{[Push1]}\ \frac{}{(\texttt{push}(\texttt{v},\delta(\texttt{X})), \sigma, \delta) \to (\texttt{skip}, \sigma, \delta[\texttt{v} \mapsto \texttt{X}])}$$

$$\text{[Push2]}\quad \frac{(\texttt{ba},\sigma,\delta) \to (\texttt{ba}',\sigma',\delta')}{(\texttt{push(ba,}\delta\texttt{(X))},\sigma,\delta) \to (\texttt{push(ba}',\delta\texttt{(X))},\sigma',\delta')}$$

We are now ready to introduce the function that performs the augmentation.

4.2 Augmentation Function

Let $\hat{\mathbb{P}}$ be the set of augmented programs. The function $aug : \mathbb{P} \to \hat{\mathbb{P}}$ takes the original program and recursively applies the function $a : \mathbb{S} \to \hat{\mathbb{P}}$ to each statement, producing its augmented version.

Destructive assignments are augmented into two statements, one to push the old value of the variable to its self-named stack on δ, and a second to perform the assignment (see 4). Constructive assignments are left unchanged due to their reversibility (see 5). Conditional statements have each branch recursively augmented, as well as extended with an operation that stores a Boolean indicating whether the true or false branch was executed (see 6). As such, *aug* and *a* are now defined, where $\texttt{cop} \in \texttt{Cop}$.

$$aug(\varepsilon) = \varepsilon \tag{1}$$
$$aug(\texttt{S;P}) = a(\texttt{S});\ aug(\texttt{P}) \tag{2}$$

$$a(\texttt{skip}) = \texttt{skip} \tag{3}$$
$$a(\texttt{X = a}) = \texttt{push}(\sigma(\texttt{X}),\delta(\texttt{X}))\texttt{; X = a} \tag{4}$$
$$a(\texttt{X cop a}) = \texttt{X cop a} \tag{5}$$
$$\begin{aligned} a(\texttt{if b then P else Q end}) = \texttt{if b then }&aug(\texttt{P})\texttt{; push(T,}\delta\texttt{(B))} \\ &\texttt{else }aug(\texttt{Q})\texttt{; push(F,}\delta\texttt{(B)) end} \end{aligned} \tag{6}$$

The traditional approach of handling while loops by initialising a counter and incrementing it for each iteration is not used here due to its adverse effects on the behaviour of the program w.r.t. the data store. While loops are instead augmented to save a sequence of Booleans representing its execution. Generating the sequence in the intuitive way (of a T for each iteration and finally an F) and storing this onto a traditional stack will require the sequence to be manipulated before being used. Such manipulation is both difficult, due to ambiguities within such sequences, and avoidable, by storing a usable order to begin with.

The desired order is that of the intuitive approach, but with any opening T switched with its corresponding closing F, while maintaining any nested T elements. This sequence can be generated provided we can distinguish between the first iteration of a loop and any other. The first iteration now requires an F, while any subsequent iteration (including the unsuccessful last iteration) requires a T (see 7).

$$\begin{aligned} a(\texttt{while b do P end}) = \texttt{if b then}&\\ &\texttt{push(F,}\delta\texttt{(W))}\texttt{; }aug(\texttt{P})\texttt{;}\\ &\texttt{while b do}\\ &\quad\texttt{push(T,}\delta\texttt{(W))}\texttt{; }aug(\texttt{P})\\ &\texttt{end; push(T,}\delta\texttt{(W))}\\ \texttt{else push(F,}\delta\texttt{(W)) end}& \end{aligned} \tag{7}$$

We now return to our example discussing Figure 1. The augmented version of this program is shown in Figure 3. The destructive assignment of Z on line 2 of Figure 1 corresponds to line 2 of Figure 3,

```
 1 if X > Y then
 2    push(σ(Z),δ(Z)); Z = Y;
 3    push(σ(Y),δ(Y)); Y = X;
 4    push(σ(X),δ(X)); X = Z;
 5    push(T,δ(B))
 6 else
 7    skip;push(F,δ(B))
 8 end
 9 if N−2 > 0 then
10    push(F,δ(W));
11    push(σ(Z),δ(Z)); Z = X;
12    push(σ(X),δ(X)); X = Y;
13    Y += Z;
14    N −= 1;
15    while N−2 > 0 do
16       push(T,δ(W));
17       push(σ(Z),δ(Z)); Z = X;
18       push(σ(X),δ(X)); X = Y;
19       Y += Z;
20       N −= 1
21    end
22    push(T,δ(W))
23 else
24    push(F,δ(W)); end
```

Figure 3: Augmented Version of the Program in Figure 1

where the push statement is used to first save the old value. Lines 5 and 7 of Figure 3 contain inserted operations to save the result of evaluating the conditional statement, while lines 10, 16, 22 and 24 are inserted commands to save the sequence of Boolean values representing the execution of the while loop. Execution of this program under the initial stores σ = {(X,4), (Y,3), (Z,0), (N,5)} and δ = {(X,{}), (Y,{}), (Z,{}), (N,{}), (B,{}), (W,{})}, produces the final stores σ' = {(X,11), (Y,18), (Z,7), (N,2)} and δ' = {(X,{7,4,3,4}), (Y,{3}), (Z,{4,3,3,0}), (N,{}), (B,{T}), (W,{T,T,T,F})}. The two final stores now contain all of the necessary information for reversal.

We are now ready to state our first result. Firstly, Proposition 1 states that if the execution of an original program terminates, then the execution of the augmented version of that program also terminates (where a program terminates if its execution finishes with the configuration $(\texttt{skip}, \sigma^*, \delta^*)$ for some σ^* and δ^*). Secondly, Proposition 1 states that augmentation produces an augmented version that modifies the data store σ in exactly the same way as that of the original program to σ', while also populating the auxiliary store δ with reversal information producing δ'.

Proposition 1. *Let* P *be a program that does not interact with the auxiliary store, σ be an arbitrary initial data store and δ be an arbitrary initial auxiliary data store. Firstly, if $(\texttt{P}, \sigma, \delta) \rightarrow^* (\texttt{skip}, \sigma', \delta'')$, for some σ' and δ'', then $(aug(\texttt{P}), \sigma, \delta) \rightarrow^* (\texttt{skip}, \sigma'', \delta''')$ for some σ'' and δ'''. Secondly, if $(\texttt{P}, \sigma, \delta) \rightarrow^* (\texttt{skip}, \sigma', \delta)$, for some σ', then $(aug(\texttt{P}), \sigma, \delta) \rightarrow^* (\texttt{skip}, \sigma', \delta')$ for some δ'.*

We note that the inverse implication, namely that if $(aug(\texttt{P}), \sigma, \delta) \rightarrow^* (\texttt{skip}, \sigma', \delta')$, for some σ' and δ', then $(\texttt{P}, \sigma, \delta) \rightarrow^* (\texttt{skip}, \sigma', \delta)$, would also be valid. However we defer this proof to future work, and now return to proving the second part of Proposition 1 (with the first following correspondingly).

Proof. By induction on the length of the sequence $(\texttt{P}, \sigma, \delta) \rightarrow^* (\texttt{skip}, \sigma', \delta)$. Since there are no transitions of length 0, the proposition holds vacuously. Assume that the proposition holds for programs R, stores σ^* and auxiliary stores δ^*, such that $(\texttt{R}, \sigma^*, \delta^*) \rightarrow^* (\texttt{skip}, \sigma_1^*, \delta^*)$ is shorter than $(\texttt{P}, \sigma, \delta) \rightarrow^* (\texttt{skip}, \sigma', \delta)$. Further assuming P is of the form S;P′ such that S is a statement and P′ is the remaining program, we have that

$$(\texttt{S;P}', \sigma, \delta) \rightarrow^* (\texttt{skip}, \sigma', \delta)$$

for some σ'. Through use of the SOS rules Seq and Skip, we have

$$(\texttt{S;P}', \sigma, \delta) \rightarrow^* (\texttt{skip;P}', \sigma'', \delta) \rightarrow (\texttt{P}', \sigma'', \delta) \rightarrow^* (\texttt{skip}, \sigma', \delta)$$

for some σ''. With this in mind, we need to show that $(aug(\texttt{S;P}'),\sigma,\delta) \rightarrow^* (\texttt{skip},\sigma',\delta')$ for some δ'.

By the definition of *aug*, clause (2) we have $(aug(\texttt{S;P}'),\sigma,\delta) = (a(\texttt{S});aug(\texttt{P}'),\sigma,\delta)$, meaning it is sufficient to prove

$$(a(\texttt{S});aug(\texttt{P}'),\sigma,\delta) \rightarrow^* (\texttt{skip};aug(\texttt{P}'),\sigma'',\delta'') \rightarrow (aug(\texttt{P}'),\sigma'',\delta'') \rightarrow^* (\texttt{skip},\sigma',\delta')$$

for some σ'', δ'' and δ'. Since $(\texttt{S;P}',\sigma,\delta) \rightarrow^* (\texttt{P}',\sigma'',\delta)$, then repeated use of the Seq rule (from conclusion to premises) produces $(\texttt{S},\sigma,\delta) \rightarrow^* (\texttt{skip},\sigma'',\delta)$. Now assume $a(\texttt{S}) = \texttt{P}_S$ for each type of statement S. Then by Lemma 1 below, we have that $(\texttt{P}_S,\sigma,\delta) \rightarrow^* (\texttt{skip},\sigma'',\delta'')$ for some δ''. Using the Seq rule (from premises to conclusion) we obtain

$$(\texttt{P}_S;aug(\texttt{P}'),\sigma,\delta) \rightarrow^* (\texttt{skip};aug(\texttt{P}'),\sigma'',\delta'')$$

Then by the Skip rule, we get $(\texttt{skip};aug(\texttt{P}'),\sigma'',\delta'') \rightarrow (aug(\texttt{P}'),\sigma'',\delta'')$. The induction hypothesis on $(\texttt{P}',\sigma'',\delta'')$ gives us

$$(aug(\texttt{P}'),\sigma'',\delta'') \rightarrow^* (\texttt{skip},\sigma',\delta')$$

for some δ'. Therefore we have obtained $(a(\texttt{S});aug(\texttt{P}'),\sigma,\delta) \rightarrow^* (aug(\texttt{P}'),\sigma'',\delta'') \rightarrow^* (\texttt{skip},\sigma',\delta')$, meaning $(aug(\texttt{S;P}'),\sigma,\delta) \rightarrow^* (\texttt{skip},\sigma',\delta')$ holds as required. Therefore the proposition holds, provided the following lemma holds. □

Lemma 1. *Let S be a statement that does not interact with the auxiliary store, σ be an initial data store and δ be an initial auxiliary data store. If $(\texttt{S},\sigma,\delta) \rightarrow^* (\texttt{skip},\sigma',\delta)$ for some σ' then $(a(\texttt{S}),\sigma,\delta) \rightarrow^* (\texttt{skip},\sigma',\delta')$ for some δ'.*

Proof. We consider each type of statement S in turn. Due to space constraints, we only include one case, with the other cases following similarly. The notation $(\texttt{Q},\sigma,\delta) \underset{\text{X}}{\rightarrow}^{l} (\texttt{Q}',\sigma^\dagger,\delta^\dagger)$ denotes l transitions by the SOS rule X produces the program Q′, store $\sigma^\dagger$ and auxiliary store $\delta^\dagger$.

Case 1.1. Consider statement X=a and its execution under the initial stores σ and δ where X is initially v′ and a evaluates to v in l steps such that

$$(\texttt{X=a},\sigma,\delta) \underset{\text{DA2}}{\longrightarrow}^{l} (\texttt{X=v},\sigma,\delta) \underset{\text{DA1}}{\longrightarrow} (\texttt{skip},\sigma[\texttt{X} \mapsto \texttt{v}],\delta).$$

Recall that $\sigma(\texttt{X}) = \texttt{v}'$. The execution of the augmented version of X=a is

$$\begin{aligned}&(\texttt{push(v',}\delta\texttt{(X));X=a},\sigma,\delta) \underset{\text{Push}_1,\ \text{Skip}}{\longrightarrow} (\texttt{X=a},\sigma,\delta[\texttt{v}' \mapsto \texttt{X}])\\ &\underset{\text{DA2}}{\longrightarrow}^{l} (\texttt{X=v},\sigma,\delta[\texttt{v}' \mapsto \texttt{X}]) \underset{\text{DA1}}{\longrightarrow} (\texttt{skip},\sigma[\texttt{X} \mapsto \texttt{v}],\delta[\texttt{v}' \mapsto \texttt{X}])\end{aligned}$$

As such, this case holds with $\sigma' = \sigma[\texttt{X} \mapsto \texttt{v}]$ and $\delta' = \delta[\texttt{v}' \mapsto \texttt{X}]$.

With all other cases following in a similar manner, Lemma 1 holds. □

5 Inversion

The second step of our initial approach is to generate the inverted version through a process named inversion. Inversion takes each statement in reverse order, generates the code fragment necessary to undo its effects, before combining these fragments to generate the inverted version. The majority of the

returned code fragments will use the reversal information on the auxiliary store, meaning the augmented version must be executed prior to the execution of this version.

Destructive assignments are replaced with another destructive assignment to the same variable, but this time assigning the value currently at the top of the self-named stack (see 11). Constructive assignments require no reversal information, and can simply be replaced by their inverse (see 12). Conditional statements saved a Boolean indicating which branch was executed, meaning the retrieval and evaluation of this now replaces the original condition, along with the recursive inversion of the branches (see 13). While loops saved a sequence of Booleans in the desired order, meaning the while loop can continually iterate until the top of the stack `W` is no longer true (see 14), along with the recursive inversion of the body. As mentioned earlier, the reverse execution of conditionals and loops does not require their conditions to be re-evaluated, increasing efficiency.

Let $\mathbb{P}^{-1}$ be the set of inverted programs. The function $inv : \mathbb{P} \to \mathbb{P}^{-1}$ takes the original program and recursively applies the function $i : \mathbb{S} \to \mathbb{P}^{-1}$ to each statement in reverse order, producing its inverted version. We now define *inv* and *i*, where `cop` $\in$ `Cop`, and `icop` = `+=` if `cop` = `-=`, and `-=` otherwise.

$$inv(\varepsilon) = \varepsilon \tag{8}$$

$$inv(\texttt{S;P}) = inv(\texttt{P});\ i(\texttt{S}) \tag{9}$$

$$i(\texttt{skip}) = \texttt{skip} \tag{10}$$

$$i(\texttt{X = a}) = \texttt{X = pop(}\delta\texttt{(X))} \tag{11}$$

$$i(\texttt{X cop a}) = \texttt{X icop a} \tag{12}$$

$$i(\texttt{if b then P else Q end}) = \texttt{if pop(}\delta\texttt{(B)) then } inv(\texttt{P}) \texttt{ else } inv(\texttt{Q}) \texttt{ end} \tag{13}$$

$$i(\texttt{while b do P end}) = \texttt{while pop(}\delta\texttt{(W)) do } inv(\texttt{P}) \texttt{ end} \tag{14}$$

We now return to our example code shown in Figure 1. Applying the function *inv* to this program produces the inverted program shown in Figure 2. The overall program order has been inverted, with the while loop now being executed first. An example destructive assignment is on line 2 of Figure 1, and is inverted via the line 11 of Figure 2. Execution of this version under the final stores σ' = {(X,11), (Y,18), (Z,7), (N,2)} and δ' = {(X,{7,4,3,4}), (Y,{3}), (Z,{4,3,3,0}), (N,{}), (B,{T}), (W,{T,T,T,F})}, produces the initial stores σ'' = {(X,4), (Y,3), (Z,0), (N,5)} and δ'' = {(X,{}), (Y,{}), (Z,{}), (N,{}), (B,{}), (W,{})}. As should be clear, $\sigma'' = \sigma$ and $\delta'' = \delta$, meaning the reversal has executed successfully.

We will now present our second result for $(\texttt{P}, \sigma, \delta)$. Recall that by Proposition 1, the execution of the augmented version of `P` produces the modified auxiliary store δ', which plays a crucial role in Proposition 2 below. Firstly, Proposition 2 states that if the original program `P` terminates on σ and δ, producing σ', then the execution of the inverted version of `P` terminates on σ' and the modified auxiliary store δ'. Secondly, Proposition 2 states that given the final stores σ' and δ' produced via execution of the augmented version of `P`, executing the corresponding inverted version on these stores restores the initial state, namely σ and δ.

Proposition 2. *Let* `P` *be a program that does not interact with the auxiliary store,* σ *be an arbitrary initial data store and* δ *be an arbitrary initial auxiliary data store. Firstly, if* $(\texttt{P}, \sigma, \delta) \to^* (\texttt{skip}, \sigma', \delta'')$*, for some* σ' *and* δ''*, then* $(inv(\texttt{P}), \sigma', \delta') \to^* (\texttt{skip}, \sigma'', \delta''')$*, for some* σ'' *and* δ'''*. Secondly, if* $(\texttt{P}, \sigma, \delta) \to^* (\texttt{skip}, \sigma', \delta)$*, for some* σ'*, then* $(inv(\texttt{P}), \sigma', \delta') \to^* (\texttt{skip}, \sigma, \delta)$*, for some* δ'*.*

We note that the first result in Proposition 2 would be valid, but postpone the proof to future work and now return to proving the second part of Proposition 2.

Proof. By induction on the length of the sequence $(\mathtt{P}, \sigma, \delta) \rightarrow^* (\mathtt{skip}, \sigma', \delta)$. Since there are no transitions of length 0, the proposition holds vacuously. Assume that the proposition holds for programs R, stores σ^* and auxiliary stores δ^*, such that $(\mathtt{R}, \sigma^*, \delta^*) \rightarrow^* (\mathtt{skip}, \sigma_1^*, \delta^*)$ is shorter than $(\mathtt{P}, \sigma, \delta) \rightarrow^* (\mathtt{skip}, \sigma', \delta)$. Further assume P is of the form $\mathtt{S;P'}$ such that S is a statement and $\mathtt{P'}$ is the remaining program. Let $(\mathtt{S;P'}, \sigma, \delta) \rightarrow^* (\mathtt{skip}, \sigma', \delta)$ for some σ'. This means that

$$(\mathtt{S;P'}, \sigma, \delta) \rightarrow^* (\mathtt{skip;P'}, \sigma'', \delta) \rightarrow (\mathtt{P'}, \sigma'', \delta) \rightarrow^* (\mathtt{skip}, \sigma', \delta)$$

for some σ''. By applying the Seq rule (conclusion to premises) to $(\mathtt{S;P'}, \sigma, \delta) \rightarrow^* (\mathtt{skip;P'}, \sigma'', \delta)$, we obtain $(\mathtt{S}, \sigma, \delta) \rightarrow^* (\mathtt{skip}, \sigma'', \delta)$. By the definition of *aug*, clause (2) we have $(aug(\mathtt{S;P'}), \sigma, \delta) = (a(\mathtt{S}); aug(\mathtt{P'}), \sigma, \delta)$ and by Proposition 1 we have

$$(a(\mathtt{S}); aug(\mathtt{P'}), \sigma, \delta) \rightarrow^* (\mathtt{skip}; aug(\mathtt{P'}), \sigma'', \delta'') \rightarrow (aug(\mathtt{P'}), \sigma'', \delta'') \rightarrow^* (\mathtt{skip}, \sigma', \delta')$$

for some δ'', δ'. Using Seq (conclusion to premise) on $(a(\mathtt{S}); aug(\mathtt{P'}), \sigma, \delta) \rightarrow^* (\mathtt{skip}; aug(\mathtt{P'}), \sigma'', \delta'')$ we obtain $(a(\mathtt{S}), \sigma, \delta) \rightarrow^* (\mathtt{skip}, \sigma'', \delta'')$.

