Basic Research in Computer Science

BRICS NS-01-6 Aceto & Panangaden (eds.): EXPRESS '01 Preliminary Proceedings

Preliminary Proceedings of the 8th International Workshop on

Expressiveness in Concurrency

EXPRESS '01

Aalborg, Denmark, August 20, 2001

Luca Aceto
Prakash Panangaden
(editors)

BRICS Notes Series **NS-01-6**

ISSN 0909-3206 **August 2001**

See back inner page for a list of recent BRICS Notes Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `NS/01/6/`

Electronic Notes in Theoretical Computer Science

EXPRESS'01

8th International Workshop on Expressiveness in Concurrency
Aalborg, Denmark
August 20, 2001

Guest Editors:
Luca Aceto, BRICS, Aalborg University, Denmark
Prakash Panangaden, McGill University, Canada

Table of Contents

Foreword v

An Automata-Theoretic Approach to the Reachability Analysis of RPPS Systems
Anne Labroue, Philippe Schnoebelen 1

Adequate Sets of Temporal Connectives in CTL
Alan Martin 21

On Logical and Concurrent Equivalences
Julian C. Bradfield, Sybille B. Fröschle 33

Rewrite Systems with Constraints
Jan Strejcek 47

On the Decidability of Fragments of the Asynchronous π-Calculus
Roberto M. Amadio, Charles Meyssonnier 67

Broadcast Calculus Interpreted in CCS upto Bisimulation
K. V. S. Prasad 85

Encoding Distributed Areas and Local Communication into the π-Calculus
Tom Chothia, Ian Stark 103

Turing Machines, Transition Systems, and Interaction
Dina Q. Goldin, Scott A. Smolka, Peter Wegner 123

Foreword

The EXPRESS workshops aim at bringing together researchers interested in the relations between various formal systems, particularly in the field of Concurrency. More specifically, they focus 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, rewrite systems etc.) on the basis of their relative expressive power.

The EXPRESS workshops were originally held as meetings of the HCM project EXPRESS, which was active with the same focus from January 1994 till December 1997. The first three workshops were held respectively in Amsterdam (1994, chaired by Frits Vaandrager), Tarquinia (1995, chaired by Rocco De Nicola), and Dagstuhl (1996, co-chaired by Ursula Goltz and Rocco De Nicola). The workshop in 1997, which took place in Santa Margherita Ligure and was co-chaired by Catuscia Palamidessi and Joachim Parrow, was organized as a conference with a call for papers and a significant attendance from outside the project. The 1998 workshop was held as a satellite workshop of the CONCUR'98 conference in Nice, co-chaired by Ilaria Castellani and Catuscia Palamidessi, and like on that occasion EXPRESS'99 was hosted by the CONCUR'99 conference in Eindhoven, co-chaired by Ilaria Castellani and Björn Victor. The EXPRESS'00 workshop was again held as a satellite workshop of CONCUR 2000, Pennsylvania State University, USA, co-chaired by Luca Aceto and Björn Victor.

This volume contains the *preliminary proceedings* of EXPRESS'01, which was held in Aalborg, Denmark, on 20 August 2001. It includes the eight papers that were selected for presentation by the program committee. The final proceedings will appear as volume 52 in the ENTCS series, which can be found at the URL:

http://www.elsevier.nl/locate/entcs/volume52.html.

We would like to thank the authors of the submitted papers, the invited speakers, and the members of the program committee for their contribution to both the meeting and this volume. Many thanks to Kim G. Larsen and Mogens Nielsen (CONCUR 2001 Conference Chairs), Anna Ingólfsdóttir (Chair of the CONCUR 2001 Organizing Committee) and Hans Hüttel (Satellite Workshops Chair) for the opportunity they gave us to organize EXPRESS'01, and for their continuous support. We would also like to thank Michael Mislove and Uffe Engberg for their great help with the editing of the proceedings. Finally, we gratefully acknowledge the support of BRICS (Basic Research in Computer Science), Centre of the Danish National Research Foundation.

Luca Aceto, BRICS, Aalborg University, Denmark
Prakash Panangaden, McGill University, Canada

EXPRESS'01 - Program Committee

Luca Aceto (DK)
Franck van Breugel (CA)
Ilaria Castellani (FR)
Rance Cleaveland (USA)
Philippa Gardner (GB)
Jan Friso Groote (NL)
Faron Moller (GB)
Prakash Panangaden (CA)
Joachim Parrow (SE)
Julian Rathke (UK)
Roberto Segala (I)

Electronic Notes in Theoretical Computer Science 52 No. 1 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume52.html 20 pages

An Automata-Theoretic Approach to the Reachability Analysis of RPPS Systems

A. Labroue [1] and Ph. Schnoebelen [2]

Laboratoire Spécification & Vérification,
ENS de Cachan & CNRS UMR 8643,
61 av. Pdt. Wilson, F-94235 Cachan Cedex, France

Abstract

We show how the tree-automata techniques proposed by Lugiez and Schnoebelen apply to the reachability analysis of RPPS systems. Using these techniques requires that we express the states of RPPS systems in a tailor-made process rewrite system where reachability is a relation recognizable by finite tree-automata.