We need to show that given σ' and δ', $(inv(\mathtt{S;P'}), \sigma', \delta') \rightarrow^* (\mathtt{skip}, \sigma, \delta)$. By the definition of *inv*, clause 9, we have $inv(\mathtt{S;P'}) = inv(\mathtt{P'}); i(\mathtt{S})$, meaning we shall show

$$(inv(\mathtt{P'}); i(\mathtt{S}), \sigma', \delta') \rightarrow^* (i(\mathtt{S}), \sigma^\dagger, \delta^\dagger) \rightarrow^* (\mathtt{skip}, \sigma, \delta)$$

for some $\sigma^\dagger$ and $\delta^\dagger$. The induction hypothesis for $(\mathtt{P'}, \sigma'', \delta'') \rightarrow^* (\mathtt{skip}, \sigma', \delta'')$, where δ'' is obtained by augmentation of S on δ, gives us $(inv(\mathtt{P'}), \sigma', \delta') \rightarrow^* (\mathtt{skip}, \sigma'', \delta'')$ where δ' is obtained by augmentation of $\mathtt{P'}$ on σ'' and δ'' as shown by $(aug(\mathtt{P}), \sigma'', \delta'') \rightarrow^* (\mathtt{skip}, \sigma', \delta')$ above. Using the rule Seq (premise to conclusion) repeatedly we get

$$(inv(\mathtt{P'}); i(\mathtt{S}), \sigma', \delta') \rightarrow^* (\mathtt{skip}; i(\mathtt{S}), \sigma'', \delta'') \rightarrow (i(\mathtt{S}), \sigma'', \delta'')$$

Therefore $\sigma^\dagger = \sigma''$ and $\delta^\dagger = \delta''$. All that remains now is to prove $(i(\mathtt{S}), \sigma'', \delta'') \rightarrow^* (\mathtt{skip}, \sigma, \delta)$, which is done in Lemma 2 below. □

Lemma 2. *Let* S *be a statement that does not interact with the auxiliary store,* σ *be an arbitrary initial data store and* δ *be an arbitrary initial auxiliary data store. Then if* $(\mathtt{S}, \sigma, \delta) \rightarrow^* (\mathtt{skip}, \sigma', \delta)$ *for some* σ'*, then* $(i(\mathtt{S}), \sigma', \delta') \rightarrow^* (\mathtt{skip}, \sigma, \delta)$ *for some* δ'*.*

Proof. We consider each type of statement S in turn. Due to space constraints, we only include one case, with the other cases following similarly.

Case 2.1. Consider statement X=a and its execution under the initial stores σ and δ where X is initially v′ and a evaluates to v in l steps such that

$$(\mathtt{X=a}, \sigma, \delta) \xrightarrow[\text{DA2}]{l} (\mathtt{X=v}, \sigma, \delta) \xrightarrow[\text{DA1}]{} (\mathtt{skip}, \sigma[\mathtt{X} \mapsto \mathtt{v}], \delta)$$

Then by Proposition 1 and Lemma 1 Case 1.1, we have that $(a(\mathtt{X=a}),\sigma,\delta) \rightarrow \ldots \rightarrow (\mathtt{skip}, \sigma', \delta')$, such that $\sigma' = \sigma[\mathtt{X} \mapsto \mathtt{v}]$ and $\delta' = \delta[\mathtt{v'} \mapsto \mathtt{X}]$. Then the execution of the inverted version of X=a is

$$(i(\mathtt{X=a}), \sigma', \delta') \xrightarrow[\text{Pop}]{} (\mathtt{X=v'}, \sigma', \delta'[\mathtt{X}]) \xrightarrow[\text{DA1}]{} (\mathtt{skip}, \sigma'[\mathtt{X} \mapsto \mathtt{v'}]), \delta'[\mathtt{X}])$$

such that $\sigma'[\mathtt{X} \mapsto \mathtt{v'}] = \sigma$ since v′ is equal to the initial value of X retrieved by the pop operation, and $\delta'[\mathtt{X}] = \delta$. Therefore the stores have been restored to their initial states, as required, meaning the case holds.

With all other cases following in a similar manner, the Lemma is proved to be correct. □

```
X+=Y+2 par (Y=X+2; X=4)
```

Figure 4: Original program

```
X+=Y+2[] par (Y=X+2[]; X=4[])
```

Figure 5: Annotated program

6 Adding Parallelism

We will now modifiy our first approach to support non-communicating parallelism [9], also referred to as *interleaving*, where the execution of two (or more) programs are interleaved while each individually maintains program order. To the best of our knowledge, RCC does not support parallelism in any form. Due to space constraints, we restrict the language to assignments and parallelism only. Conditionals and loops can be modified in a similar way and so are omitted. Let us reuse previous notation such that $\mathbb{P}$ is now the set of programs of this restricted language, $\hat{\mathbb{P}}$ is now the set of annotated programs, $\mathbb{P}^{-1}$ is now the set of inverted programs and $\mathbb{S}$ is now the set of statements of this restricted language. The definition of a statement becomes

$$\texttt{S} ::= \texttt{skip} \mid \texttt{X = Exp} \mid \texttt{X Cop Exp} \mid \texttt{P par P}$$

6.1 Challenges

Supporting parallelism introduces three challenges. Execution of parallel programs results in a non-deterministic execution order. Our first approach works as the programs are sequential, allowing the inverted program to follow the *inverted program order*. However there may be different execution orders of the same parallel program due to interleaving. So, without care, programs can be executed forwards under one interleaving and reversed under another, which is clearly incorrect. Consider the program in Figure 4 represented via (`P1 par (Q1;Q2)`), where the three possible execution interleavings are (`P1;Q1;Q2`), (`Q1;P1;Q2`) or (`Q1;Q2;P1`). Imagine the program here is executed under the first interleaving with the initial data store σ where `X=1` and `Y=1`, resulting in the final state σ' where `X=4` and `Y=6`. Without further information, inversion may assume the third interleaving was executed and so inverts the statements in the order (`P1;Q2;Q1`), clearly producing the incorrect final state where `X=4` and `Y=1`. We therefore will require both the auxiliary store δ and the inverse program as before, as well as the order in which the statements were executed forwards, termed *interleaving order*.

Another challenge is the *atomicity of statements*. Execution of a statement typically takes several steps to complete, increasing the possible execution paths and likelihood of races. Consider Figure 4 with no assumption of atomicity and initial state σ as above. Imagine `Y` is first evaluated in `P1`, then all of `Q1` and `Q2` are executed, before `P1` finally completes. This leads to the final state σ' where `X=7` and `Y=3`, values not reachable when assuming statement atomicity.

Finally, `push` operations inserted in our first approach relate to a specific statement and these must be executed atomically. Consider the program `X+=1 par X=5` augmented via our first approach into `X+=1 par push(`σ`(X),`δ`(X)); X=5`, with no such atomicity and `X` initially `1`. Assume that the `push` statement executes first, storing the value `1` onto δ`(X)`. The constructive assignment is then executed, incrementing `X` by one to `2`. Finally the destructive assignment is executed, updating `X` to `5`. The inverted version of this program is `X-=1 par X=pop(`δ`(X)`. Reversing the same interleaving first inverts the destructive assignment, assigning the value `1` from δ`(X)` to the variable `X`. The constructive assignment is then inverted, decrementing `X` by one to `0`. Clearly, this reversal has been unsuccessful.

6.2 Overcoming these Challenges

We update our approach to now capture the interleaving order. An identifier, or element of the set of natural numbers used in ascending order, is associated with each statement, each time it is executed, and stored onto a stack within the source code, very much like the *communication keys* of CCSK [13, 14]. These identifiers index any reversal information stored, and are used to direct the execution of the inverted version. This makes the execution order deterministic, thus removing the first challenge.

The updated approach will not introduce `push` statements in order to avoid issues relating to statement atomicity. A combination of this, and the fact that the interleaving order is not determined until runtime, mean all state-saving will be deferred to the operational semantics. A separate set of operational semantics are defined for forward execution. As such, inversion will no longer introduce `pop` statements, with all interaction with the reversal information being deferred to another separate set of operational semantics. In future work, we will add support for conditionals and loops, and further extend this with locks and mutual exclusion to allow this approach to be implemented within the language syntax.

We make the assumption of the atomicity of statements, restricting all interleaving to statement level, though this will be removed in future work.

6.3 Annotation and Forward Execution

The process of augmentation is replaced with *annotation*. This takes the original program, appends the necessary stacks into the program statements' source code, before returning the annotated version. Each statement is associated with a stack, necessary for programs containing loops as each statement may be executed multiple times requiring multiple identifiers. Stacks are not strictly necessary for our restricted language, however will be vital when we introduce conditionals and loops and so are included here for continuity. Each of these stacks is initially empty, and named uniquely via the function `nextS`. Let $\mathbb{S}'$ be the set of annotated statements. The function $ann : \mathbb{P} \to \hat{\mathbb{P}}$ takes the original program and recursively applies the re-defined function $a : \mathbb{S} \to \mathbb{S}'$ to each statement, producing the annotated version, where `e` is an arithmetic expression.

$$\begin{aligned} ann(\varepsilon) &= \varepsilon\ \texttt{A} & ann(\texttt{S;P}) &= a(\texttt{S}); ann(\texttt{P}) \\ a(\texttt{skip}) &= \texttt{skip A} & a(\texttt{X = e}) &= \texttt{X = e A} \\ a(\texttt{X cop e}) &= \texttt{X cop e A} & a(\texttt{P par Q}) &= ann(\texttt{P})\ \texttt{par}\ ann(\texttt{Q}) \end{aligned}$$

At this point, we have introduced a new syntactic category for annotated programs. Annotated programs and statements are defined below, with arithmetic and Boolean expressions as in Section 2. The sets $\mathbb{P}$ and $\mathbb{S}$ are also extended accordingly.

$$\begin{aligned} &\texttt{AP} ::= \varepsilon\ \texttt{A} \mid \texttt{AS; AP} \\ &\texttt{AS} ::= \texttt{skip A} \mid \texttt{X = Exp A} \mid \texttt{X Cop Exp A} \mid \texttt{AP par AP} \end{aligned}$$

Consider Figure 4. Applying the function *ann* to this program produces the annotated version in Figure 5, where each statement now has an empty stack.

As mentioned previously, annotation does not handle state-saving with this now implemented within the operational semantics. Each time a statement execution completes, a unique identifier is added to that statement's source code stack. To synchronise the use of these identifiers, the next available identifier is retrieved through the function `next()`. To avoid further data races, there is mutual exclusion on the use of this atomic function between the parallel programs. This function, typically used as `m = next()`,

```
X+=Y+2[1] par (Y=X+2[2]; X=4[3])
```

Figure 6: Final Annotated program

```
X−=Y+2[1] par (X=4[3]; Y=X+2[2])
```

Figure 7: Inverted program

assigns the value of the next available identifier to `m`, while incrementing the value it will return next time by one. Identifiers will index reversal information, meaning the stacks within the auxiliary store now consist of elements of the form (`i`,`v`), where `i` is an identifier and `v` is a value. The following operational semantics defines the forwards execution, where $f(\mathtt{A})$ indicates an update of the source code stack `A`.

$$\text{[Skip]}\quad \frac{}{(\mathtt{skip\ A;P},\sigma,\delta)\rightarrow(\mathtt{P},\sigma,\delta)} \qquad \text{[Seq1]}\quad \frac{(\mathtt{S\ A},\sigma,\delta)\rightarrow(\mathtt{S'}\ \ f(\mathtt{A}),\sigma',\delta')}{(\mathtt{S\ A;P},\sigma,\delta)\rightarrow(\mathtt{S'}\ \ f(\mathtt{A});\mathtt{P},\sigma',\delta')}$$

$$\text{[DA1]}\quad \frac{}{(\mathtt{X\ =\ v\ A},\sigma,\delta)\rightarrow(\mathtt{skip\ m{:}A},\sigma[\mathtt{X}\mapsto\mathtt{v}],\delta[(\mathtt{m},\sigma(\mathtt{X}))\mapsto\mathtt{X}])}\ \text{where } \mathtt{m}=\mathtt{next()}$$

$$\text{[CA1]}\quad \frac{}{(\mathtt{X\ cop\ v\ A},\sigma,\delta)\rightarrow(\mathtt{skip\ m{:}A},\sigma[\mathtt{X}\mapsto\sigma(\mathtt{X})\ \mathtt{op\ v}],\delta)}\ \text{where } \mathtt{m}=\mathtt{next()}$$

$$\text{[DA2]}\quad \frac{(\mathtt{e},\sigma,\delta)\rightarrow(\mathtt{e'},\sigma',\delta')}{(\mathtt{X\ =\ e\ A},\sigma,\delta)\rightarrow(\mathtt{X\ =\ e'\ A},\sigma',\delta')} \qquad \text{[CA2]}\quad \frac{(\mathtt{e},\sigma,\delta)\rightarrow(\mathtt{e'},\sigma',\delta')}{(\mathtt{X\ cop\ e\ A},\sigma,\delta)\rightarrow(\mathtt{X\ cop\ e'\ A},\sigma',\delta')}$$

$$\text{[P1]}\quad \frac{}{(\mathtt{P\ par\ skip},\sigma,\delta)\rightarrow(\mathtt{P},\sigma,\delta)} \qquad \text{[P2]}\quad \frac{}{(\mathtt{skip\ par\ Q},\sigma,\delta)\rightarrow(\mathtt{Q},\sigma,\delta)}$$

$$\text{[P3]}\quad \frac{(\mathtt{P},\sigma,\delta)\rightarrow(\mathtt{P'},\sigma',\delta')}{(\mathtt{P\ par\ Q},\sigma,\delta)\rightarrow(\mathtt{P'\ par\ Q},\sigma',\delta')} \qquad \text{[P4]}\quad \frac{(\mathtt{Q},\sigma,\delta)\rightarrow(\mathtt{Q'},\sigma',\delta')}{(\mathtt{P\ par\ Q},\sigma,\delta)\rightarrow(\mathtt{P\ par\ Q'},\sigma',\delta')}$$

Sequential composition is handled similarly to our first approach, with the exception that the source code stacks are present and potentially modified. The expressions within assignments are handled either by [DA2] or [CA2] respectively, with no identifier association due to our assumption of the atomicity of statements. When the execution of a destructive assignment completes, the rule [DA1] associates a new identifier `m` within the source code stack `A`, and uses it to index the old value $\sigma(\mathtt{X})$ stored on δ. Constructive assignments complete via the rule [CA1], where an identifier is associated but no reversal information is stored. Parallel composition executes as expected, with either program able to make a step of execution, until one side is complete meaning the statement becomes sequential.

After the execution of the annotated program under these semantics, the *final* annotated version with populated stacks is produced. Linking again to our example with the execution interleaving (`P1;Q1;Q2`), initial state σ with values `X=1` and `Y=1` and initial auxiliary store δ, the final annotated version is shown in Figure 6. The program state σ' after this execution has the values `X=4` and `Y=6`, while δ' contains the necessary reversal information.

6.4 Inversion and Reverse Execution

Inversion now takes the *final annotated program* and produces a relatively similar inverted version. This contains all statements of the given program in its inverted program order, with the inverted version of all constructive assignments. Due to the similarity between annotated and inverted versions, we now let $\mathbb{S}'$ be the set of both annotated and inverted statements of this approach. The function $inv : \hat{\mathbb{P}} \rightarrow \mathbb{P}^{-1}$ recursively applies the re-defined function $i : \mathbb{S}' \rightarrow \mathbb{S}'$ to each statement in reverse order. Both *inv* and *i*

are now given, with icop as defined in Section 5.

$$\begin{aligned} inv(\varepsilon\ \texttt{A}) &= \varepsilon\ \texttt{A} & inv(\texttt{AS;AP}) &= inv(\texttt{AP});i(\texttt{AS}) \\ i(\texttt{skip A}) &= \texttt{skip A} & i(\texttt{X = e A}) &= \texttt{X = e A} \\ i(\texttt{X cop e A}) &= \texttt{X icop e A} & i(\texttt{P par Q}) &= (inv(\texttt{P}))\ \texttt{par}\ (inv(\texttt{Q})) \end{aligned}$$

The inverted version does not make use of the reversal information, and instead must be executed under a separate set of operational semantics for reverse execution. These semantics are responsible for all interaction with any information saved, as well as using the identifiers to direct inversion along the correct interleaving order. This is implemented using the mutually exclusive and atomic function previous(), related to the function next() such that next() = previous() + 1. The statement m = previous() checks that the current value of m matches the current value of previous(), as well as decrementing the value the function will return next time by 1. The statement m == previous() again checks that m is equal to previous(), but does not decrement the value it will return next time. This forces all steps of the evaluation to happen sequentially, reflecting our assumption of statement atomicity. The functions previous() and next() are strongly related, meaning the execution of one must update the value of the other accordingly. Here the rules for sequential and parallel composition are similar to those in Section 6.3, but with the transition relation $\rightsquigarrow$ replacing $\rightarrow$, hence they are omitted to save space.

$$\text{[RDA]}\quad \frac{\texttt{A = m:A}' \quad \delta(\texttt{X}) = \texttt{(m,v):X}' \quad \texttt{m = previous()}}{(\texttt{X = e A},\sigma,\delta) \rightsquigarrow (\texttt{skip A}',\sigma[\texttt{X} \mapsto \texttt{v}],\delta[\texttt{X/X}'])}$$

$$\text{[RCA1]}\quad \frac{\texttt{A = m:A}' \quad \texttt{m = previous()}}{(\texttt{X cop v A},\sigma,\delta) \rightsquigarrow (\texttt{skip A}',\sigma[\texttt{X} \mapsto \sigma(\texttt{X})\ \texttt{op v}],\delta)}$$

$$\text{[RCA2]}\quad \frac{(\texttt{e},\sigma,\delta) \rightsquigarrow (\texttt{e}',\sigma',\delta') \quad \texttt{A = m:A}' \quad \texttt{m == previous()}}{(\texttt{X cop e A},\sigma,\delta) \rightsquigarrow (\texttt{X cop e}'\ \texttt{A},\sigma',\delta')}$$

Destructive assignments are handled via the single rule [RDA] as no evaluation of the expression e is required. A destructive assignment can be executed provided its most recent identifier matches both the current value of previous() and the index of the top element of its stack on δ. Provided these conditions hold, the variable is restored to its previous value retrieved from its stack on δ, before both of these stacks are popped. Constructive assignments require the two rules [RCA1] and [RCA2] as the expression must still be evaluated. Each step of the evaluation is executed sequentially by ensuring the identifiers match without removing them. Only when the assignment has executed will the identifiers be removed, restricting interleaving until this point.

Applying the function *inv* to the final annotated program in Figure 6 produces the inverted version in Figure 7. Execution of this inverted version under the reverse operational semantics starting with the state σ' with values X=4 and Y=6, results in the reverse statement order of Q2;Q1;P1, the state σ with values X=1 and Y=1 and the auxiliary store δ. Therefore the execution has been successfully reversed with all variables restored to their initial values.

6.5 Correctness

We now outline our correctness results for the second approach. Annotation of a program P assigns empty stacks to the statements of the program. During execution, these stacks are populated with identifiers. Let's denote such an *update* of the stacks of $ann(\texttt{P})$ as $\rho(ann(\texttt{P}))$. We now give propositions corresponding to those in Sections 4 and 5, however we defer all termination parts to future work. Proposition 3

shows that the behaviour of the original and annotated programs are semantically equivalent with respect to the data store σ, and that the annotated program will populate both the stacks within the source code and the auxiliary store.

Proposition 3. *Let* P *be a program and ann*(P) = P′. *If* $(\mathtt{P},\sigma,\delta) \rightarrow^* (\mathtt{skip},\sigma',\delta)$ *for some* σ', *then* $(\mathtt{P}',\sigma,\delta) \rightarrow^* (\mathtt{skip\ C},\sigma',\delta')$ *for some* C *and* δ' *and the computation* $(\mathtt{P}',\sigma,\delta) \rightarrow^* (\mathtt{skip\ C},\sigma',\delta')$ *produces an update* $\rho(\mathtt{P}')$ *for some* ρ.

Proposition 4 shows that executing the inverted program under the two stores and the updated source code stacks does indeed reverse all components to their initial values, as well as using the identifiers stored in the code to direct the execution.

Proposition 4. *Let* P *be a program and ann*(P) = P′. *If* $(\mathtt{P},\sigma,\delta) \rightarrow^* (\mathtt{skip},\sigma',\delta)$ *for some* σ', *then* $(inv(\rho(\mathtt{P}')),\sigma',\delta') \rightsquigarrow^* (\mathtt{skip\ C},\sigma,\delta)$ *for some* C, δ' *and* ρ.

At least two additional lemmas are used throughout the proofs of the two propositions above. These correspond to the lemmas used in Sections 4 and 5, and are listed below.

Lemma 3. *Let* S *be a program statement and ann*(S) = S A *for some* A. *If* $(\mathtt{S},\sigma,\delta) \rightarrow^* (\mathtt{skip},\sigma',\delta)$ *for some* σ', *then* $(\mathtt{S\ A},\sigma,\delta) \rightarrow^* (\mathtt{skip\ A}',\sigma',\delta')$ *for some* A′ *and* δ'.