Keywords: verification of infinite-state systems, process algebra, reachability analysis, tree automata, model checking.

1 Introduction

This paper is concerned with the verification of RPPS systems (for *Recursive Parallel Program Schemes*), an abstract model introduced in [13,15] that models the control flow of programming languages with recursive coroutines. As shown in, e.g., [9,10], the reachability analysis of such models has important applications in the static analysis of programming languages with parallel constructs.

While RPPS systems can be seen as some kind of Petri nets with nested markings (the viewpoint adopted in [13,15]), we argue that it is worthwhile to see them as an infinite-state *process algebra* (or process rewrite system). This approach is very active (see [4] for a recent survey of achievements), partly because it tackles a wide range of verification problems (bisimulation checking, temporal logic model checking, etc.), and also partly because there exist several interesting process algebras (with quite different expressive power) obtained by simple syntactic restrictions on the allowed rewrite rules [20,18].

[1] Email: labroue@lsv.ens-cachan.fr
[2] Email: phs@lsv.ens-cachan.fr

Tree automata
Recently [17] showed how reachability problems for the PA process algebra [3] could be solved simply and elegantly via *tree-automata* techniques. Beyond the use of tree-automata, the approach heavily relies on an important idea: one should not consider process terms modulo any of the usual structural congruences. These congruences make process notations much lighter, and bring them closer to the intended semantics, but they hide regularity and are not really compatible with the tree-automata approach.

The tree-automata approach to PA is further developed in [16] where it is shown that the reachability relation between PA processes is an effectively recognizable relation, which gives decidability of the first-order transition logic over PA.

Our contribution
In this paper, we investigate whether the Lugiez & Schnoebelen approach to PA can be made to work for RPPS systems.

There are three main results in the paper. First we design RPA, a process rewrite system that encodes RPPS systems in a carefully chosen way. Then we prove that reachability between RPA terms is a recognizable relation: we use alternating tree-automata for a more direct proof. Finally, we show how reachability between RPPS markings can be reduced to reachability questions between RPA terms, ending with a direct automata-theoretic algorithm. As a corollary, we obtain a proof of NP-completeness for reachability between RPPS markings.

The difficulties in this work come from the fact that natural ways of encoding RPPS markings in a process-algebraic notation make it hard to define corresponding transitions via SOS (for *Structural Operational Semantics*, see [1]) rules without losing the recognizability theorem we aim at. In particular, we see no way of using the PA process algebra for this task.

Related works
Previous decidability results on RPPS [13,15] relied on more ad-hoc tableaux methods or the well-structure of RPPS [11]. These results were weaker than what we offer in section 7.

The use of recognizable sets of configurations for symbolic model checking has recently been called "*Regular model checking*" in [3]. This approach is weaker (but more practical) since it does not require that *iterated* successors or predecessors of a set of states form an effectively computable recognizable language: only *immediate* predecessors or successors are handled (sometimes, the transitive closure of loops can be handled).

There exist several other systems for which the reachability relation is

[3] A fragment allowing recursive definitions mixing sequential and parallel composition, without synchronization [2].

recognizable: it is semilinear for BPP [8], definable in the additive theory of reals for timed automata [7], a recognizable relation between words for some string rewrite systems [5] including pushdown processes (see [14] for applications to μ-calculus model checking). Our approach differs in two points: recognizability is in a tree-automata framework, and it requires that we invent a new process algebra in which to encode RPPS systems.

Plan of the paper
We first recall RPPS schemes (Section 2) before we introduce RPA (Section 3) and show how to encode RPPS schemes faithfully (Section 4). Then we recall the basic tree-automata notions (Section 5) we need to prove our main theorem (Section 6) and explain the practical implications (Section 7). A final section explains how reachability between RPPS markings can be solved in NP with tree automata.

2 Recursive-parallel program schemes

RPPS systems were introduced as an abstract model for RP programs: we refer the reader to [13,15] for motivations and examples. Here we present the formal model without justification.

2.1 The structure of RPPS systems

$A = \{a, b, \ldots\}$ is a set of *action names* that does not contain the special actions `call`, `wait`, and `end`. We write $\tilde{A}$ (ranged over by $\alpha, \beta, \ldots$) for $A \cup \{\texttt{call}, \texttt{wait}, \texttt{end}\}$.

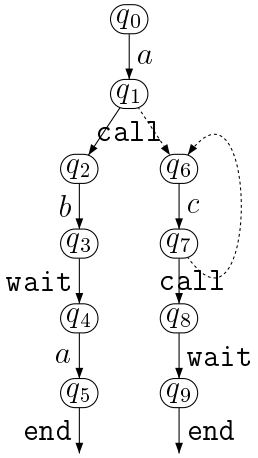


Fig. 1. A scheme

A *scheme* is a finite rooted graph $G = \langle Q, q_0, \Lambda \rangle$ where

- Q is a finite set of *nodes*,
- $q_0 \in Q$ is the *initial node*,

- Λ is the labeled flow function that maps any node q to a tuple in $(A \times Q) \cup (\{\texttt{call}\} \times Q \times Q) \cup (\{\texttt{wait}\} \times Q) \cup \{\texttt{end}\}$.

Λ has a clumsy mathematical appearance but is graphically easy to understand: every node is followed by in general one node, sometimes a pair of nodes or no node at all. For example, the system depicted in Fig. 1 has $\Lambda(q_0) = \langle a, q_1\rangle, \Lambda(q_1) = \langle \texttt{call}, q_2, q_6\rangle, \ldots, \Lambda(q_9) = \texttt{end}$.

2.2 Behavioral semantics

The behavioral semantics of G is given via an infinite labeled transition system $\mathcal{M}_G$. Informally, a state of $\mathcal{M}_G$ is a multiset of nodes (denoting the current control states of concurrent processes) organized with a father-son relationship (relating a process with the father process that spawned it via a `call` instruction). The corresponding formal definition is given below, and we refer to [13,15] for more intuitions.

Formally, the set of *hierarchical states* (also, "markings", or "states") of a system G is the least set $M(G)$ s.t. for any n nodes (not necessarily distinct) $q_1, \ldots, q_n$ of G, and hierarchical states $s_1, \ldots, s_n \in M(G)$ the multiset $s = \{(q_1, s_1), \ldots, (q_n, s_n)\}$ is in $M(G)$ [4]. In particular, $\emptyset \in M(G)$. We use the customary notations "$s + s'$", "$s \subseteq s'$", ... to denote sum, inclusion, ... of multisets and hence of hierarchical states. Below we write (q, s) for the singleton multiset $\{(q, s)\}$. The size $|s|$ of a state is given by $|\{(q_i, s_i) \mid i = 1, \ldots\}| \stackrel{\text{def}}{=} \sum_{i=1,\ldots}(1 + |s_i|)$.

We now formally define what are the transitions $\rightarrow \subseteq M(G) \times \tilde{A} \times M(G)$ between hierarchical states: $\rightarrow$ is the least set of triples (s, a, s'), written $s \xrightarrow{a} s'$, satisfying the following rules:

$$\textbf{action: if } \Lambda(q) = (a, q') \text{ then } (q, s) \xrightarrow{a} (q', s) \text{ for all } s, \tag{Ga}$$

$$\textbf{end: if } \Lambda(q) = \texttt{end} \text{ then } (q, s) \xrightarrow{\texttt{end}} s \text{ for all } s, \tag{Ge}$$

$$\textbf{call: if } \Lambda(q) = (\texttt{call}, q', q'') \text{ then } (q, s) \xrightarrow{\texttt{call}} (q', s + (q'', \emptyset)) \text{ for all } s, \tag{Gc}$$

$$\textbf{wait: if } \Lambda(q) = (\texttt{wait}, q') \text{ then } (q, \emptyset) \xrightarrow{\texttt{wait}} (q', \emptyset), \tag{Gw}$$

$$\textbf{paral1: if } s \xrightarrow{\alpha} s' \text{ then } s + s'' \xrightarrow{\alpha} s' + s'' \text{ for all } s'', \tag{Gp1}$$

$$\textbf{paral2: if } s \xrightarrow{\alpha} s' \text{ then } (q, s) \xrightarrow{\alpha} (q, s') \text{ for all } q \in Q. \tag{Gp2}$$

Rules **paral1** and **paral2** for parallelism express that any activity $s \xrightarrow{\alpha} s'$ can still take place when brothers are present (i.e. in some $s + s''$) or when a parent is present (i.e. in some (q, s)). The **wait** rule states how we can

[4] A hierarchical state of the form $s = \{(q_1, s_1), \ldots, (q_n, s_n)\}$ has n completely independent concurrent activities. One such activity, say (q_i, s_i), is the invocation of a coroutine (currently in state/node q_i) together with its family of children invocations (the s_i part).

only perform a `wait` statement in state q if the invoked children are all terminated (and then not present anymore). The other rules state how children invocations are created and kept around.