Lemma 4. *Let* S *be a program statement and ann*(S) = S A *for some* A. *If* $(\mathtt{S},\sigma,\delta) \rightarrow^* (\mathtt{skip},\sigma',\delta)$ *for some* σ', *then* $(inv(\mathtt{S\ A}'),\sigma',\delta') \rightsquigarrow^* (\mathtt{skip\ C},\sigma,\delta)$ *for some* A′, δ' *and* C.

7 Conclusion

We have presented an approach to reversing an imperative programming language, using the state-saving notion. We have defined two functions, namely *aug* and *inv*, capable of producing the augmented version and inverted version of an originally irreversible program, respectively. We have proved that our augmentation does not alter the behaviour of the program with respect to the data store, and that it saves the necessary information to revert the program state after execution to that of before. The auxiliary store used to save this reversal information is also proved to revert to its initial state, ensuring no extra garbage data is produced.

We also described a modification to our first approach to include parallelism within a restricted language, while avoiding a number of issues parallelism introduces. We defined a function *ann* and redefined *inv* to support the recording of the interleaving order into the source code. Two sets of operational semantics are defined, one performing the state-saving for forwards execution, and another performing the inversion for reverse execution. Finally, we propose the correctness results for this modified approach.

In the future, we shall relax the language restriction and support both conditional statements and while loops alongside parallel statements. The assumption of statement atomicity will be removed when considering a richer language which supports locks and mutual exclusion, allowing the approach described here to be implemented within the language itself. We will continue to extend the approach towards the complexity of C.

Acknowledgements

We are grateful to the referees for their detailed and helpful comments and suggestions. The authors acknowledge partial support of COST Action IC1405 on Reversible Computation - extending horizons of computing. The second author acknowledges the support by the University of Leicester in granting him Academic Study Leave, and thanks Nagoya University for support during the study leave. The third author acknowledges the support by JSPS KAKENHI grants JP17H0722 and JP17K19969.

References

[1] H. Agrawal, R. A. DeMillo & E. H. Spafford (1991): *An Execution-Backtracking Approach to Debugging*. *IEEE Software* 8(3), pp. 21–26, doi:10.1109/52.88940.

[2] B. Biswas and R. Mall (1999): *Reverse Execution of Programs*. *SIGPLAN Notices* 34(4), pp. 61–69, doi:10.1145/312009.312079.

[3] C. D. Carothers, K. S. Perumalla & R. Fujimoto (1999): *Efficient Optimistic Parallel Simulations using Reverse Computation*. *ACM Transactions on Modelling and Computer Simulation* 9(3), pp. 224–253, doi:10.1145/347823.347828.

[4] D. Cingolani, M. Ianni, A. Pellegrini & F. Quaglia: *Mixing Hardware and Software Reversibility for Speculative Parallel Discrete Event Simulation*. In: *RC 2016*, *LNCS* 9720, Springer, doi:10.1007/978-3-319-40578-0_9.

[5] R. Fujimoto (1990): *Parallel Discrete Event Simulation*. *Communications of the ACM* 33(10), pp. 30–53, doi:10.1145/84537.84545.

[6] R. Glück & M. Kawabe (2004): *Derivation of Deterministic Inverse Programs Based on LR Parsing*. In: *FLOPS 2004*, *LNCS* 2998, Springer, pp. 291–306, doi:10.1007/978-3-540-24754-8_21.

[7] R. Glück & M. Kawabe (2005): *Revisiting an Automatic Program Inverter for LISP*. *SIGPLAN Notices* 40(5), pp. 8–17, doi:10.1145/1071221.1071222.

[8] D. Gries (1981): *The Science of Programming*. Springer, doi:10.1007/978-1-4612-5983-1.

[9] H. Hüttel (2010): *Transitions and Trees - An Introduction to Structural Operational Semantics*. Cambridge University Press, doi:10.1017/CBO9780511840449.

[10] R. Landauer (1961): *Irreversibility and Heat Generation in the Computing Process*. *IBM Journal of Research and Development* 5(3), pp. 183–191, doi:10.1147/rd.53.0183.

[11] C. Lutz (1986): *Janus: A Time-Reversible Language. A Letter to Dr. Landauer*. `http://tetsuo.jp/ref/janus.pdf`.

[12] K. Perumalla (2014): *Introduction to Reversible Computing*. CRC Press.

[13] I. C. C. Phillips & I. Ulidowski (2007): *Reversing Algebraic Process Calculi*. *J. Log. Algebr. Program.* 73(1-2), pp. 70–96, doi:10.1016/j.jlap.2006.11.002.

[14] I. C. C. Phillips, I. Ulidowski & S. Yuen (2012): *A Reversible Process Calculus and the Modelling of the ERK Signalling Pathway*. In: *RC2012*, *LNCS* 7581, Springer, pp. 218–232, doi:10.1007/978-3-642-36315-3_18.

[15] M. Schordan, D. R. Jefferson, P. D. Barnes Jr., T. Oppelstrup & D. J. Quinlan (2015): *Reverse Code Generation for Parallel Discrete Event Simulation*. In: *RC 2015*, *LNCS* 9138, Springer, pp. 95–110, doi:10.1007/978-3-319-20860-2_6.

[16] M. Schordan, T. Oppelstrup, D. Jefferson, P. D. Barnes Jr. & D. J. Quinlan (2016): *Automatic Generation of Reversible C++ Code and Its Performance in a Scalable Kinetic Monte-Carlo Application*. In: *SIGSIM-PADS 2016*, ACM, pp. 111–122, doi:10.1145/2901378.2901394.

[17] G. Vulov, C. Hou, R. W. Vuduc, R. Fujimoto, D. J. Quinlan & D. R. Jefferson (2011): *The Backstroke Framework for Source Level Reverse Computation Applied to Parallel Discrete Event Simulation*. In: *WSC 2011*, WSC, doi:10.1109/WSC.2011.6147998.

[18] T. Yokoyama, H. B. Axelsen & R. Glück (2008): *Principles of a Reversible Programming Language*. In: *Proceedings of the 5th Annual Conference on Computing Frontiers, 2008*, ACM, pp. 43–54, doi:10.1145/1366230.1366239.

[19] T. Yokoyama & R. Glück (2007): *A Reversible Programming Language and its Invertible Self-interpreter*. In: *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, ACM, pp. 144–153, doi:10.1145/1244381.1244404.

Using Session Types for Reasoning About Boundedness in the π-Calculus

Hans Hüttel*
Department of Computer Science, Aalborg University, Denmark

The classes of depth-bounded and name-bounded processes are fragments of the π-calculus for which some of the decision problems that are undecidable for the full calculus become decidable. P is depth-bounded at level k if every reduction sequence for P contains successor processes with at most k active nested restrictions. P is name-bounded at level k if every reduction sequence for P contains successor processes with at most k active bound names. Membership of these classes of processes is undecidable. In this paper we use binary session types to decise two type systems that give a sound characterization of the properties: If a process is well-typed in our first system, it is depth-bounded. If a process is well-typed in our second, more restrictive type system, it will also be name-bounded.

1 Introduction

In the π-calculus, the notion of name restriction is particularly important. The study of properties of name binding is a testbed for studying properties of bindable entities and notions of scoping in programming languages. In a restriction process $(\nu x)P$ the name x has P as its scope and it is customary to think of x as a new name, known only to P. It is the interplay between restriction and replication (or recursion) that leads to the π-calculus being Turing-powerful. Without either of these two constructs, this is no longer the case [9].

With full Turing power comes undecidability of commonly encountered decision problems such as the termination problem "Given process P, will P terminate?" and the coverability problem "Given process P and process Q, is there a computation of P that will eventually reach a process that has Q as a subprocess?". Several classes of processes have been identified for which (some of) these problems remain decidable. Examples are the *finitary processes* without replication or recursion, the *finite-control processes* [3] in which every process has a uniform bound on the number of parallel components in any computation, the *bounded processes* [2] for which there are only finitely many successors of any reduction up to a special notion of structural congruence with permutation over a finite set of names, and *processes with unique receiver and bounded input* [1].

More recently, there has been work in this area that studies limitations on the use of restriction that will ensure decidability. The notion of *depth-bounded* processes was introduced by Meyer in [11]. A process P is depth-bounded at level k if there is an upper bound k, such that any reduction sequence for P will only lead to successor processes that have at most k active nested restrictions – that is, restrictions not occurring underneath some prefix. Termination and coverability are both decidable for depth-bounded processes. The class of depth-bounded processes is expressive and contains a variety of other decidable subsets of the π-calculus. Moreover, for any fixed k it is decidable if a process P is depth-bounded at level k; however, it is undecidable if there exists a k for which P is depth-bounded [11].

*E-mail: hans@cs.aau.dk

K. Peters and S. Tini (Eds.): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics (EXPRESS/SOS 2017).
EPTCS 255, 2017, pp. 67–82, doi:10.4204/EPTCS.255.5

In a more recent paper [4], D'Osualdo and Ong have introduced a type system that gives a sound characterization of depth-boundedness: If P is well-typed, then P is depth-bounded. The underlying idea of this type system is to analyze properties of the hierarchy of restrictions within a process.

Another class of π-calculus processes is that of *name-bounded* processes, introduced by Hüchting et al. [8]. A process P is name-bounded at level k if any reduction sequence for P will only lead to successor processes with at most k active bound names.

The goal of this paper is to use binary session types [7] to give sound characterizations of depth-boundedness, respectively name-boundedness in the π-calculus: If a process is well-typed, we know that it is depth-bounded, respectively name-bounded. The advantages of this approach are the following: Firstly, unlike the type system proposed by D'Osualdo and Ong [4] we can directly keep track of how names are used and where they appear in a process, since this is central to session type disciplines. The linear nature of session names ensures that every name of this kind will always, when used, occur in precisely two parallel components. Secondly, the session type disciplines are resource-conscious; we can therefore ensure that new bound names are only introduced whenever existing bound names can no longer be used. Both type systems use finite session types to achieve this for recursive processes. Informally, a new recursive call can only occur once all sessions involving the bound names of the current recursive call have been used up. In the proof of the soundness of the system for characterizing name-boundedness system, we make use of the fact that it is a more restrictive version of that for depth-boundedness.

The rest of our paper is organized as follows. Section 2 describes the π-calculus that we will consider; section 3 introduces the notions of boundedness. Section 4 presents a type system for depth-bounded processes, which is analyzed in sections 5 and 6. Section 7 presents a type system for name-bounded processes. Section 8 discusses the relationship with other classes of processes.

2 A typed π-calculus with recursion

We follow Meyer [11] and use a π-calculus with *recursion* instead of replication. The reason behind this choice of syntax is that we would like infinite behaviours to make use of bound names in a non-trivial manner that guarantees boundedness properties. In general, the combination of restriction and replication in $!(\nu x)P$ will result in a process that fails to be name-bounded.

2.1 Syntax

We assume the existence of a countably infinite set of names, $\mathscr{N}$, and let $a,b,\ldots$ and $x,y,\ldots$ range over $\mathscr{N}$. Moreover, we assume a countably infinite set of recursion variables, $\mathscr{R}$, and let $X,Y,\ldots$ range over $\mathscr{R}$.

2.1.1 Processes

Following [5] we will use a version of the π-calculus with *polarized names* in order to ensure that the endpoints of a channel will not end up in the same parallel component. We assume polarities ranged over by $p,q\ldots$. The polarities $+$ and $-$ are dual; we define $\overline{+}=-$ and $\overline{-}=+$. The empty polarity ε is self-dual and used for names used as channels that are not session channels and to tag name occurrences in the binding constructs of input and restriction. We call the set of polarized names $\mathscr{N}_{\text{pol}}$.

The formation rules of processes are given by

$$P ::= x^p(y).P_1 \mid \overline{x^p}\langle y^q\rangle.P_1 \mid P_1 \mid P_2 \mid (\nu x : T)P_1 \mid \mu X.P_1 \mid X \mid \mathbf{0}$$
$$p ::= + \mid - \mid \varepsilon$$

As usual, $x^p(y).P_1$ denotes a process that inputs a name on channel x and continues as P_1; the unpolarized name y is bound in P_1. $\overline{x^p}\langle y^q\rangle.P_1$ is a process that outputs the polarized name y^q on channel x and continues as P_1. $P_1 \mid P_2$ is the parallel execution of P_1 and P_2. $\mu X.P_1$ is a recursive process with body P_1. We assume that every such recursive process is *guarded*; every occurrence of a recursion variable must be found underneath an input or an output prefix. In $\mu X.P_1$ the μX is called a binding occurrence of X. A process P is *recursion-closed* if every recursion variable X in P has a binding occurrence for some subprocess $\mu X.P_1$ and if all recursion variables are distinct. We employ a notion of typed restriction, which we will now explain.

2.1.2 Typed restrictions

In the restriction $(\nu x : T)P_1$ the unpolarized name x is bound in P_1 and annotated with type T. Our set of types $\mathscr{T}$ is a non-recursive version of the binary session types introduced by Gay and Hole [5] and defined by the formation rules

$$T ::= S \mid \mathsf{Ch}(T)$$
$$S ::= (S_1, S_2) \mid !T.S \mid ?T.S \mid \mathsf{end}$$

A type T can be a *linear* endpoint type S or pair of endpoints (S_1, S_2), or an *unlimited* channel type $\mathsf{Ch}(T)$. An endpoint type S of the form $!T.S$ denotes that a channel of this type can output a name of type T; afterwards, the channel will have type S. An endpoint type of the form $?T.S$ denotes that a channel of this type can input a name of type T; afterwards, the channel will have type S. The special endpoint type end is the type of an endpoint that allows no further communication. If $T = (!T_1.S_2, ?T_1'.S_2')$ we let $T \downarrow = (S_2, S_2')$; this denotes the successor of a pair of endpoint types. If $T = \mathsf{Ch}(T_1)$, then $T \downarrow = T$.

We use the type annotation of restrictions to keep track of the subject name that led to a reduction and of how the types of bound names evolve.

The sets of free and bound names of a process, $\mathrm{fn}(P)$ and $\mathrm{bn}(P)$, are defined as usual. To simplify the presentation, we assume all free and bound names distinct. We let $P\{^y/_x\}$ denote the capture-avoiding substitution that replaces all free occurrences of x in P by y. A name $n \in \mathrm{bn}(P)$ is *active* if it does not appear underneath a prefix.

2.1.3 Structural congruence

Structural congruence is the least congruence relation for the process constructs that is closed under the axioms in Table 1.

Following Meyer [11], we sometimes consider processes in *restricted form*. A process is in inner normal form, if every restriction $(\nu x : T)$ only encloses parallel components that contain x. A process is in outer normal form if every restriction not underneath a prefix appears at the outermost level.

Definition 1 (Normal forms)**.** Let P be a process.

- P is in *inner normal form* if for every subprocess $(\nu x : T)(P_1 \mid \cdots \mid P_k)$ where none of the P_i are parallel compositions of processes, we have $x \in \mathrm{fn}(P_i)$ for all $1 \leq i \leq k$.

(NEW-1)	$(\nu x:T)(\nu y:T')P \equiv (\nu y:T')(\nu x:T)P$	(NIL-1)	$P \mid \mathbf{0} \equiv P$
(NEW-2)	$(\nu x:T)P \mid Q \equiv (\nu x:T)(P \mid Q)$ if $x \notin \mathrm{fn}(Q)$	(NIL-2)	$(\nu x:T)\mathbf{0} \equiv \mathbf{0}$
(PAR-1)	$P \mid Q \equiv Q \mid P$		
(PAR-2)	$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$		

Table 1: Structural congruence: Axioms and rules

$$\text{(COM-ANNOT)} \qquad a^p(x).P_1 \mid \overline{a^{\overline{p}}}\langle y^q\rangle.P_2 \xrightarrow{\{a\}} P_1\{^{y^q}/_x\} \mid P_2$$

$$\text{(PAR-ANNOT)} \qquad \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}$$

$$\text{(NEW-ANNOT)} \qquad \frac{P \xrightarrow{\alpha} P'}{(\nu x:T)P \xrightarrow{\alpha} (\nu x:T')P'} \quad \text{where} \quad \begin{array}{l} T = T' \text{ if } x \notin \alpha \\ T' = T\downarrow \text{ if } x \in \alpha \end{array}$$

$$\text{(UNFOLD-ANNOT)} \qquad \frac{P > Q \quad Q \xrightarrow{\alpha} P'}{P \xrightarrow{\{\mathsf{rec}\}\cup\alpha} P'}$$

$$\text{(STRUCT-ANNOT)} \qquad \frac{P \equiv Q \quad Q \xrightarrow{\alpha} Q' \quad Q' \equiv P'}{P \xrightarrow{\alpha} P'}$$

Table 2: Annotated reduction rules

- P is in *outer normal form* if $P = (\nu x_1)\ldots(\nu x_k)P_1$ such that $x_i \in \mathrm{fn}(P_1)$ for all $1 \leq i \leq k$ and such that all restrictions in P_1 appear underneath prefixes.

Proposition 1. *For every process P we can construct a process $P_1 \equiv P$ in inner normal form and a process $P_2 \equiv P$ in outer normal form.*

2.2 An annotated reduction semantics

We define the behaviour of processes by an annotated reduction semantics that keeps track of when recursive unfoldings are necessary. Reductions are of the form $P \xrightarrow{\alpha} P'$ where either $\alpha = \{a\}, a \in \mathscr{N}$ or $\alpha = \{\mathsf{rec}, a\}$ for $a \in \mathscr{N}$. The latter annotation indicates that recursive unfolding was necessary to obtain the reduction. We define $\mathrm{n}(\{a\}) = a$ and $\mathrm{n}(\{\mathsf{rec}, a\}) = a$. The reduction rules are found in Table 2. Note that in the rule (NEW-ANNOT) the type associated with the bound name x evolves, if x if x is responsible for the communication and T is a session type.

If P reduces to P' in zero or more reduction steps, we write $P \rightarrow^* P'$.

Recursion is described by an unfolding relation which we define in Table 3. In the definition, we use the notion of *unfolding contexts*. An unfolding context $C[\,]$ is an incomplete process terms whose hole indicates where a prefix that participates in a reduction step appears as the direct result of unfolding a

recursive process.

Definition 2 (Unfolding contexts)**.** The set of unfolding contexts is given by the formation rules

$$C ::= [\,] \mid P \mid (\nu x : T)C$$

$$\text{(UNFOLD)} \quad \mu X.P > P[\mu X.P/X] \qquad\qquad \text{(CONTEXT)} \quad \frac{P > P'}{C[P] > C[P']}$$

Table 3: The rules for unfolding

Example 1. We can write the process

$$P \stackrel{\text{def}}{=} (\nu c : T)\mu X.a(x).\overline{x}\langle x\rangle.X \mid \mu Y.(\nu b : U)\overline{a}\langle b\rangle.x(y).Y$$

as

$$C_1[\mu X.a(x).\overline{x}\langle x\rangle.X] \text{ where } C_1 = [(\nu c : T)[] \mid \mu Y.(\nu b : U)\overline{a}\langle b\rangle.x(y).Y]$$

or

$$C_2[\mu Y.(\nu b : U)\overline{a}\langle b\rangle.x(y).Y] \text{ where } C_2 = (\nu c : T)\mu X.a(x).\overline{x}\langle b\rangle.X \mid [].$$

3 Notions of boundedness

Meyer introduces three notions of boundedness [11] for the π-calculus, and we now introduce them.

Depth-bounded processes A process P is depth-bounded if every configuration reachable from it can be rewritten so as to have no more than k nested restrictions. To define this, we first introduce a function $\mathsf{nest}(P)$ that counts the maximal number of active nested restrictions. A restriction is active if it does not occur underneath a prefix – this is similar to [4].

Definition 3. The nest function is defined by the clauses

$$\mathsf{nest}(\mathbf{0}) = 0 \qquad \mathsf{nest}(X) = 0$$
$$\mathsf{nest}((\nu x : T)P) = 1 + \mathsf{nest}(P) \qquad \mathsf{nest}(P_1 \mid P_2) = \max(\mathsf{nest}(P_1), \mathsf{nest}(P_2))$$
$$\mathsf{nest}(\mu X.P_1) = \mathsf{nest}(P_1) \qquad \mathsf{nest}(x^p(y).P_1) = \mathsf{nest}(\overline{x^p}\langle y^q\rangle.P_1) = 0$$

The restriction depth of a process is then the minimal nesting depth up to structural congruence.

Definition 4. The depth of a process P is given by

$$\mathsf{depth}(P) = \min\{\mathsf{nest}(Q) \mid Q \equiv P\}.$$

We define a normalization ordering $\succ$ on processes that removes bound names not found in a process. It is generated by the axiom

$$(\nu x)P \succ P \quad \text{if } x \notin \mathrm{fn}(P)$$

and closed under structural congruence. A process P is *normalized* if it has no superfluous bound names, that is, if $P \not\succ$; we write $P \rightsquigarrow Q$ if $P \succ^* Q$ and Q is normalized. [1]

Definition 5 (Depth-bounded process)**.** A process P is depth-bounded if there is a $k \in \mathbb{N}$ such that for every P' where $P \rightarrow^* P'$ we have that for some P'' with $P'' \equiv P'$ we have $\mathsf{depth}(P') \leq k$.

[1] Note that $(\nu x : T)P \equiv P \quad \text{if } x \notin \mathrm{fn}(P)$ is a derived identity if we include the axiom $(\nu x)\mathbf{0} \equiv \mathbf{0}$.

Name-boundedness A process P is *name-bounded* if there exists a constant $k \in \mathbb{N}$ such that whenever $P \to^* P'$ and $P' \rightsquigarrow P''$, then P'' has at most k restrictions. It is obvious that every name-bounded process is also depth-bounded.

Example 2. The term

$$P_1 = \mu X.(\nu r_1)(\overline{r_1^+}\langle a\rangle.X \mid r_1^-(x).X)$$

is depth-bounded with $\mathsf{depth}(P_1) = 1$. The term

$$P_2 = \mu X.(\nu r_1)(\nu r_2)(\overline{r_1^+}\langle r_2\rangle.X \mid r_1^-(x).X \mid \overline{r_2^+}\langle r_1\rangle \mid r_2^-(x))$$

is depth-bounded with $\mathsf{depth}(P_2) = 2$. Neither P_1 nor P_2 is name-bounded.

Width-boundedness A third notion of boundedness is that of *width-boundedness*. A process P is width-bounded if there exists a constant $k \in \mathbb{N}$ such that whenever $P \to^* P'$ we have that every bound name in P' occurs in at most k parallel components. This coincides with the notion of *fencing* recently used by Lange et al. [10] introduced in their analysis of Go programs.

4 Using session types for depth-boundedness

We now present a session type system that gives a sound characterization of depth-boundedness. Our account of binary session types similar to that used by Gay and Hole [5].

4.1 Types and type environments

Our type judgements are of the form $\Gamma, \Delta \vdash P$, where Γ contains the type bindings of the free polarized names in P. A type judgment is to be read as stating that P is well-behaved using the type information found in the type environment Γ and the recursion environment Δ (explained in Section 4.2).

Definition 6. A type environment Γ is a partial function $\Gamma : \mathscr{N}_{\text{pol}} \rightharpoonup \mathscr{T}$ with finite support.

- Γ is *unlimited* if for every $x \in \mathrm{dom}(\Gamma)$ we have $\Gamma(x) = \mathsf{Ch}(T)$ for some T or $\Gamma(x) = \mathsf{end}$
- Γ is *linear* if for every $x \in \mathrm{dom}(\Gamma)$ we have that $\Gamma(x) \neq \mathsf{Ch}(T)$ for all T. We let Γ_{lin} denote the largest sub-environment of Γ that is linear.
- If for every $x \in \mathrm{dom}(\Gamma)$ we have that $\Gamma(x) = \mathsf{end}$ or $\Gamma(x) = (\mathsf{end}, \mathsf{end})$, we say that Γ is *terminal*.

We define duality of endpoint types in the usual way (note that duality is not defined for base types).

Definition 7 (Duality of endpoint types)**.** Duality of endpoint types is defined inductively by

$$\overline{!T.S} = ?T.\overline{S} \qquad \overline{?T.S} = !T.\overline{S} \qquad \overline{\mathsf{end}} = \mathsf{end}$$

A type $T = (S_1, S_2)$ is *balanced* if $S_1 = \overline{S_2}$. A type environment Γ is balanced if for all $x \in \mathrm{dom}(\Gamma)$ we have that $\Gamma(x)$ is a balanced type or a base type B.

Definition 8 (Depth of types)**.** The depth of an endpoint type S is denoted $d(S)$ and is defined inductively by

$$d(!T.S) = 1 + d(S) \qquad d(?T.S) = 1 + d(S) \qquad d(\mathsf{end}) = 0$$

For a type $T = (S_1, S_2)$ we let $d(T) = \max(S_1, S_2)$. For all other T, we define $d(T) = 0$.

Definition 9 (Addition of type environments)**.** Let Γ_1 and Γ_2 be type environments such that $\mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \emptyset$. Then $\Gamma_1 + \Gamma_2$ is the type environment Γ that satisfies

$$\Gamma(x) = \begin{cases} \Gamma_1(x) & \text{if } x \in \mathrm{dom}(\Gamma_1) \setminus \mathrm{dom}(\Gamma_2) \\ \Gamma_2(x) & \text{if } x \in \mathrm{dom}(\Gamma_2) \setminus \mathrm{dom}(\Gamma_1) \end{cases}$$

4.2 Recursion and recursion environments

In our type system, recursion variables are typed with type environments. A *recursion environment* Δ is a function that to each recursion variable X assigns a type environment Γ. The idea is that Γ will represent the names and associated types needed to type a process $\mu X.P$.

Definition 10. A recursion environment Δ is a partial function $\Delta : \mathscr{R} \rightharpoonup (\mathscr{N}_{\mathrm{pol}} \rightharpoonup \mathscr{T})$ with finite support. We let $\Delta_\emptyset$ denote the empty recursion environment.

Definition 11. Let Δ_1 and Δ_2 be recursion environments where for all $X \in \mathrm{dom}(\Delta_1) \cap \mathrm{dom}(\Delta_2)$ we have $\Delta_1(X) = \Delta_2(X)$. $\Delta_1 + \Delta_2$ is the recursion environment Δ satisfying

$$\Delta(X) = \begin{cases} \Delta_1(X) & \text{if } X \in \mathrm{dom}(\Delta_1) \setminus \mathrm{dom}(\Delta_2) \\ \Delta_2(X) & \text{if } X \in \mathrm{dom}(\Delta_2) \setminus \mathrm{dom}(\Delta_1) \\ \Delta_1(X) & \text{otherwise} \end{cases}$$

4.3 Type rules

The set of valid type judgments is defined by the rules in Table 4. The type rules differ from the rules from standard session type systems in their treatment of recursion in two ways.

The rule (VAR) ensures that a recursion variable X can only be well-typed for Γ and Δ if the type environment Γ_1 associated with X mentions all the names in Γ. Moreover, the rule requires that the linear part of the type environment must be *terminal* and that the linear names present when a recursion variable X is reached include the ones found in the type environment used to type the process $\mu X.P$. Therefore, when a recursion variable is reached and a recursive call is made, the restricted names in the unfolding will be new: the existing sessions have been "used up".

The rule (CHAN) ensures that channels that are not session channels can only be bound within a non-recursive process, as the recursion environment present must be $\Delta_\emptyset$. Therefore, names that are not session names cannot accumulate because of recursive calls and lead to an unbounded restriction depth.

The need for private names to be linear inside a recursive process arises because an unlimited channel can be exploited by a recursive process to introduce unbounded nesting, as the following example from [4] illustrates.

Example 3. Consider the following process that cannot be typed; we therefore leave out type annotations and polarities in its description. Let

$$P = (\nu s)(\nu n)(\nu v)(\nu a)(\overline{s}\langle a\rangle \mid \mu S.(s(x).(\nu b)((\overline{v}\langle b\rangle.\overline{n}\langle x\rangle \mid \overline{s}\langle b\rangle) \mid S)))$$

The process can evolve as follows.

$$P \rightarrow^* (\nu s)(\nu n)(\nu v)(\nu a)(P_1 \mid (\nu b)(\nu b')((\overline{v}\langle b\rangle.\overline{n}\langle a\rangle \mid \overline{v}\langle b'\rangle.\overline{n}\langle b\rangle \mid \overline{s}\langle b'\rangle)))$$

where $P_1 = \mu S.(s(x).(\nu b)((\overline{v}\langle b\rangle.\overline{n}\langle x\rangle \mid \overline{s}\langle b\rangle) \mid S))$ can introduce further nesting since the channel s will, when used together with recursion, be used with an arbitrary number of new names that cannot be eliminated.

(IN-1) $\dfrac{\Gamma, x^p : T_2, y : T_1, \Delta \vdash P}{\Gamma, x^p : ?T_1.T_2, \Delta \vdash x^p(y).P}$ where $T_1 \neq \mathsf{end}$

(IN-2) $\dfrac{\Gamma, x^p : \mathsf{Ch}(T_1), y : T_1, \Delta \vdash P}{\Gamma, x^p : \mathsf{Ch}(T_1), \Delta \vdash x(y).P}$ where $T_1 \neq \mathsf{end}$

(OUT-1) $\dfrac{\Gamma, x^p : T_2, \Delta \vdash P}{\Gamma, x^p : !T_1.T_2, y^q : T_1, \Delta \vdash \overline{x^p}\langle y^q \rangle.P}$ $T_1 \neq \mathsf{end}$

(PAR) $\dfrac{\Gamma_1, \Delta_1 \vdash P_1 \quad \Gamma_2, \Delta_2 \vdash P_2}{\Gamma_1 + \Gamma_2, \Delta_1 + \Delta_2 \vdash P_1 \mid P_2}$

(OUT-2) $\dfrac{\Gamma, x : \mathsf{Ch}(T_2), y^q : T_2, \Delta \vdash P}{\Gamma, x : \mathsf{Ch}(T_2), y^q : T_2, \Delta \vdash \overline{x}\langle y^q \rangle.P}$ where T_2 unlimited

(SESSION) $\dfrac{\Gamma, x^+ : S, x^- : \overline{S}, \Delta \vdash P}{\Gamma, \Delta \vdash (\nu x : (S, \overline{S}))P}$

(NIL) $\Gamma, \Delta \vdash \mathbf{0}$ Γ unlimited

(VAR) $\Gamma, \Delta \vdash X$ $\Delta(X) = \Gamma_1$, $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(\Gamma_1)$, Γ_{lin} is terminal

(REC) $\dfrac{\Gamma, \Delta, X : \Gamma \vdash P}{\Gamma, \Delta \vdash \mu X.P}$

(CHAN) $\dfrac{\Gamma, x : \mathsf{Ch}(T), \Delta_\emptyset \vdash P}{\Gamma, \Delta_\emptyset \vdash (\nu x : \mathsf{Ch}(T))P}$

Table 4: Type rules for depth-boundedness

Note that the (PAR) rule implies that a process P that can be typed in a linear environment must be width-bounded with bound 2, since every name can then occur in either precisely one or precisely two parallel components.

Delegation of session names is handled by (OUT-1); session channels are linear, so the name y^p cannot appear in the continuation P. A special feature of our type system is that endpoint channels that are no longer usable cannot be delegated. Thus, in the rules (IN-1), (IN-2), and (OUT-1), the object type T_1 must be different from end.

5 A subject reduction property

To show our characterization of depth-boundedness, we state a type preservation property: For any well-typed process P, the type of the channel that gives rise to a reduction of P will evolve according to its session type.

Since this channel may be a restricted channel, we must also describe how the session types of restricted channels evolve. Every process in which all bound names are pairwise distinct gives rise to an internal type environment (Definition 12) that collects the types of the bound names; this is an overapproximation of the types of the active names in the process. This environment is defined as follows.

Definition 12. Let P be a process whose bound names are pairwise distinct. Γ_P denotes the internal type

environment of P; it is defined by the following clauses (where π denotes a prefix).

$$\Gamma_{P_1|P_2} = \Gamma_{P_1}, \Gamma_{P_2} \qquad \Gamma_{(\nu x:T)P} = x : T, \Gamma_P$$
$$\Gamma_{\mu X.P} = \Gamma_P \qquad \Gamma_{\pi.P} = \Gamma_P$$
$$\Gamma_X = \emptyset$$

The following substitution lemma for variables tells us about the annotated reductions of open process terms.

Lemma 1 (Substitution of variables in reductions)**.** *If* $P[\mu X.P/X] \xrightarrow{\{x\}} P'$ *then* $P \xrightarrow{\{x\}} P''$*, with* $P' = P''[\mu X.P/X]$.

Proof. Induction in the structure of P. □

Lemma 2 (Substitution of variables in typings of recursion)**.** *Suppose* $\Gamma, \Delta \vdash \mu X.P$ *and* $\Gamma, \Delta \vdash Q$*. Then* $\Gamma, \Delta \vdash Q[\mu X.P/X]$.

Proof. Induction in the structure of Q.

$Q = \mathbf{0}$**:** Trivial.

$Q = X$**:** Immediate, since $Q[\mu X.P/X] = \mu X.P$.

$Q = Y$ **(with** $Y \neq X$**):** Immediate.

$Q = Q_1 \mid Q_2$**:** We must then have concluded $\Gamma, \Delta \vdash Q$ using (PAR) with premises $\Gamma_1, \Delta \vdash Q_1$ and $\Gamma_2, \Delta \vdash Q_2$. By induction hypothesis we then have

$$\Gamma_1, \Delta \vdash Q_1[\mu X.P/X]$$
$$\Gamma_2, \Delta \vdash Q_2[\mu X.P/X]$$

We now use the (PAR) rule and get

$$\Gamma, \Delta \vdash Q_1[\mu X.P/X] \mid Q_2[\mu X.P/X]$$

The result now follows by the distributive property of substitution.

$Q = (\nu x : T)P_1$**:** We must have conclude $\Gamma, \Delta \vdash Q$ using (SESSION) with premise $\Gamma, x : S, \Delta \vdash P_1$. By induction hypothesis we have that $\Gamma, x : S, \Delta \vdash P_1[\mu X.P/X]$. But then by the (SESSION) rule we get that $\Gamma, \Delta \vdash (\nu x : T)P_1[\mu X.P/X]$, and we conclude that $\Gamma, \Delta \vdash Q[\mu X.P/X]$.

$Q = \mu Y.Q_1$**:** We must have concluded $\Gamma, \Delta \vdash Q$ using (REC) with premise $\Gamma, \Delta \vdash Q_1$. By induction hypothesis we have

$$\Gamma, \Delta \vdash Q_1[\mu X.P/X]$$

We can now apply (REC) to get the desired result.

$Q = a(x).Q_1$**:** We must have concluded $\Gamma, \Delta \vdash Q$ using (IN) with premise $\Gamma_1, a : T_2, x : T_1, \Delta \vdash Q_1$ and assuming that $\Gamma = \Gamma_1, a : ?T_1.T_2$. By applying the induction hypothesis, we get that

$$\Gamma_1, a : T_2, x : T_1, \Delta \vdash Q_1[\mu X.P/X]$$

An application of (IN) and the properties of substitution now gives us the result.

$Q = \overline{a}\langle x\rangle.Q_1$**:** Similar to the previous case.

□

We also need a substitution lemma for names.

Lemma 3 (Substitution of names)**.** *If* $\Gamma, x : T, \Delta \vdash P$ *and* $y \notin n(P)$ *then* $\Gamma, y : T, \Delta \vdash P\{^y/_x\}$.

Proof. Induction in the type rules. □

5.1 A fidelity theorem

For a binary session type system, subject reduction takes the form of *fidelity*: the communications in a well-typed process proceed according to the protocol specified by the channels involved.

Lemma 4 (Subject congruence and normalization)**.** *Suppose* $\Gamma, \Delta \vdash P$*. Then*

- *If* $P \equiv Q$*, then also* $\Gamma, \Delta \vdash Q$
- *If* $P \succ Q$*, then also* $\Gamma, \Delta \vdash Q$

Proof. Induction in the rules defining $\equiv$ and $\succ$. □

The fidelity theorem is a type preservation result: It states that the endpoint types evolve according to the reduction performed. If the name x giving rise to the reduction is free, the annotation of x in the type environment changes. If x is bound, its annotation in the restriction $(\nu x : T)$ changes to $(\nu x : T')$, where $T' = T\downarrow$.

Theorem 5 (Fidelity)**.** *Let* Γ *be a balanced type environment and let P be recursion-closed. If* $\Gamma, \Delta_\emptyset \vdash P$ *and* $P \xrightarrow{\alpha} P'$ *where* $x = \mathrm{n}(\alpha)$ *then*

- *if* $x \in \mathrm{fn}(P)$ *and* $\Gamma = \Gamma'', x : T$*, then* $\Gamma', \Delta_\emptyset \vdash P'$ *where* Γ' *is balanced and* $\Gamma' = \Gamma'', x : T\downarrow$
- *if* $x \notin \mathrm{fn}(P)$*, then* $\Gamma, \Delta_\emptyset \vdash P'$ *and if* $\Gamma_P = \Gamma'', x : T$ *then* $\Gamma_{P'} = \Gamma'', x : T\downarrow$ *and* $\Gamma_{P'}$ *is balanced.*

Proof. Induction in the reduction rules.

Com-Annot Here, only the first case is relevant. We know that $P = a^p(x).P_1 \mid \overline{a^p}\langle y^q\rangle.P_2$. Since $\Gamma, \Delta_\emptyset \vdash P$, we must have that $\Gamma = \Gamma_1 + \Gamma_2$ where

$$\Gamma_1, \Delta_\emptyset \vdash a^p(x).P_1 \tag{1}$$

and

$$\Gamma_2, \Delta_\emptyset \vdash \overline{a^p}\langle y^q\rangle.P_2. \tag{2}$$

We must have used (IN) to conclude (1), so we have $\Gamma_1(a^p) = ?T_1.S$ and, letting $\Gamma_1 = \Gamma_1' + a^p : ?T_1.S$, we have

$$\Gamma_1', a^p : S, x : T_1, \Delta_\emptyset \vdash P_1. \tag{3}$$

Similarly, we must have used (OUT) to conclude (2). Since Γ is balanced, we have $\Gamma_2(a^{\overline{p}}) = !T_1.\overline{S}$. By the substitution lemma Lemma 3 and (3), we have $\Gamma_1', a^p : S, y^q : T_1, \Delta_\emptyset \vdash P_1\{^y/_x\}$. Similarly, letting $\Gamma_2 = \Gamma_2', a^{\overline{p}} : !T_1.\overline{S}, y^q : T_1$, we get $\Gamma_2', a^{\overline{p}} : \overline{S}, \Delta_\emptyset \vdash P_2$. An application of (PAR) now gives us that

$$\Gamma_1' + \Gamma_2' + a^p : S, a^{\overline{p}} : \overline{S}, y : T_1, \Delta_\emptyset \vdash P_1\{^y/_x\} \mid P_2$$

The type environment $\Gamma_1' + \Gamma_2' + a^p : S, a^{\overline{p}} : \overline{S}, y : T_1$ is balanced, since Γ_1' and Γ_2' are balanced and since y must appear with polarity $\overline{q}$ in one of these (because Γ is balanced).

Par-Annot Since $\Gamma, \Delta_\emptyset \vdash P \mid Q$, we have that $\Gamma_1, \Delta_\emptyset \vdash P$ where $\Gamma = \Gamma_1 + \Gamma_2$. The result now follows easily by an application of the induction hypothesis to the reduction $P \xrightarrow{a} P'$ and subsequent use of the (PAR) rule.

New-Annot There are two cases here: whether $x = a$ or $x \neq a$. In both cases, the result follows immediately by the induction hypothesis and use of the (SESSION) rule.

Unfold-Annot Follows from Lemma 4 and a direct application of the induction hypothesis.

Struct-Annot Follows from Lemma 4 and a direct application of the induction hypothesis.

□

6 Soundness of the type system for depth-boundedness

In the following we will consider the correctness properties of the type system for depth boundedness.

6.1 Properties of unfolding and nesting

We first establish a collection of properties that hold for arbitrary processes. Next we show that there are further properties guaranteed by well-typed processes.

The following lemma describes how reductions occur. Reductions can happen directly or may need unfoldings.

Lemma 6. *Let P be an arbitrary recursion-closed process.*

1. *If $P \xrightarrow{\{x\}} P'$, then there exists an unfolding context C and a process Q such that $P \equiv C[Q]$ and $P' \equiv C[Q']$, and $Q \xrightarrow{\{x\}} Q'$ is an instance of* (COM-ANNOT).
2. *If $P \xrightarrow{\{\mathsf{rec},x\}} P'$ then there exists an unfolding context C and either $P \equiv C[\mu X.Q_1]$ for some Q_1 where $Q_1[\mu X.Q_1/X] \xrightarrow{\{x\}} Q_1'$ and $P' \equiv C[Q_1']$ or $P \equiv C[(\mu X.Q_1) \mid Q_2]$ where $Q_1[\mu X.Q_1/X] \mid Q_2 \xrightarrow{\{\mathsf{rec},x\}} Q_1' \mid Q_2'$ is an instance of* (COM-ANNOT) *and $P' \equiv C[Q_1' \mid Q_2']$.*

Proof. By induction in the annotated reduction rules. The proof of Case 2 uses Case 1. □

6.2 Nesting properties of well-typed processes

We now restrict our attention to well-typed processes. The only potential source of unbounded restriction depth is the presence of recursion, and we now show how our type system controls the introduction of new bound names in the presence of recursion.

The first lemma tells us that bound names introduced by an unfolding do not interfere with names in its surrounding process that represent terminated channels.

Lemma 7. *If $\Gamma, \Delta \vdash (\nu c : (\mathsf{end}, \mathsf{end}))P$ then $c \notin \mathrm{fn}(P)$.*

The following lemma tells us that names that appear in an unfolding context will not reappear free in the result of unfolding a recursive process.

Definition 13 (Known bound names)**.** The set of known bound names in an unfolding context is defined by

$$\mathrm{kn}([] \mid P) = \emptyset \qquad\qquad \mathrm{kn}((\nu x : T)C) = \{x\} \cup \mathrm{kn}(C)$$

Lemma 8. *Suppose we have* $\Gamma,\Delta \vdash C[X]$ *where* $C[X]$ *is recursion-closed and* X *occurs in* $\mu X.P$. *Then we also have* $\Gamma,\Delta \vdash C[\mu X.P]$ *and* $\mathrm{kn}(C) \cap \mathrm{fn}(\mu X.P) = \emptyset$.

Theorem 9. *Let* P *be recursion-closed. Suppose* $\Gamma,\Delta_\emptyset \vdash P$. *Then* P *is depth-bounded.*

Outline. The session types provide a bound on the nesting depth of a well-typed process. Suppose $\Gamma,\Delta_\emptyset \vdash P$. Let $d(\Gamma,P)$ denote the sum of the depths of the session types in Γ and in Γ_P, i.e.

$$d(\Gamma,P) = \sum_{x:T\in\Gamma \text{ or } x:T\in\Gamma_P} d(T)$$

In a process P with k bound names, we know from Theorem 5 that there can be at most $(d(\Gamma+\Gamma_P)/2)-k$ reduction steps before an unfolding has to take place, since every reduction step will decrease the depth of one of the session types in $\mathrm{ran}(\Gamma)\cup\mathrm{ran}(\Gamma_P)$. Whenever unfoldings occur, the bound names in the unfolding are distinct from those already known and will all be names of session channels. Moreover, when the unfolding is reached, the channel used in the reduction will no longer be available. As a consequence we see that the nesting depth will therefore not increase. □

7 A type system for name-boundedness

We now show to modify our previous type system such that every well-typed process will be name-bounded. The challenge is again one of controlling recursion. As before, the crucial observation is that if private channels are linear, then all the channels that have been used when a recursion unfolding takes place, can then be discarded.

In the case of name-boundedness, extra care must be taken, since recursion may now accumulate an unbounded number of finite components that each contain pairwise distinct bound names.

Example 4. The untyped process

$$P_2 = \mu X.(\nu r_1)(\nu r_2)(\overline{r_1}\langle a\rangle.X \mid r_1(x).X \mid \overline{r_2}\langle a\rangle \mid r_2(x))$$

shows two problems that must be dealt with. Firstly, unfolding a recursion may introduce more parallel recursive components that each have their own bound names. In this case, every communication on r_1 will introduce two new parallel copies of the recursive process. Secondly, unfolding may introduce finite (non-recursive) components which contain bound names that persist – in this case, we get new copies of $(\nu r_2)(\overline{r_2}\langle a\rangle \mid r_2(x))$ for every unfolding.

The type language is

$$S_{\mathsf{lin}} ::= ?T_{\mathsf{lin}}.S_{\mathsf{lin}} \mid !T_{\mathsf{lin}}.S_{\mathsf{lin}} \mid \mathsf{end} \qquad S_{\mathsf{un}} ::= \mathsf{Ch}(S_{\mathsf{un}})$$
$$T_{\mathsf{lin}} ::= (S_{\mathsf{lin}}, S_{\mathsf{un}}) \mid (S_{\mathsf{lin}}, S_{\mathsf{lin}}) \qquad T ::= T_{\mathsf{lin}} \mid S_{\mathsf{un}}$$

Note that names of unlimited type S_{un} can only be used to delegate channels of unlimited type.

The type rules are as in the original type system, but we now modify the notions of addition for type environments and for recursion environments. We add pairs (Γ_1,Δ_1) and (Γ_2,Δ_2) as follows.

Definition 14. Let Γ_1,Γ_2 be type environments and let Δ_1,Δ_2 be recursion environments where at least one of Δ_1,Δ_2 is $\Delta_\emptyset$. We define $(\Gamma_1,\Delta_1)+(\Gamma_2,\Delta_2) = (\Gamma_1+\Gamma_2,\Delta_1+\Delta_2)$ where Γ_1 is unlimited if $\Delta_1 = \Delta_\emptyset$ and Γ_2 is linear if $\Delta_2 \neq \Delta_\emptyset$.

The intention is that an empty recursion environment must now go together with an unlimited type environment. In other words: Non-recursive subprocesses can only contain unlimited names.

We say that a type environment Γ is *limited* if for every $x \in \mathrm{dom}(\Gamma)$ we have that $\Gamma(x) = (T_{\mathsf{lin}}, \overline{T_{\mathsf{lin}}})$ for some T_{lin}. That is, the environment is balanced, and no name has an unlimited type.

A type environment Γ is *skew* if $\Gamma = \Gamma_1 + \Gamma_2$ with $\mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \emptyset$, Γ_1 is linear and for all $x \in \mathrm{dom}(\Gamma_2)$ we have that $\Gamma(x) = (T_{\mathsf{lin}}, T_{\mathsf{un}})$ for some $T_{\mathsf{lin}}, T_{\mathsf{un}}$.

7.1 Fidelity

As in the case of the previous type system, we need a fidelity result.

Theorem 10 (Fidelity)**.** *Let Γ be a type environment. If $\Gamma, \Delta_\emptyset \vdash P$ and $P \xrightarrow{x} P'$ then*

- *if $x \in \mathrm{fn}(P)$ and $\Gamma = \Gamma'', x : T$, then $\Gamma', \Delta_\emptyset \vdash P'$ where Γ' is balanced and $\Gamma' = \Gamma'', x : T \downarrow$*
- *if $x \notin \mathrm{fn}(P)$, then $\Gamma, \Delta_\emptyset \vdash P'$ and if $\Gamma_P = \Gamma'', x : T$ then $\Gamma_{P'} = \Gamma'', x : T \downarrow$ and $\Gamma_{P'}$ is balanced.*

Since the new type system specialized the previous one, this result is easily established.

7.2 Soundness for name-boundedness

We will show that if a process is well-typed in a limited environment, then it is name-bounded.

To show that a well-typed process P is name-bounded, we will show that

- For some k, whenever $P \rightarrow^* P'$, then P' has at most k recursion instances in P'
- For some m, whenever $P \rightarrow^* P'$, every recursive subprocess of P' contains at most m distinct bound names
- There are only free names in the non-recursive part of P

Since every well-typed process is known to be depth-bounded, the result will then follow.

Our first lemma gives a characterization of well-typed recursive processes: They can contain at most one instance of each recursion variable.

Lemma 11. *Let $\mu X.P$ be a process for which all binding occurrences of recursion variables are distinct. If $\Gamma, \Delta \vdash \mu X.P$, there is at most one occurrence of X in P.*

Proof. Suppose to the contrary that there is more than one occurrence of X in P. We then have that $\mu X.P = \mu X.(\nu \vec{n})(C_1[X] \mid C_2[X] \mid P')$ where n is a set of names (possibly empty), and C_1 and C_2 are process contexts.

The derivation of the type judgement $\Gamma, \Delta \vdash \mu X.(\nu \vec{n})(C_1[X] \mid C_2[X] \mid P')$ must have used the (REC) type rule in its final step, having premise $\Gamma, \Delta, X : \Gamma \vdash (\nu \vec{n})(C_1[X] \mid C_2[X] \mid P'$. But the derivation of this judgement must have used the (SESSION) rule a number of times, preceded by an application of (PAR) with premises $\Gamma_1, \Delta, X : \Gamma \vdash C_1[X]$ and $\Gamma_2, \Delta_\emptyset \vdash C_2[X]$ where Γ_2 is unlimited. However, there can be no derivation of the latter, since this would require the rule (VAR) in which it is assumed that the type environment is linear.

We therefore conclude that our initial assumption was wrong; there can be at most one occurrence of X in P. □

This lemma tells us that there can be no finite, non-recursive subprocesses of a recursive process with their own bound names; any bound name found in a non-recursive subprocess will also appear in the recursive part of the process.

Lemma 12. *If $\Gamma,\Delta \vdash \mu X.(C[X] \mid P)$ where $\mu X.(C[X] \mid P)$ is in inner normal form and $C[X]$ is a process context, then for every $n \in \mathrm{bn}(P)$ we have that $n \in \mathrm{bn}(C[X])$.*

Proof. Consider a name $n \in \mathrm{bn}(P)$. Suppose $n \notin \mathrm{bn}(C[X])$. Since $\mu X.(C[X] \mid P)$ is in inner normal form, we would then have a subprocess $(\nu n : T)P'$ of P that would be typed using the (SESSION) rule. But for this rule to be applicable, a recursion variable must be present in the type environment. This cannot be the case, as P is non-recursive. □

We now show that the number of recursive subprocesses that will appear in any reduction sequence for a well-typed process is bounded. Let $\mathsf{recs}(P)$ denote the number of simultaneous recursion instances in P and let $\mathsf{recv}(P)$ denote the multiset of recursion variable occurrences in P.

Together, the following two lemmas give an upper bound on the number of recursion instances in any reduction sequence of a well-typed process.

Lemma 13. *Suppose $\Gamma,\Delta \vdash P$ and $P \xrightarrow{\alpha} P'$ was proved without using instances of* (UNFOLD-ANNOT). *Then $\mathsf{recs}(P) \geq \mathsf{recs}(P')$.*

Lemma 14. *Suppose $\Gamma,\Delta \vdash P$ where $\mathrm{dom}(\Delta) \cap \mathsf{recv}(P) = \emptyset$ and $P > P_1$. Then $\mathsf{recs}(P) \geq \mathsf{recs}(P_1)$.*

The following normal form theorem is crucial.

Theorem 15. *If $\Gamma,\Delta \vdash P$, then there exists a $k \geq 0$ such that whenever $P \to^* P'$, we have $P \equiv P_1 \mid P_2$ where $\mathsf{recs}(P_1) \leq k$, $\mathsf{recs}(P_2) = 0$ and P_2 contains no restrictions.*

Proof. We show that for all $n \geq 0$, if $P \to^n P'$, then we have $P \equiv P_1 \mid P_2$ where $\mathsf{recs}(P_1) \leq k$, $\mathsf{recs}(P_2) = 0$ and P_2 contains no restrictions. The proof of this proceeds by induction in n.

$n = 0$: Here we let $k = \mathsf{recs}(P)$ and proceed by induction in the type derivation of $\Gamma,\Delta \vdash P$. We consider each rule in turn.

- **(IN-1), (IN-2), (OUT-1) and (OUT-2):** None of these rules could have been used, since P would then have no reductions.
- **(PAR):** Here we can use the commutativity and associativity axioms for structural congruence to rewrite P in the desired form.
- **(VAR):** Cannot apply, since we assume that $\Gamma,\Delta \vdash P$.
- **(REC), (NIL), (SESSION):** These are immediate.

Assume for n, prove for $n+1$: This is a straightforward induction in the type rules.

□

Theorem 16. *If $\Gamma,\Delta \vdash P$, then P is name-bounded.*

Proof. There is a $k \geq 0$ such that if $\Gamma,\Delta \vdash P$, whenever $P \to^* P'$, there are at most k recursive subprocesses of P'. Since the new type system is a subsystem of the type system for depth-boundedness, there exists a d such that the recursion depth of P' is at most d for any such P'.

Every bound name in a non-recursive subterm of a recursive subprocess occurs in the recursion part as well. Now consider an outer normal form P'' of P'. We have $P'' = (\nu x_1)\ldots(\nu x_d)P^{(3)}$ for some $P^{(3)}$ that does not contain restrictions at the outermost level. Moreover, for some $k' \leq k$ we have $P^{(3)} \equiv P_1^{(3)} \mid \cdots P_{k'}^{(3)} \mid P_{k'+1}^{(3)}$ where $P_1^{(3)},\ldots,P_{k'}^{(3)}$ contain recursion instances and $P_{k'+1}^{(3)}$ is a process not containing recursion instances. We know that for some d there are at most $d \cdot k$ bound names in P'. □

8 The relation to other classes of processes

Because of the use of binary session types, typable process in our systems will be width-bounded with name width 2. On the other hand, both type systems allow us to type processes that are not finitary. The classes of typable processes differ from those already studied. The process $P_1 \stackrel{\text{def}}{=} (\mu X.(\nu a)a(x).X \mid \overline{a}\langle b\rangle \mid \overline{b}\langle c\rangle)$ is not a finite-control process, since the reduction sequence $P \rightarrow^k P_1 \mid \overline{b}\langle c\rangle \mid \cdots \mid \overline{b}\langle b\rangle$ that results in $k-1$ parallel components, each being a simple output, shows that the number of parallel components along a computation can be unbounded for a well-typed process. This means that P_1 is neither a finite-control process [3] nor a bounded process in the sense of [2]. On the other hand, P_1 is depth-bounded, and in fact also width-bounded as every bound name occurs in precisely two parallel components. Moreover, the typable processes are incomparable with the processes studied in [1] since these do not allow for delegation of input capabilities.

9 Conclusions and ideas for further work

In this paper we have presented two session type systems for a π-calculus with recursion. One guarantees depth-boundedness, and the other system, which is a subsystem of it, guarantees name-boundedness. Both systems assume that names are always used in finite-length sessions before a recursive call is initiated.

In the paper by D'Osualdo and Ong [4] a type inference algorithm is proposed that makes it possible to provide a safe bound on the restriction depth for depth-bounded processes. A further topic of investigation is to adapt the type inference algorithm proposed in [6] to the setting of the type systems of the present paper. We conjecture that this is straightforward. The type systems presented in this paper are simpler than many other session type systems, in that they do not involve recursive types; the sole difference is that of the presence of recursion instead of replication in the π-calculus.

In both systems, the number of parallel components in a well-typed system can be unbounded, and well-typed processes need not be finite-control. Conversely, finite-control processes need not be well-typed in the present systems, since finite-control processes are not necessarily width-bounded with width 2.

Another important question to be answered is that of the exact relationshop between our type system for depth-boundedness and the type system due to D'Osualdo and Ong [4].

References

[1] Roberto M. Amadio and Charles Meyssonnier. On decidability of the control reachability problem in the asynchronous pi-calculus. *Nordic J. of Computing*, 9(2):70–101, June 2002.

[2] Luís Caires. Behavioral and spatial observations in a logic for the pi-calculus. In Igor Walukiewicz, editor, *Proceedings of FOSSACS 2004*, LNCS 2987, pp. 72-89, Springer, 2004. doi:10.1007/978-3-540-24727-2_7.

[3] Mads Dam. Model checking mobile processes. *Inf. Comput.*, 129(1):35–51, 1996. doi:10.1006/inco.1996.0072.

[4] Emanuele D'Osualdo and Luke Ong. A type system for proving depth boundedness in the pi-calculus. *CoRR*, abs/1502.00944, 2015. `http://arxiv.org/abs/1502.00944`

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

[6] Eva Fajstrup Graversen, Jacob Buchreitz Harbo, Hans Hüttel, Mathias Ormstrup Bjerregaard, Niels Sonnich Poulsen, and Sebastian Wahl. Type inference for session types in the π-calculus. In T. Hildebrandt, A. Ravara, J. van der Werf, and M. Weidlich (ed.) *Proc. of WS-FM 2014 and WS-FM/BEAT 2015*, LNCS 9421, Springer, 2015. doi:10.1007/978-3-319-33612-1_7.

[7] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *Proceedings of ESOP'98*, pages 122–138, 1998. doi:10.1007/BFb0053567.

[8] Rainer Hüchting, Rupak Majumdar, and Roland Meyer. *A Theory of Name Boundedness*, pages 182–196. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. doi:10.1007/978-3-642-40184-8_14.

[9] Ivan Lanese, Jorge A. Pérez, Davide Sangiorgi, and Alan Schmitt. On the expressiveness and decidability of higher-order process calculi. *Inf. Comput.*, 209(2):198–226, 2011. doi:10.1016/j.ic.2010.10.001.

[10] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. Fencing off Go: Liveness and safety for channel-based programming. In *POPL 2017*, pp. 748–761, New York, NY, USA, 2017. ACM. doi:10.1145/3009837.

[11] Roland Meyer. On boundedness in depth in the pi-calculus. In Giorgio Ausiello, Juhani Karhumäki, Giancarlo Mauri, and C.-H. Luke Ong, editors, *Proceedings of TCS 2008*, volume 273 of *IFIP*, pages 477–489. Springer, 2008. doi:10.1007/978-0-387-09680-3_32.

Distributive Laws for Monotone Specifications*

Jurriaan Rot
Radboud University, Nijmegen

jrot@cs.ru.nl

Turi and Plotkin introduced an elegant approach to structural operational semantics based on universal coalgebra, parametric in the type of syntax and the type of behaviour. Their framework includes abstract GSOS, a categorical generalisation of the classical GSOS rule format, as well as its categorical dual, coGSOS. Both formats are well behaved, in the sense that each specification has a unique model on which behavioural equivalence is a congruence. Unfortunately, the combination of the two formats does not feature these desirable properties. We show that *monotone* specifications—that disallow negative premises—do induce a canonical distributive law of a monad over a comonad, and therefore a unique, compositional interpretation.

1 Introduction

Structural operational semantics (SOS) is an expressive and popular framework for defining the operational semantics of programming languages and calculi. There is a wide variety of specification formats that syntactically restrict the full power of SOS, but guarantee certain desirable properties to hold [1]. A famous example is the so-called GSOS format [5]. Any GSOS specification induces a unique interpretation which is compositional with respect to (strong) bisimilarity.

In their seminal paper [22], Turi and Plotkin introduced an elegant mathematical approach to structural operational semantics, where the type of syntax is modeled by an endofunctor Σ and the type of behaviour is modeled by an endofunctor B. Operational semantics is then given by a *distributive law* of Σ over B. In this context, models are *bialgebras*, which consist of a Σ-algebra and a B-coalgebra over a common carrier. One major advantage of this framework over traditional approaches is that it is parametric in the type of behaviour. Indeed, by instantiating the theory to a particular functor B, one can obtain well behaved specification formats for probabilistic and stochastic systems, weighted transition systems, streams, and many more [14, 15, 4].

Turi and Plotkin introduced several kinds of natural transformations involving Σ and B, the most basic one being of the form $\Sigma B \Rightarrow B\Sigma$. If B is a functor representing labelled transition systems, then a typical rule that can be represented in this format is the following:

$$\frac{x \xrightarrow{a} x' \qquad y \xrightarrow{a} y'}{x \otimes y \xrightarrow{a} x' \otimes y'} \tag{1}$$

This rule should be read as follows: if x can make an a-transition to x', and y an a-transition to y', then $x \otimes y$ can make an a-transition to $x' \otimes y'$. Any specification of the above kind induces a unique *supported*

*The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement nr. 320571, and the Netherlands Organisation for Scientific Research (NWO), CoRE project, dossier number: 612.063.920. Part of this research was carried out during a visit of the author to the University of Warsaw, supported by the Warsaw Center of Mathematics and Computer Science (WCMCS).

K. Peters and S. Tini (Eds.): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics (EXPRESS/SOS 2017).
EPTCS 255, 2017, pp. 83–97, doi:10.4204/EPTCS.255.6

model, which is a B-coalgebra over the initial algebra of Σ. If Σ represents a signature and B represents labelled transition systems, then this model is a transition system of which the state space is the set of closed terms in the signature, and, informally, a term makes a transition to another term if and only if there is a rule in the specification justifying this transition.

A more interesting kind is an *abstract GSOS specification*, which is a natural transformation of the form $\Sigma(B \times \mathsf{Id}) \Rightarrow B\Sigma^*$, where Σ^* is the *free monad* for Σ (assuming it exists). If B is the functor that models (image-finite) transition systems, and Σ is a functor representing a signature, then such specifications correspond to classical GSOS specifications [22, 4]. As opposed to the basic format, GSOS rules allow complex terms in conclusions, as in the following rule specifying a constant c:

$$\frac{}{c \xrightarrow{a} \sigma(c)} \tag{2}$$

where σ is some other operator in the signature (represented by Σ), which can itself be defined by some GSOS rules. The term $\sigma(c)$ is constructed from a constant and a unary operator from the signature, as opposed to the conclusion $x' \otimes y'$ of the rule in (1), which consists of a single operator and variables. Indeed, the free monad Σ^* occurring in an abstract GSOS specification is precisely what allows a complex term such as $\sigma(c)$ in the conclusion.

Dually, one can consider *coGSOS specifications*, which are of the form $\Sigma B^\infty \Rightarrow B(\Sigma + \mathsf{Id})$, where B^∞ is the cofree comonad for B (assuming it exists). In the case of image-finite labelled transition systems, this format corresponds to the *safe ntree format* [22]. A typical coGSOS rule is the following:

$$\frac{x \xrightarrow{a} x' \qquad x' \overset{a}{\not\to}}{\sigma(x) \xrightarrow{a} x'} \tag{3}$$

This rule uses two steps of lookahead in the premise; this is supported by the cofree comonad B^∞ in the natural transformation. The symbol $x' \overset{a}{\not\to}$ represents a *negative premise*, which is satisfied whenever x' does not make an a-transition.

Both GSOS and coGSOS specifications induce *distributive laws*, and as a consequence they induce unique supported models on which behavioural equivalence is a congruence. The two formats are incomparable in terms of expressive power: GSOS specifications allow rules that involve complex terms in the conclusion, whereas coGSOS allows arbitrary lookahead in the arguments. It is straightforward to *combine* GSOS and coGSOS as a natural transformation of the form $\Sigma B^\infty \Rightarrow B\Sigma^*$, called a *biGSOS specification*, generalising both formats. However, such specifications are, in some sense, too expressive: they do not induce unique supported models, as already observed in [22]. For example, the rules (2) and (3) above (which are GSOS and coGSOS respectively) can be combined into a single biGSOS specification. Suppose this combined specification has a model. By the axiom for c, there is a transition $c \xrightarrow{a} \sigma(c)$ in this model. However, is there a transition $\sigma(c) \xrightarrow{a} \sigma(c)$? If there is not, then by the rule for σ, there is; but if there is such a transition, then it is not derivable, so it is not in the model! Thus, a supported model does not exist. In fact, it was recently shown that, for biGSOS, it is undecidable whether a (unique) supported model exists [17].

The use of negative premises in the above example (and in [17]) is crucial. In the present paper, we introduce the notion of *monotonicity* of biGSOS specifications, generalising monotone abstract GSOS [8]. In the case that B is a functor representing labelled transition systems, this corresponds to the absence of negative premises, but the format does allow lookahead in premises as well as complex terms in conclusions. Monotonicity requires an *order* on the functor B—technically, our definition of monotonicity is based on the *similarity* order [10] induced on the final coalgebra.

We show that if there is a pointed DCPO structure on the functor B, then any monotone biGSOS specification yields a *least* model as its operational interpretation. Indeed, monotone specifications do not necessarily have a unique model, but it is the least model which makes sense operationally, since this corresponds to the natural notion that every transition has a *finite* proof. Our main result is that if the functor B has a DCPO structure, then every monotone specification yields a canonical *distributive law* of the free monad for Σ over the cofree comonad for B. Its unique model coincides with the least supported model of the specification. As a consequence, behavioural equivalence on this model is a congruence.

However, the conditions of these results are a bit too restrictive: they rule out labelled transition systems, the main example. The problem is that the functors typically used to model transition systems either fail to have a cofree comonad (the powerset functor) or to have a DCPO structure (the finite or countable powerset functor). In the final section, we mitigate this problem using the theory of (countably) presentable categories and accessible functors. This allows us to relax the requirement of DCPO structure only to countable sets, given that the functor B is countably accessible (this is weaker than being finitary, a standard condition in the theory of coalgebras) and the syntax consists only of countably many operations each with finite arity. In particular, this applies to labelled transition systems (with countable branching) and certain kinds of weighted transition systems.

Related work The idea of studying distributive laws of monads over comonads that are not induced by GSOS or coGSOS specifications has been around for some time (e.g., [4]), but, according to a recent overview paper [15], general bialgebraic formats (other than GSOS or coGSOS) which induce such distributive laws have not been proposed so far. In fact, it is shown by Klin and Nachyła that the general problem of extending biGSOS specifications to distributive laws is undecidable [16, 17]. The current paper shows that one does obtain distributive laws from biGSOS specifications when monotonicity is assumed (negative premises are disallowed). A fundamentally different approach to positive formats with lookahead, not based on the framework of bialgebraic semantics but on labelled transition systems modeled very generally in a topos, was introduced in [21]. It is deeply rooted in labelled transition systems, and hence seems incomparable to our approach based on generic coalgebras for ordered functors. An abstract study of distributive laws of monads over comonads and possible morphisms between them is in [18], but it does not include characterisations in terms of simpler natural transformations.

Structure of the paper Section 2 contains the necessary preliminaries on bialgebras and distributive laws. In Section 3 we recall the notion of similarity on coalgebras, which we use in Section 4 to define monotone specifications and prove the existence of least supported models. Section 5 contains our main result: canonical distributive laws for monotone biGSOS specifications. In Section 6, this is extended to countably accessible functors.

Notation We use the categories Set of sets and functions, PreOrd of preorders and monotone functions, and $\mathsf{DCPO}_\perp$ of pointed DCPOs and continuous maps. By $\mathscr{P}$ we denote the (contravariant) power set functor; $\mathscr{P}_c$ is the countable power set functor and $\mathscr{P}_f$ the finite power set functor. Given a relation $R \subseteq X \times Y$, we write $\pi_1 \colon R \to X$ and $\pi_2 \colon R \to Y$ for its left and right projection, respectively. Given another relation $S \subseteq Y \times Z$ we denote the composition of R and S by $R \circ S$. We let $R^{\mathsf{op}} = \{(y,x) \mid (x,y) \in R\}$. For a set X, we let $\Delta_X = \{(x,x) \mid x \in X\}$. The graph of a function $f \colon X \to Y$ is $\mathsf{Graph}(f) = \{(x, f(x)) \mid x \in X\}$. The image of a set $S \subseteq X$ under f is denoted simply by $f(S) = \{f(x) \mid x \in S\}$, and the inverse image of $V \subseteq Y$ by $f^{-1}(V) = \{x \mid f(x) \in V\}$. The pairing of two functions f, g with a common domain is denoted by $\langle f, g \rangle$ and the copairing (for functions f, g with a common codomain) by $[f, g]$. The set of functions

from X to Y is denoted by Y^X. Any relation $R \subseteq Y \times Y$ can be lifted pointwise to a relation on Y^X; in the sequel we will simply denote such a pointwise extension by the relation itself, i.e., for functions $f, g\colon X \to Y$ we have $f\,R\,g$ iff $f(x)\,R\,g(x)$ for all $x \in X$, or, equivalently, $(f \times g)(\Delta_X) \subseteq R$.

Acknowledgements The author is grateful to Henning Basold, Marcello Bonsangue, Bartek Klin and Beata Nachyła for inspiring discussions and suggestions.

2 (Co)algebras, (co)monads and distributive laws

We recall the necessary definitions on algebras, coalgebras, and distributive laws of monads over comonads. For an introduction to coalgebra see [20, 12]. All of the definitions and results below and most of the examples can be found in [15], which provides an overview of bialgebraic semantics. Unless mentioned otherwise, all functors considered are endofunctors on Set.

2.1 Algebras and monads

An *algebra* for a functor $\Sigma\colon \mathsf{Set} \to \mathsf{Set}$ consists of a set X and a function $f\colon \Sigma X \to X$. An *(algebra) homomorphism* from $f\colon \Sigma X \to X$ to $g\colon \Sigma Y \to Y$ is a function $h\colon X \to Y$ such that $h \circ f = g \circ \Sigma h$. The category of algebras and their homomorphisms is denoted by $\mathsf{alg}(\Sigma)$.

A *monad* is a triple $\mathscr{T} = (T, \eta, \mu)$ where $T\colon \mathsf{Set} \to \mathsf{Set}$ is a functor and $\eta\colon \mathsf{Id} \Rightarrow T$ and $\mu\colon TT \Rightarrow T$ are natural transformations such that $\mu \circ T\eta = \mathsf{id} = \mu \circ \eta T$ and $\mu \circ \mu T = \mu \circ T\mu$. An *(Eilenberg-Moore, or EM)-algebra* for $\mathscr{T}$ is a T-algebra $f\colon TX \to X$ such that $f \circ \eta_X = \mathsf{id}$ and $f \circ \mu_X = f \circ Tf$. We denote the category of EM-algebras by $\mathsf{Alg}(\mathscr{T})$.

We assume that a *free monad* (Σ^*, η, μ) for Σ exists. This means that there is a natural transformation $\iota\colon \Sigma\Sigma^* \Rightarrow \Sigma^*$ such that ι_X is a *free algebra* on the set X of generators, that is, the copairing of

$$\Sigma\Sigma^* X \xrightarrow{\iota_X} \Sigma^* X \xleftarrow{\eta_X} X$$

is an initial algebra for $\Sigma + X$. By Lambek's lemma, $[\iota_X, \eta_X]$ is an isomorphism. Any algebra $f\colon \Sigma X \to X$ induces a $\Sigma + X$-algebra $[f, \mathsf{id}]$, and therefore by initiality a Σ^*-algebra $f^*\colon \Sigma^* X \to X$, which we call the *inductive extension* of f. In particular, the inductive extension of ι_X is μ_X. This construction preserves homomorphisms: if h is a homomorphism from f to g, then it is also a homomorphism from f^* to g^*.

Example 1. An algebraic signature (a countable collection of operator names with finite arities) induces a polynomial functor Σ, meaning here a countable coproduct of finite products. The free monad Σ^* constructs terms, that is, $\Sigma^* X$ is given by the grammar $t ::= \sigma(t_1, \ldots, t_n) \mid x$ where x ranges over X and σ ranges over the operator names (and n is the arity of σ), so in particular $\Sigma^* \emptyset$ is the set of closed terms over Σ.

2.2 Coalgebras and comonads

A *coalgebra* for the functor B consists of a set X and a function $f\colon X \to BX$. A *(coalgebra) homomorphism* from $f\colon X \to BX$ to $g\colon Y \to BY$ is a function $h\colon X \to Y$ such that $Bh \circ f = g \circ h$. The category of B-coalgebras and their homomorphisms is denoted by $\mathsf{coalg}(B)$.

A *comonad* is a triple $\mathscr{D} = (D, \varepsilon, \delta)$ consisting of a functor $D\colon \mathsf{Set} \to \mathsf{Set}$ and natural transformations $\varepsilon\colon D \Rightarrow \mathsf{Id}$ and $\delta\colon D \Rightarrow DD$ satisfying axioms dual to the monad axioms. The category of Eilenberg-Moore coalgebras for $\mathscr{D}$, defined dually to EM-algebras, is denoted by $\mathsf{CoAlg}(\mathscr{D})$.

We assume that a *cofree comonad* $(B^\infty, \delta, \varepsilon)$ for B exists. This means that there is a natural transformation $\theta\colon B^\infty \Rightarrow BB^\infty$ such that θ_X is a *cofree coalgebra* on the set X, that is, the pairing of

$$BB^\infty X \xleftarrow{\theta_X} B^\infty X \xrightarrow{\varepsilon_X} X$$

is a final coalgebra for $B \times X$. Any coalgebra $f\colon X \to BX$ induces a $B \times X$-coalgebra $\langle f, \mathsf{id}\rangle$, and therefore by finality a B^∞-coalgebra $f^\infty\colon X \to B^\infty X$, which we call the *coinductive extension* of f. In particular, the coinductive extension of θ_X is δ_X. This construction preserves homomorphisms: if h is a homomorphism from f to g, then it is also a homomorphism from f^∞ to g^∞.

Example 2. Consider the Set functor $BX = A \times X$ for a fixed set A. Coalgebras for B are called *stream systems*. There exists a final B-coalgebra, whose carrier can be presented as the set A^ω of all streams over A, i.e., $A^\omega = \{\sigma \mid \sigma\colon \omega \to A\}$ where ω is the set of natural numbers. For a set X, $B^\infty X = (A \times X)^\omega$. Given $f\colon X \to A \times X$, its coinductive extension $f^\infty\colon X \to B^\infty X$ maps a state $x \in X$ to its infinite unfolding. The final coalgebra of $GX = A \times X + 1$ consists of finite and infinite streams over A, that is, elements of $A^* \cup A^\omega$. For a set X, $G^\infty X = (A \times X)^\omega \cup (A \times X)^* \times X$.

Example 3. Labelled transition systems are coalgebras for the functor $(\mathscr{P}-)^A$, where A is a fixed set of labels. Image-finite transition systems are coalgebras for the functor $(\mathscr{P}_f-)^A$, and coalgebras for $(\mathscr{P}_c-)^A$ are transition systems which have, for every action $a \in A$ and every state x, a countable set of outgoing a-transitions from x. A final coalgebra for $(\mathscr{P}-)^A$ does not exist (so there is no cofree comonad for it). However, both $(\mathscr{P}_f-)^A$ and $(\mathscr{P}_c-)^A$ have a final coalgebra, consisting of possibly infinite rooted trees, edge-labelled in A, modulo strong bisimilarity, where for each label, the set of children is finite respectively countable. The cofree comonad of $(\mathscr{P}_f-)^A$ respectively $(\mathscr{P}_c-)^A$, applied to a set X, consist of all trees as above, node-labelled in X.

Example 4. A *complete monoid* is a (necessarily commutative) monoid M together with an infinitary sum operation consistent with the finite sum [7]. Define the functor $\mathscr{M}\colon \mathsf{Set} \to \mathsf{Set}$ by $\mathscr{M}(X) = \{\varphi \mid \varphi\colon X \to M\}$ and, for $f\colon X \to Y$, $\mathscr{M}(h)(\varphi) = \lambda y.\sum_{x \in f^{-1}(y)} \varphi(x)$. A *weighted transition system* over a set of labels A is a coalgebra $f\colon X \to (\mathscr{M}X)^A$. Similar to the case of labelled transition systems, we obtain weighted transition systems whose branching is countable for each label as coalgebras for the functor $(\mathscr{M}_c-)^A$, where $\mathscr{M}_c$ is defined by $\mathscr{M}_c(X) = \{\varphi\colon X \to M \mid \varphi(x) \neq 0 \text{ for countably many } x \in X\}$. We note that this only requires a countable sum on M to be well-defined and, by further restricting to finite support, weighted transition systems are defined for any commutative monoid (see, e.g., [14]). Labelled transition systems are retrieved by taking the monoid with two elements and logical disjunction as sum. Another example arises by taking the monoid $M = \mathbb{R}^+ \cup \{\infty\}$ of non-negative reals extended with a top element ∞, with the supremum operation.

2.3 GSOS, coGSOS and distributive laws

Given a signature, a *GSOS* rule [5] σ of arity n is of the form

$$\frac{\{x_{i_j} \xrightarrow{a_j} y_j\}_{j=1..m} \qquad \{x_{i_k} \overset{b_k}{\not\to}\}_{k=1..l}}{\sigma(x_1,\ldots,x_n) \xrightarrow{c} t} \tag{4}$$

where m and l are the number of positive and negative premises respectively; $a_1,\ldots,a_m,b_1,\ldots,b_l,c \in A$ are labels; $x_1,\ldots,x_n$, $y_1,\ldots,y_m$ are pairwise distinct variables, and t is a term over these variables. An *abstract GSOS specification* is a natural transformation of the form

$$\Sigma(B \times \mathsf{Id}) \Rightarrow B\Sigma^*.$$

As first observed in [22], specifications in the GSOS format are generalised by abstract GSOS specifications, where Σ models the signature and $BX = (\mathscr{P}_f X)^A$.

A *safe ntree* rule (as taken from [15]) for σ is of the form $\frac{\{z_i \xrightarrow{a_i} y_i\}_{i\in I} \quad \{w_j \overset{b_j}{\not\to}\}_{j\in J}}{\sigma(x_1,\ldots,x_n) \xrightarrow{c} t}$ where I and J are countable possibly infinite sets, the z_i, y_i, w_j, x_k are variables, and $b_j, c, a_i \in A$; the x_k and y_i are all distinct and they are the only variables that occur in the rule; the dependency graph of premise variables (where positive premises are seen as directed edges) is well-founded, and t is either a variable or a term built of a single operator from the signature and the variables. A *coGSOS specification* is a natural transformation of the form

$$\Sigma B^\infty \Rightarrow B(\Sigma + \mathsf{Id}).$$

As stated in [22], every safe ntree specification induces a coGSOS specification where Σ models the signature and $BX = (\mathscr{P}_f X)^A$.

A *distributive law* of a monad $\mathscr{T} = (T, \eta, \mu)$ over a comonad $\mathscr{D} = (D, \varepsilon, \delta)$ is a natural transformation $\lambda\colon TD \Rightarrow DT$ so that $\lambda \circ D\eta = \eta D$, $\varepsilon T \circ \lambda = T\varepsilon$, $\lambda \circ \mu T = D\mu \circ \lambda T \circ T\lambda$ and $D\lambda \circ \lambda D \circ T\delta = \delta T \circ \lambda$. A *$\lambda$-bialgebra* is a triple (X, f, g) where X is a set, f is an EM-algebra for $\mathscr{T}$ and g is an EM-coalgebra for $\mathscr{D}$, such that $g \circ f = Df \circ \lambda_X \circ Tg$.

Every distributive law λ induces, by initiality, a unique coalgebra $h\colon T\emptyset \to DT\emptyset$ such that $(T\emptyset, \mu_\emptyset, h)$ is λ-bialgebra. If $\mathscr{D}$ is the cofree comonad for B, then h is the coinductive extension of a B-coalgebra $m\colon T\emptyset \to BT\emptyset$, which we call the *operational model* of λ. Behavioural equivalence on the operational model is a congruence. This result applies in particular to abstract GSOS and coGSOS specifications, which both extend to distributive laws of monad over comonad.

A *lifting* of a functor $T\colon \mathsf{Set} \to \mathsf{Set}$ to $\mathsf{CoAlg}(\mathscr{D})$ is a functor $\overline{T}$ making the following commute:

$$\begin{array}{ccc} \mathsf{CoAlg}(\mathscr{D}) & \xrightarrow{\overline{T}} & \mathsf{CoAlg}(\mathscr{D}) \\ \downarrow & & \downarrow \\ \mathsf{Set} & \xrightarrow{T} & \mathsf{Set} \end{array}$$

where the vertical arrows represent the forgetful functor, sending a coalgebra to its carrier. Further, a monad $(\overline{T}, \overline{\eta}, \overline{\mu})$ on $\mathsf{CoAlg}(\mathscr{D})$ is a lifting of a monad $\mathscr{T} = (T, \eta, \mu)$ on Set if $\overline{T}$ is a lifting of T, $U\overline{\eta} = \eta U$ and $U\overline{\mu} = \mu U$. A lifting of $\mathscr{T}$ to $\mathsf{coalg}(B)$ is defined similarly.

Distributive laws of $\mathscr{T}$ over $\mathscr{D}$ are in one-to-one correspondence with liftings of (T, η, μ) to $\mathsf{CoAlg}(\mathscr{D})$ (see [13, 22]). If $\mathscr{D}$ is the cofree comonad for B, then $\mathsf{CoAlg}(\mathscr{D}) \cong \mathsf{coalg}(B)$, hence a further equivalent condition is that $\mathscr{T}$ lifts to $\mathsf{coalg}(B)$. In that case, the operational model of a distributive law can be retrieved by applying the corresponding lifting to the unique coalgebra $!\colon \emptyset \to B\emptyset$.

3 Similarity

In this section, we recall the notion of *simulations* of coalgebras from [10], and prove a few basic results concerning the similarity preorder on final coalgebras.

An *ordered functor* is a pair $(B, \sqsubseteq)$ of functors $B\colon \mathsf{Set} \to \mathsf{Set}$ and $\sqsubseteq\colon \mathsf{Set} \to \mathsf{PreOrd}$ such that

$$\begin{array}{ccc} & & \mathsf{PreOrd} \\ & \overset{\sqsubseteq}{\nearrow} & \downarrow \\ \mathsf{Set} & \xrightarrow{B} & \mathsf{Set} \end{array}$$

commutes, where the arrow from PreOrd to Set is the forgetful functor. Thus, given an ordered functor, there is a preorder $\sqsubseteq_{BX} \subseteq BX \times BX$ for any set X, and for any map $f\colon X \to Y$, Bf is monotone.

The (canonical) *relation lifting* of B is defined on a relation $R \subseteq X \times Y$ by

$$\mathsf{Rel}(B)(R) = \{(b,c) \in BX \times BY \mid \exists d \in BR.\, B\pi_1(d) = b \text{ and } B\pi_2(d) = c\}\,.$$

For a detailed account of relation lifting, see, e.g., [11]. Let $(B, \sqsubseteq)$ be an ordered functor. The *lax relation lifting* $\mathsf{Rel}_\sqsubseteq$ is defined as follows:

$$\mathsf{Rel}_\sqsubseteq(B)(R \subseteq X \times Y) = \sqsubseteq_{BX} \circ \mathsf{Rel}(B)(R) \circ \sqsubseteq_{BY}\,.$$

Let (X, f) and (Y, g) be B-coalgebras. A relation $R \subseteq X \times Y$ is a *simulation* (between f and g) if $R \subseteq (f \times g)^{-1}(\mathsf{Rel}_\sqsubseteq(B)(R))$. The greatest simulation between coalgebras f and g is called *similarity*, denoted by $\lesssim^g_f$, or $\lesssim_f$ if $f = g$, or simply $\lesssim$ if f and g are clear from the context.

Given a set X and an ordered functor $(B, \sqsubseteq)$, we define the ordered functor $(B \times X, \widetilde{\sqsubseteq})$ by

$$(b,x)\,\widetilde{\sqsubseteq}_{BX}\,(c,y) \quad \text{iff} \quad b \sqsubseteq_{BX} c \text{ and } x = y\,.$$

The induced notion of simulation can naturally be expressed in terms of the original one:

Lemma 1. *Let $\lesssim$ be the similarity relation between coalgebras $\langle f, f'\rangle\colon X \to BX \times Z$ and $\langle g, g'\rangle\colon X \to BX \times Z$. Then for any relation $R \subseteq X \times X$, we have $R \subseteq (\langle f, f'\rangle \times \langle g, g'\rangle)^{-1}(\mathsf{Rel}_{\widetilde{\sqsubseteq}}(B \times Z)(R))$ iff $R \subseteq (f \times g)^{-1}(\mathsf{Rel}_\sqsubseteq(B)(R))$ and for all $(x,y) \in R$: $f'(x) = g'(x)$.*

Given an ordered functor $(B, \sqsubseteq)$ we write

$$\lesssim_{B^\infty X}$$

for the similarity order induced by $(B \times X, \widetilde{\sqsubseteq})$ on the cofree coalgebra $(B^\infty X, \langle \theta_X, \varepsilon_X\rangle)$. We discuss a few examples of ordered functors and similarity—see [10] for many more.

Example 5. For the functor $L_f X = (\mathscr{P}_f X)^A$ ordered by (pointwise) subset inclusion, a simulation as defined above is a (strong) simulation in the standard sense. For elements $p, q \in L^\infty_f X$, we have $p \lesssim_{L^\infty_f X} q$ iff there exists a (strong) simulation between the underlying trees of p and q, so that related pairs agree on labels in X.

Example 6. For any $G\colon \mathsf{Set} \to \mathsf{Set}$, the functor $B = G + 1$, where $1 = \{\bot\}$, can be ordered as follows: $x \leq y$ iff $x = \bot$ or $x = y$, for all $x, y \in BX$. If $G = A \times \mathsf{Id}$ then $B^\infty X$ consists of finite and infinite sequences of the form $x_0 \xrightarrow{a_0} x_1 \xrightarrow{a_1} x_2 \xrightarrow{a_2} \ldots$ with $x_i \in X$ and $a_i \in A$ for each i (cf. Example 2). For $\sigma, \tau \in B^\infty X$ we have $\sigma \lesssim_{B^\infty X} \tau$ if σ does not terminate before τ does, and σ and τ agree on labels in X and A on each position where σ is defined.

Lemma 2. *Coalgebra homomorphisms h, k preserve similarity: if $x \lesssim y$ then $h(x) \lesssim k(y)$.*

In the remainder of this section we state a few technical properties concerning similarity on cofree comonads, which will be necessary in the following sections. The proofs use Lemma 2 and a few basic, standard properties of relation lifting.

Pointwise inequality of coalgebras implies pointwise similarity of coinductive extensions:

Lemma 3. *Let $(B, \sqsubseteq)$ be an ordered functor, and let f and g be B-coalgebras on a common carrier X. If $(f \times g)(\Delta_X) \subseteq \sqsubseteq_{BX}$ then $(f^\infty \times g^\infty)(\Delta_X) \subseteq \lesssim_{B^\infty X}$.*

Recall from Section 2 that any B-homomorphism yields a B^∞-homomorphism between coinductive extensions. A similar fact holds for inequalities.

Lemma 4. *Let $(B, \sqsubseteq)$ be an ordered functor where B preserves weak pullbacks, and let $f\colon X \to BX$, $g\colon Y \to BY$ and $h\colon X \to Y$.*

- *If $Bh \circ f \sqsubseteq_{BY} g \circ h$ then $B^\infty h \circ f^\infty \lesssim_{B^\infty Y} g^\infty \circ h$, and conversely,*
- *if $Bh \circ f \sqsupseteq_{BY} g \circ h$ then $B^\infty h \circ f^\infty \gtrsim_{B^\infty Y} g^\infty \circ h$.*

4 Monotone biGSOS specifications

As discussed in the introduction, GSOS and coGSOS have a straightforward common generalisation, called *biGSOS* specifications. Throughout this section we assume $(B,\sqsubseteq)$ is an ordered functor, B has a cofree comonad and Σ has a free monad.

Definition 1. A *biGSOS specification* is a natural transformation of the form $\rho\colon \Sigma B^\infty \Rightarrow B\Sigma^*$. A triple (X,a,f) consisting of a set X, an algebra $a\colon \Sigma X \to X$ and a coalgebra $f\colon X \to BX$ (i.e., a bialgebra) is called a ρ*-model* if the following diagram commutes:

$$\begin{array}{ccccc} \Sigma X & & \xrightarrow{\quad a \quad} & & X \\ {\scriptstyle \Sigma f^\infty}\downarrow & & & & \downarrow{\scriptstyle f} \\ \Sigma B^\infty X & \xrightarrow{\rho_X} & B\Sigma^* X & \xrightarrow{Ba^*} & BX \end{array}$$

If $BX = (\mathscr{P}_f X)^A$, then one can obtain biGSOS specifications from concrete rules in the *ntree* format, which combines GSOS and safe ntree, allowing lookahead in premises, negative premises and complex terms in conclusions.

Of particular interest are ρ-models on the initial algebra $\iota_\emptyset\colon \Sigma\Sigma^*\emptyset \to \Sigma^*\emptyset$:

$$\begin{array}{ccccc} \Sigma\Sigma^*\emptyset & & \xrightarrow{\quad \iota_\emptyset \quad} & & \Sigma^*\emptyset \\ {\scriptstyle \Sigma f^\infty}\downarrow & & & & \downarrow{\scriptstyle f} \\ \Sigma B^\infty\Sigma^*\emptyset & \xrightarrow{\rho_{\Sigma^*\emptyset}} & B\Sigma^*\Sigma^*\emptyset & \xrightarrow{B\mu_\emptyset} & B\Sigma^*\emptyset \end{array} \tag{5}$$

(Notice that $\iota_\emptyset^* = \mu_\emptyset$.) We call these *supported models*. Indeed, for labelled transition systems, this notion coincides with the standard notion of the supported model of an SOS specification (e.g., [1]).

In the introduction, we have seen that biGSOS specifications do not necessarily induce a supported model. But even if they do, such a model is not necessarily unique, and behavioural equivalence is not even a congruence, in general, as shown by the following example.

Example 7. In this example we consider a signature with constants c and d, and unary operators σ and τ. Consider the specification (represented by concrete rules) on labelled transition systems where c and d are not assigned any behaviour, and σ and τ are given by the following rules:

$$\frac{x \xrightarrow{a} x' \qquad x' \xrightarrow{a} x''}{\sigma(x) \xrightarrow{a} x''} \qquad \frac{}{\tau(x) \xrightarrow{a} \sigma(\tau(x))}$$

The behaviour of $\tau(x)$ is independent of its argument x. Which transitions can occur in a supported model? First, for any t there is a transition $\tau(t) \xrightarrow{a} \sigma(\tau(t))$. Moreover, a transition $\sigma(\tau(t)) \xrightarrow{a} t''$ *can* be in the model, although it does not need to be. But if it is there, it is supported by an *infinite* proof.

In fact, one can easily construct a model in which the behaviour of $\sigma(\tau(c))$ is different from that of $\sigma(\tau(d))$—for example, a model where $\sigma(\tau(c))$ does not make any transitions, whereas $\sigma(\tau(d)) \xrightarrow{a} t$ for some t. Then behavioural equivalence is not a congruence; c is bisimilar to d, but $\sigma(\tau(c))$ is not bisimilar to $\sigma(\tau(d))$.

The above example features a specification that has many different interpretations as a supported model. However, there is only one which makes sense: the *least* model, which only features *finite* proofs. It is sensible to speak about the least model of this specification, since it does not contain any negative premises. More generally, absence of negative premises can be defined based on an ordered functor and the induced similarity order.

Definition 2. A biGSOS specification $\rho\colon \Sigma B^\infty \Rightarrow B\Sigma^*$ is *monotone* if the restriction of $\rho_X \times \rho_X$ to $\mathsf{Rel}(\Sigma)(\lesssim_{B^\infty X})$ corestricts to $\sqsubseteq_{B\Sigma^* X}$, for any set X.

If Σ represents an algebraic signature, then monotonicity can be conveniently restated as follows (c.f. [6], where monotone GSOS is characterised in a similar way). For every operator σ:

$$\frac{b_1 \lesssim_{B^\infty X} c_1 \quad \ldots \quad b_n \lesssim_{B^\infty X} c_n}{\rho_X(\sigma(b_1,\ldots,b_n)) \sqsubseteq_{B\Sigma^* X} \rho_X(\sigma(c_1,\ldots,c_n))}$$

for every set X and every $b_1,\ldots,b_n,c_1,\ldots,c_n \in B^\infty X$. Thus, in a monotone specification, if c_i simulates b_i for each i, then the behaviour of $\sigma(b_1,\ldots,b_n)$ is "less than" the behaviour of $\sigma(c_1,\ldots,c_n)$.

In the case of labelled transition systems, it is straightforward that monotonicity rules out (non-trivial use of) negative premises. Notice that the example specification in the introduction consisting of rules (2) and (3), which does not have a model, is not monotone. This is no coincidence: every monotone biGSOS specification has a model, if $B\Sigma^*\emptyset$ is a pointed DCPO, as we will see next. In fact, the proper canonical choice is the *least* model, corresponding to behaviour obtained in finitely many proof steps.

4.1 Models of monotone specifications

Let ρ be a monotone biGSOS specification. Suppose $B\Sigma^*\emptyset$ is a pointed DCPO. Then the set of coalgebras $\mathsf{coalg}(B)_{\Sigma^*\emptyset} = \{f \mid f\colon \Sigma^*\emptyset \to B\Sigma^*\emptyset\}$, ordered pointwise, is a pointed DCPO as well.

Consider the function $\varphi\colon \mathsf{coalg}(B)_{\Sigma^*\emptyset} \to \mathsf{coalg}(B)_{\Sigma^*\emptyset}$, defined as follows:

$$\varphi(f) = B\mu_\emptyset \circ \rho_{\Sigma^*\emptyset} \circ \Sigma f^\infty \circ \iota_\emptyset^{-1} \tag{6}$$

Since $\iota_\emptyset$ is an isomorphism, a function f is a fixed point of φ if and only if it is a supported model of ρ (Equation (5)). We are interested in the *least* supported model. To show that it exists, since $\mathsf{coalg}(B)_{\Sigma^*\emptyset}$ is a pointed DCPO, it suffices to show that φ is monotone.

Lemma 5. *The function φ is monotone.*

Proof. Suppose $f,g\colon \Sigma^*\emptyset \to B\Sigma^*\emptyset$ and $f \sqsubseteq_{B\Sigma^*\emptyset} g$. By Lemma 3, we have $f^\infty \lesssim_{B^\infty\Sigma^*\emptyset} g^\infty$. From standard properties of relation lifting we derive $\Sigma f^\infty\ \mathsf{Rel}(\Sigma)(\lesssim_{B^\infty\Sigma^*\emptyset})\ \Sigma g^\infty$ and now the result follows by monotonicity of ρ (assumption) and monotonicity of $B\mu_\emptyset$ (B is ordered). □

Corollary 1. *If $B\Sigma^*\emptyset$ is a pointed DCPO and ρ is a monotone biGSOS specification, then ρ has a least supported model.*

The condition of the Corollary is satisfied if B is of the form $B = G+1$ (c.f. Example 6), that is, $B = G+1$ for some functor G (where the element in the singleton 1 is interpreted as the least element of the pointed DCPO). Consider, as an example, the functor $BX = A \times X + 1$ of finite and infinite streams over A. Any specification that does not mention termination (i.e., a specification for the functor $GX = A \times X$) yields a monotone specification for B.

Example 8. Consider the following specification (in terms of rules) for the functor $BX = \mathbb{N} \times X + 1$ of (possibly terminating) stream systems over the natural numbers. It specifies a unary operator σ, a binary operator $\oplus$, infinitely many unary operators $m \otimes -$ (one for each $m \in \mathbb{N}$), and constants $ones, pos, c$:

$$\frac{x \xrightarrow{n} x' \qquad x' \xrightarrow{m} x''}{\sigma(x) \xrightarrow{n} n \otimes (m \otimes \sigma(x''))} \qquad \frac{x \xrightarrow{n} x' \qquad y \xrightarrow{m} y'}{x \oplus y \xrightarrow{n+m} x' \oplus y'} \qquad \frac{x \xrightarrow{n} x'}{m \otimes x \xrightarrow{m \times n} m \otimes x'}$$

$$\frac{}{ones \xrightarrow{1} ones} \qquad \frac{}{pos \xrightarrow{1} ones \oplus pos} \qquad \frac{}{c \xrightarrow{1} \sigma(c)}$$

where $+$ and $\times$ denote addition and multiplication of natural numbers, respectively. This induces a monotone biGSOS specification; the rule for σ is GSOS nor coGSOS, since it uses both lookahead and a complex conclusion. By the above Corollary, it has a model. The coinductive extension maps *pos* to the increasing stream of positive integers, and $\sigma(pos)$ is the stream $(1,6,120,\ldots) = (1!,3!,5!,\ldots)$. But c does not represent an infinite stream, since $\sigma(c)$ is undefined.

The case of *labelled transition systems* is a bit more subtle. The problem is that $(\mathscr{P}_f\Sigma^*\emptyset)^A$ and $(\mathscr{P}_c\Sigma^*\emptyset)^A$ are not DCPOs, in general, whereas the functor $(\mathscr{P}-)^A$ does not have a cofree comonad. However, if the set of closed terms $\Sigma^*\emptyset$ is countable, then $(\mathscr{P}_c\Sigma^*\emptyset)^A$ is a pointed DCPO, and thus Corollary 1 applies. The specification in Example 7 can be viewed as a specification for the functor $(\mathscr{P}_c-)^A$, and it has a countable set of terms. Therefore it has, by the Corollary, a least supported model. In this model, the behaviour of $\sigma(t)$ is empty, for any $t \in \Sigma^*\emptyset$.

5 Distributive laws for biGSOS specifications

In the previous section we have seen how to construct a least supported model of a monotone biGSOS specification, as the least fixed point of a monotone function. In the present section we show that, given a monotone biGSOS specification, the construction of a least model generalizes to a *lifting* of the free monad Σ^* to the category of B-coalgebras. It then immediately follows that there exists a canonical distributive law of the monad Σ^* over the comonad B^∞, and that the (unique) operational model of this distributive law corresponds to the least supported model as constructed above.

In order to proceed we define a $\mathsf{DCPO}_\perp$*-ordered functor* as an ordered functor (Section 3) where PreOrd is replaced by $\mathsf{DCPO}_\perp$. Below we assume that $(B,\sqsubseteq)$ is $\mathsf{DCPO}_\perp$-ordered, and Σ and B are as before (having a free monad and cofree comonad respectively).

Example 9. A general class of functors that are $\mathsf{DCPO}_\perp$-ordered are those of the form $B+1$, where the singleton 1 is interpreted as the least element and all other distinct elements are incomparable (see Example 6). Another example is the functor $(\mathscr{P}-)^A$ of labelled transition systems with arbitrary branching, but this example can not be treated here because there exists no cofree comonad for it. The case of labelled transition systems is treated in Section 6.

Let $\mathsf{coalg}(B)_{\Sigma^*X}$ be the set of B-coalgebras with carrier Σ^*X, pointwise ordered as a DCPO by the order on B. The lifting of Σ^* to $\mathsf{coalg}(B)$ that we are about to define maps a coalgebra $c\colon X \to BX$ to the least coalgebra $\overline{c}\colon \Sigma^*X \to B\Sigma^*X$, w.r.t. the above order on $\mathsf{coalg}(B)_{\Sigma^*X}$, making the following diagram commute.

$$\begin{array}{ccccccc}
\Sigma B^\infty\Sigma^*X & \xrightarrow{\rho_{\Sigma^*X}} & B\Sigma^*\Sigma^*X & \xrightarrow{B\mu_X} & B\Sigma^*X & \xleftarrow{B\eta_X} & BX \\
{\scriptstyle \Sigma(\overline{c})^\infty}\uparrow & & & & \uparrow{\scriptstyle \overline{c}} & & \uparrow{\scriptstyle c} \\
\Sigma\Sigma^*X & & \xrightarrow[\iota_X]{} & & \Sigma^*X & \xleftarrow[\eta_X]{} & X
\end{array}$$

Equivalently, $\overline{c}$ is the least fixed point of the operator $\varphi_c\colon \mathsf{coalg}(B)_{\Sigma^*X} \to \mathsf{coalg}(B)_{\Sigma^*X}$ defined by

$$\varphi_c(f) = [B\mu_X \circ \rho_{\Sigma^*\emptyset} \circ \Sigma f^\infty, B\eta_X \circ c] \circ [\iota_X, \eta_X]^{-1}.$$

Following the proof of Lemma 5 it is easy to verify:

Lemma 6. *For any $c\colon X \to BX$, the function φ_c is monotone.*

For the lifting of Σ^*, we need to show that the above construction preserves coalgebra morphisms.

Theorem 1. *The functor* $\overline{\Sigma^*}\colon \mathsf{coalg}(B) \to \mathsf{coalg}(B)$ *defined by*

$$\overline{\Sigma^*}(X,c) = (\Sigma^* X, \overline{c}) \qquad \textit{and} \qquad \overline{\Sigma^*}(h) = \Sigma^* h$$

is a lifting of the functor Σ^*.

Proof. Let (X,c) and (Y,d) be $B\Sigma^*$-coalgebras. We need to prove that, if $h\colon X \to Y$ is a coalgebra homomorphism from c to d, then $\Sigma^* h$ is a homomorphism from $\overline{c}$ to $\overline{d}$.

The proof is by transfinite induction on the iterative construction of $\overline{c}$ and $\overline{d}$ as limits of the ordinal-indexed initial chains of φ_c and φ_d respectively. For the limit (and base) case, given a (possibly empty) directed family of coalgebras $f_i\colon \Sigma^* X \to B\Sigma^* X$ and another directed family $g_i\colon \Sigma^* Y \to B\Sigma^* Y$, such that $B\Sigma^* h \circ f_i = g_i \circ \Sigma^* h$ for all i, we have $B\Sigma^* h \circ \bigvee_i f_i = \bigvee_i (B\Sigma^* h \circ f_i) = \bigvee_i (g_i \circ \Sigma^* h) = (\bigvee_i g_i) \circ \Sigma^* h$ by continuity of $B\Sigma^* h$ and assumption.

Let $f\colon \Sigma^* X \to B\Sigma^* X$ and $g\colon \Sigma^* Y \to B\Sigma^* Y$ be such that $B\Sigma^* h \circ f = g \circ \Sigma^* h$. To prove: $B\Sigma^* h \circ \varphi_c(f) = \varphi_d(g) \circ \Sigma^* h$, i.e., commutativity of the outside of:

$$\begin{array}{ccccccccc}
\Sigma^* X & \xrightarrow{[\iota_X,\eta_X]^{-1}} & \Sigma\Sigma^* X + X & \xrightarrow{\Sigma f^\infty + c} & \Sigma B^\infty \Sigma^* X + BX & \xrightarrow{\rho_{\Sigma^* X} + \mathsf{id}} & B\Sigma^*\Sigma^* X + BX & \xrightarrow{[B\mu_X, B\eta_X]} & B\Sigma^* X \\
\downarrow{\scriptstyle \Sigma^* h} & & \downarrow{\scriptstyle \Sigma\Sigma^* h + h} & & \downarrow{\scriptstyle \Sigma B^\infty \Sigma^* h + Bh} & & \downarrow{\scriptstyle B\Sigma^*\Sigma^* h + Bh} & & \downarrow{\scriptstyle B\Sigma^* h} \\
\Sigma^* Y & \xrightarrow[[\iota_Y,\eta_Y]^{-1}]{} & \Sigma\Sigma^* Y + Y & \xrightarrow[\Sigma g^\infty + d]{} & \Sigma B^\infty \Sigma^* Y + BY & \xrightarrow[\rho_{\Sigma^* Y} + \mathsf{id}]{} & B\Sigma^*\Sigma^* Y + BY & \xrightarrow[[B\mu_Y, B\eta_Y]]{} & B\Sigma^* Y
\end{array}$$

From left to right, the first square commutes by naturality of $[\iota,\eta]$ (and the fact that it is an isomorphism), the second by assumption that $\Sigma^* h$ is a B-coalgebra homomorphism from f to g (and therefore a B^∞-coalgebra homomorphism) and the assumption that h is a coalgebra homomorphism from c to d, the third by naturality of ρ, and the fourth by naturality of μ and η. □

We show that the (free) *monad* (Σ^*, η, μ) lifts to $\mathsf{coalg}(B)$. This is the heart of the matter. The main proof obligation is to show that μ_X is a coalgebra homomorphism from $\overline{\Sigma^*}(\overline{\Sigma^*}(X,c))$ to $\overline{\Sigma^*}(X,c)$, for any B-coalgebra (X,c).

Theorem 2. *The* monad (Σ^*, η, μ) *on* Set *lifts to the monad* $(\overline{\Sigma^*}, \eta, \mu)$ *on* $\mathsf{coalg}(B)$*, if* B *preserves weak pullbacks.*

The lifting gives rise to a distributive law of monad over comonad.

Theorem 3. *Let* $\rho : \Sigma B^\infty \Rightarrow B\Sigma^*$ *be a monotone biGSOS specification, where* B *is* $\mathsf{DCPO}_\perp$*-ordered and preserves weak pullbacks. There exists a distributive law* $\lambda : \Sigma^* B^\infty \Rightarrow B^\infty \Sigma^*$ *of the free monad* Σ^* *over the cofree comonad* B^∞ *such that the operational model of* λ *is the least supported model of* ρ*.*

Proof. By Theorem 2, we obtain a lifting of (Σ^*, η, μ) to $\mathsf{coalg}(B)$. As explained in the preliminaries, such a lifting corresponds uniquely to a distributive law of the desired type. The operational model of λ is obtained by applying the lifting to the unique coalgebra $!\colon \emptyset \to B\emptyset$. But that coincides, by definition of the lifting, with the least supported model as defined in Section 4. □

It follows from the general theory of bialgebras that the unique coalgebra morphism from the least supported model to the final coalgebra is an algebra homomorphism, i.e., behavioural equivalence on the least supported model of a monotone biGSOS specification is a congruence.

Labelled transition systems The results above do not apply to labelled transition systems. The problem is that the cofree comonad for the functor $(\mathscr{P}-)^A$ does not exist. A first attempt would be to restrict to the finitely branching transition systems, i.e., coalgebras for the functor $(\mathscr{P}_f-)^A$. But this functor is not $\mathsf{DCPO}_\perp$-ordered, and indeed, contrary to the case of GSOS and coGSOS, even with a finite biGSOS specification one can easily generate a least model with infinite branching, so that a lifting as in the previous section can not exist.

Example 10. Consider the following specification on (finitely branching) labelled transition systems, involving a unary operator σ and a constant c:

$$\frac{}{c \xrightarrow{a} \sigma(c)} \qquad \frac{}{\sigma(x) \xrightarrow{a} \sigma(\sigma(x))} \qquad \frac{x \xrightarrow{a} x' \xrightarrow{a} x'' \xrightarrow{a} x'''}{\sigma(x) \xrightarrow{a} x'''}$$

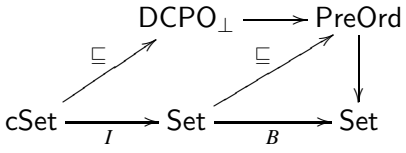
The left rule for σ constructs an infinite chain of transitions from $\sigma(x)$ for any x, so in particular for $\sigma(c)$. The right rule takes the transitive closure of transitions from $\sigma(c)$, so in the least model there are infinitely many transitions from $\sigma(c)$.

The model in the above example has countable branching. One might ask whether it can be adapted to generate uncountable branching, i.e., that we can construct a biGSOS specification for the functor $(\mathscr{P}_c-)^A$, such that the model of this specification would feature uncountable branching. However, as it turns out, this is not the case, at least if we assume Σ to be a polynomial functor (a countable coproduct of finite products, modelling a signature with countably many operations each of finite arity), and the set of labels A to be countable. This is shown more generally in the next section.

6 Liftings for countably accessible functors

In the previous section, we have seen that one of the most important instances of the framework—the case of labelled transition systems—does not work, because of size issues: the functors in question either do not have a cofree comonad, or are not DCPO-ordered. In the current section, we solve this problem by showing that, if both functors B, Σ are reasonably well-behaved, then it suffices to have a DCPO-ordering of B only on *countable* sets.

More precisely, let cSet be the full subcategory of countable sets, with inclusion $I\colon \mathsf{cSet} \to \mathsf{Set}$. We assume that $(B, \sqsubseteq)$ is an ordered functor on Set, and that its restriction to countable sets is $\mathsf{DCPO}_\perp$-ordered:



This is a weaker assumption than in Section 5: before, every set BX was assumed to be a pointed DCPO, whereas here, they only need to be pointed DCPOs when X is countable (and just a preorder otherwise).

Example 11. The functor $(\mathscr{P}_c-)^A$ coincides with the $\mathsf{DCPO}_\perp$-ordered functor $(\mathscr{P}-)^A$ when restricted to countable sets, hence it satisfies the above assumption. Notice that $(\mathscr{P}_c-)^A$ is not $\mathsf{DCPO}_\perp$-ordered. The functor $(\mathscr{P}_f-)^A$ does not satisfy the above assumption.

The functor $(\mathscr{M}-)^A$, for the complete monoid $\mathbb{R}^+ \cup \{\infty\}$ (Example 4), is ordered as a complete lattice [19], so also $\mathsf{DCPO}_\perp$-ordered. Similar to the above, the functor $(\mathscr{M}_c-)^A$ is $\mathsf{DCPO}_\perp$-ordered when restricted to countable sets, i.e., satisfies the above assumption.

We define $\mathsf{coalg}_c(B)$ to be the full subcategory of B-coalgebras whose carrier is a countable set, with inclusion $\overline{I}\colon \mathsf{coalg}_c(B) \to \mathsf{coalg}(B)$. The associated forgetful functor is denoted by $U\colon \mathsf{coalg}_c(B) \to \mathsf{cSet}$.

The pointed DCPO structure on each BX, for X countable, suffices to carry out the fixed point constructions from the previous sections for coalgebras over countable sets, if we assume that Σ^* preserves countable sets. Notice, moreover, that the (partial) order on the functor B is still necessary to define the simulation order on $B^\infty X$, and hence speak about monotonicity of biGSOS specifications. The proof of the following theorem is essentially the same as in the previous section.

Theorem 4. *Suppose Σ^* preserves countable sets, and B is an ordered functor which preserves weak pullbacks and whose restriction to cSet is $\mathsf{DCPO}_\perp$-ordered. Let $(\Sigma_c^*, \eta^c, \mu^c)$ be the restriction of (Σ^*, η, μ) to cSet. Any monotone biGSOS specification $\rho\colon \Sigma B^\infty \Rightarrow B\Sigma^*$ gives rise to a lifting $(\overline{\Sigma^*}_c, \overline{\eta}^c, \overline{\mu}^c)$ of the monad $(\Sigma_c^*, \eta^c, \mu^c)$ to $\mathsf{coalg}_c(B)$.*

In the remainder of this section, we will show that, under certain assumptions on B and Σ^*, the above lifting extends to a lifting of the monad Σ^* from Set to $\mathsf{coalg}(B)$, and hence a distributive law of the monad Σ^* over the cofree comonad B^∞. It relies on the fact that, under certain conditions, we can present every coalgebra as a (filtered) colimit of coalgebras over countable sets.

We use the theory of locally (countably, i.e., ω_1-) presentable categories and (countably) accessible categories. Because of space limits we can not properly recall that theory in detail here (see [3]); we only recall a concrete characterisation of when a functor on Set is countably accessible, since that will be assumed both for B and Σ^* later on. On Set, a functor $B\colon \mathsf{Set} \to \mathsf{Set}$ is *countably accessible* if for every set X and element $x \in BX$, there is an injective function $i\colon Y \to X$ from a finite set Y and an element $y \in BY$ such that $Bi(y) = x$. Intuitively, such functors are determined by how they operate on countable sets.

Example 12. Any finitary functor is countably accessible. Further, the functors $(\mathscr{P}_c -)^A$ and $(\mathscr{M}_c -)^A$ (c.f. Example 11) are countably accessible if A is countable.

A functor is called *strongly countably accessible* if it is countably accessible and additionally preserves countable sets, i.e., it restricts to a functor $\mathsf{cSet} \to \mathsf{cSet}$. We will assume this for our "syntax" functor Σ^*. If Σ correponds to a signature with countably many operations each of finite arity (so is a countable coproduct of finite products) then Σ^* is strongly countably accessible.

The central idea of obtaining a lifting to $\mathsf{coalg}(B)$ from a lifting to $\mathsf{coalg}_c(B)$ is to *extend* the monad on $\mathsf{coalg}_c(B)$ along the inclusion $\overline{I}\colon \mathsf{coalg}_c(B) \to \mathsf{coalg}(B)$. Concretely, a functor $T\colon \mathsf{Set} \to \mathsf{Set}$ extends $T_c\colon \mathsf{cSet} \to \mathsf{cSet}$ if there is a natural isomorphism $\alpha\colon IT_c \Rightarrow TI$. A monad (T, η, μ) on Set extends a monad (T_c, η_c, μ_c) on cSet if T_c extends T with some isomorphism α such that $\alpha \circ I\eta_c = \eta I$ and $\alpha \circ I\mu_c = \mu I \circ T\alpha \circ \alpha T_c$. This notion of extension is generalised naturally to arbitrary locally countably presentable categories. Monads on the category of countably presentable objects can always be extended.

Lemma 7. *Let $\mathscr{C}$ be a locally countably presentable category, with $I\colon \mathscr{C}_c \to \mathscr{C}$ the subcategory of countably presentable objects. Any monad (T_c, η^c, μ^c) on $\mathscr{C}_c$ extends uniquely to a monad (T, η, μ) on $\mathscr{C}$, along $I\colon \mathscr{C}_c \to \mathscr{C}$.*

Since B is countably accessible, $\mathsf{coalg}(B)$ is locally countably presentable and $\mathsf{coalg}_c(B)$ is the associated category of countably presentable objects [2]. This means every B-coalgebra can be presented as a filtered colimit of B-coalgebras with countable carriers. The above lemma applies, so we can extend the monad on $\mathsf{coalg}_c(B)$ of Theorem 4 to a monad on $\mathsf{coalg}(B)$, resulting in Theorem 6 below. The latter relies on Theorem 5, which ensures that, doing so, we will get a lifting of the monad on Set that we started with.

In the remainder of this section, we will consider a slightly relaxed version of functor liftings, up to isomorphism, similar to extensions defined before. This is harmless—those still correspond to distributive laws—but since the monad on $\mathsf{coalg}(B)$ is constructed only up to isomorphism, it is more natural to work with in this setting. We say $(\overline{T},\overline{\eta},\overline{\mu})$ lifts (T,η,μ) (up to isomorphism) if there is a natural isomorphism $\alpha\colon U\overline{T} \Rightarrow TU$ such that $\alpha \circ U\overline{\eta} = \eta U$ and $\alpha \circ U\overline{\mu} = \mu U \circ T\alpha \circ \alpha\overline{T}$.

Theorem 5. *Let $B\colon \mathsf{Set} \to \mathsf{Set}$ be countably accessible. Suppose (T_c,η^c,μ^c) is a monad on cSet, which lifts to a monad $(\overline{T}_c,\overline{\eta}^c,\overline{\mu}^c)$ on $\mathsf{coalg}_c(B)$. Then*

1. *(T_c,η^c,μ^c) extends to (T,η,μ) along $I\colon \mathsf{Set}_c \to \mathsf{Set}$,*
2. *$(\overline{T}_c,\overline{\eta}^c,\overline{\mu}^c)$ extends to $(\overline{T},\overline{\eta},\overline{\mu})$ along $\overline{I}\colon \mathsf{coalg}_c(B) \to \mathsf{coalg}(B)$,*
3. *$(\overline{T},\overline{\eta},\overline{\mu})$ is a lifting (up to isomorphism) of (T,η,μ).*

By instantiating the above theorem with the lifting of Theorem 4, the third point gives us the desired lifting to $\mathsf{coalg}(B)$. In particular T_c is instantiated to the restriction Σ^*_c of Σ^*, which means that the extension in the first point is just Σ^* itself.

Theorem 6. *Let $\rho\colon \Sigma B^\infty \Rightarrow B\Sigma^*$ be a monotone biGSOS specification, where B is an ordered functor whose restriction to countable sets is $\mathsf{DCPO}_\perp$-ordered, B is countably accessible, B preserves weak pullbacks, and Σ^* is strongly countably accessible. There exists a distributive law $\lambda : \Sigma^*B^\infty \Rightarrow B^\infty\Sigma^*$ of the free monad Σ^* over the cofree comonad B^∞ such that the operational model of λ is the least supported model of ρ.*

As explained in Example 12 and Example 11, if B is either $(\mathscr{P}_c-)^A$ or $(\mathscr{M}_c-)^A$ (weighted in the non-negative real numbers) with A countable, then it satisfies the above hypotheses (that $\mathscr{M}_c$ preserves weak pullbacks follows essentially from [9]). So the above theorem applies to labelled transition systems and weighted transition systems (of the above type) over a countable set of labels, as long as the syntax is composed of countably many operations each with finite arity. Hence, behavioural equivalence on the operational model of any biGSOS specification for such systems is a congruence.

7 Future work

In this paper we provided a bialgebraic foundation of positive specification formats over ordered functors, involving rules that feature lookahead in the premises as well as complex terms in conclusions. From a practical point of view, it would be interesting to find more concrete rules formats corresponding to the abstract format of the present paper. In particular, concrete GSOS formats for weighted transition systems exist [14]; they could be a good starting point.

It is currently unclear to us whether the assumption of weak pullback preservation in the main results is necessary. This assumption is used in our proof of Lemma 4, which in turn is used in the proof that the free monad lifts to the category of coalgebras (Theorem 2). Finally, we would like to study *continuous* specifications, as opposed to specifications that are only monotone, as in the current paper. Continuous specifications should be better behaved than monotone ones. However, it is currently not yet clear how to characterize continuity of a specification both at the concrete, syntactic level.

References

[1] L. Aceto, W. Fokkink & C. Verhoef (2001): *Structural Operational Semantics*. In: *Handbook of Process Algebra*, Elsevier Science, pp. 197–292, doi:10.1016/B978-044482830-9/50021-7.

[2] J. Adámek & H-E. Porst (2004): *On tree coalgebras and coalgebra presentations*. *Theor. Comput. Sci.* 311(1-3), pp. 257–283, doi:10.1016/S0304-3975(03)00378-5.

[3] J. Adámek & J. Rosický (1994): *Locally Presentable and Accessible Categories*. Cambridge Tracts in Mathematics, Cambridge University Press, doi:10.1017/CBO9780511600579.

[4] F. Bartels (2004): *On generalised coinduction and probabilistic specification formats*. Ph.D. thesis, CWI, Amsterdam.

[5] B. Bloom, S. Istrail & A. Meyer (1995): *Bisimulation Can't be Traced*. *J. ACM* 42(1), pp. 232–268, doi:10.1145/200836.200876.

[6] F. Bonchi, D. Petrisan, D. Pous & J. Rot (2017): *A general account of coinduction up-to*. *Acta Inf.* 54(2), pp. 127–190, doi:10.1007/s00236-016-0271-4.

[7] M. Droste & W. Kuich (2009): *Semirings and formal power series*. In: *Handbook of Weighted Automata*, Springer, pp. 3–28, doi:10/bj2xgm.

[8] M. Fiore & S. Staton (2010): *Positive structural operational semantics and monotone distributive laws*. In: *CMCS Short Contributions*, p. 8.

[9] H. P. Gumm & T. Schröder (2001): *Monoid-labeled transition systems*. *Electr. Notes Theor. Comput. Sci.* 44(1), pp. 185–204, doi:10.1016/S1571-0661(04)80908-3.

[10] J. Hughes & B. Jacobs (2004): *Simulations in coalgebra*. *Theor. Comput. Sci.* 327(1-2), pp. 71–108, doi:10.1016/j.tcs.2004.07.022.

[11] B. Jacobs (2016): *Introduction to Coalgebra: Towards Mathematics of States and Observation*. *Cambridge Tracts in Theoretical Computer Science* 59, Cambridge University Press, doi:10.1017/CBO9781316823187.

[12] B. Jacobs & J. Rutten (2011): *An introduction to (co)algebras and (co)induction*. In: *Advanced Topics in Bisimulation and Coinduction*, Cambridge University Press, pp. 38–99, doi:10.1017/CBO9780511792588.003.

[13] P. T. Johnstone (1975): *Adjoint lifting theorems for categories of algebras*. *Bulletin of the London Mathematical Society* 7(3), pp. 294–297, doi:10.1112/blms/7.3.294.

[14] B. Klin (2009): *Structural Operational Semantics for Weighted Transition Systems*. In J. Palsberg, editor: *Semantics and Algebraic Specification*, *LNCS* 5700, Springer, pp. 121–139, doi:10/cxqzcf.

[15] B. Klin (2011): *Bialgebras for structural operational semantics: An introduction*. *TCS* 412(38), pp. 5043–5069, doi:10.1016/j.tcs.2011.03.023.

[16] B. Klin & B. Nachyła (2014): *Distributive Laws and Decidable Properties of SOS Specifications*. In Johannes Borgström & Silvia Crafa, editors: *Proc. EXPRESS/SOS 2014*, *EPTCS* 160, pp. 79–93, doi:10.4204/EPTCS.160.8.

[17] B. Klin & B. Nachyła (2017): *Some undecidable properties of SOS specifications*. *J. Log. Algebr. Meth. Program.* 87, pp. 94–109, doi:10.1016/j.jlamp.2016.08.005.

[18] J. Power & H. Watanabe (2002): *Combining a monad and a comonad*. *Theor. Comput. Sci.* 280(1-2), pp. 137–162, doi:10.1016/S0304-3975(01)00024-X.

[19] J. Rot & M. M. Bonsangue (2016): *Structural congruence for bialgebraic semantics*. *J. Log. Algebr. Meth. Program.* 85(6), pp. 1268–1291, doi:10.1016/j.jlamp.2016.08.001.

[20] J. J. M. M. Rutten (2000): *Universal coalgebra: a theory of systems*. *TCS* 249(1), pp. 3–80. Available at `http://dx.doi.org/10.1016/S0304-3975(00)00056-6`.

[21] S. Staton (2008): *General Structural Operational Semantics through Categorical Logic*. In: *LICS*, IEEE Computer Society, pp. 166–177, doi:10.1109/LICS.2008.43.

[22] D. Turi & G. Plotkin (1997): *Towards a Mathematical Operational Semantics*. In: *LICS*, IEEE Computer Society, pp. 280–291, doi:10.1109/LICS.1997.614955.