Finally, $\mathcal{M}_G$ is $\langle M(G), \tilde{A}, \to, s_0 \rangle$ where the initial state is $s_0 \stackrel{\text{def}}{=} (q_0, \emptyset)$.

Example 2.1 $(q_0,\emptyset) \xrightarrow{a} (q_1,\emptyset) \xrightarrow{\texttt{call}} (q_2,(q_6,\emptyset)) \xrightarrow{c} (q_2,(q_7,\emptyset)) \xrightarrow{\texttt{call}} (q_2,(q_8,(q_6,\emptyset))) \xrightarrow{b} (q_3,(q_8,(q_6,\emptyset)))\cdots$ is an execution sequence of the system $\mathcal{M}_G$ associated with the scheme of Fig. 1.

As the **wait** rule shows, nodes that can only be exited via a `wait` step behave conditionally: we denote by $Q^?$ the set of the states q of Q such that $\Lambda(q) = (\texttt{wait}, q')$ for some q', while $Q^!$ denotes $Q \setminus Q^?$.

3 The process algebra RPA

We now define RPA, a process algebra designed to encode RPPS schemes.

3.1 RPA terms

We assume a scheme $G = \langle Q, q_0, \Lambda \rangle$ is fixed and consider the set $\mathit{Const} \stackrel{\text{def}}{=} Q \cup \{0\}$ ranged over by $c, \ldots$ T_G, the set of *RPA terms*, or just "terms", ranged over by $t, u, v, \ldots$ is given by the following syntax:

$$t, u ::= c \mid t \blacktriangleright u.$$

For t a term, we write $\mathit{State}(t)$ the set of all nodes from Q that occur in t. The *size* of t, denoted $|t|$, is the number of symbols in t, given by $|c| \stackrel{\text{def}}{=} 1$ and $|t \blacktriangleright u| \stackrel{\text{def}}{=} 1 + |t| + |u|$.

RPA terms are binary trees but the left- and right-hand sides do not play the same rôle, so that it is more natural to see them as combs with some c from Const at the deep left end, and a list of subterms on the right of the spine (see example on Fig. 2). This motivates introducing the convenient abbreviation "$c \blacktriangleright^n (u_1, \ldots, u_n)$", defined inductively by $c \blacktriangleright^0 () = 0$ and $c \blacktriangleright^n (u_1, \ldots, u_n) = \big(c \blacktriangleright^{n-1} (u_1, \ldots, u_{n-1})\big) \blacktriangleright u_n$. We only use the "$\blacktriangleright^n$" abbreviation with a $c \in \mathit{Const}$ in the left-hand side.

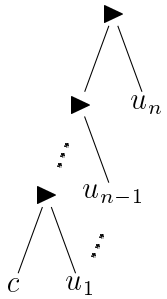


Fig. 2. $c \blacktriangleright^n (u_1, \ldots, u_n)$

A (guarded) RPA *declaration* is a finite set $\Delta \subseteq Q \times \tilde{A} \times Const \times T_G$ of rules, written $\{q_i \xrightarrow{\alpha}_\Delta c_i, t_i \mid i = 1, \ldots, n\}$. The q_i's need not be distinct. For technical convenience, we require that all $q \in Q$ appear in the left-hand side of at least one rule.

3.2 Semantics

Let $Act \stackrel{\text{def}}{=} \tilde{A} \times \{!, ?\}$. For convenience, we write $\alpha^!$ and $\alpha^?$ rather than $(\alpha, !)$ and $(\alpha, ?)$. A declaration Δ defines a labeled transition $\rightarrow \subseteq T_G \times Act \times T_G$, given by the following SOS rules:

$$\text{R1}\ \frac{}{q \xrightarrow{\alpha^!} q' \blacktriangleright t}\ \text{if } (q \xrightarrow{\alpha}_\Delta q', t) \in \Delta \text{ and } q \in Q^! \qquad \text{R3}\ \frac{t \xrightarrow{\alpha^!} t'}{u \blacktriangleright t \xrightarrow{\alpha^!} u \blacktriangleright t'}$$

$$\text{R2}\ \frac{}{q \xrightarrow{\alpha^?} q' \blacktriangleright t}\ \text{if } (q \xrightarrow{\alpha}_\Delta q', t) \in \Delta \text{ and } q \in Q^? \qquad \text{R4}\ \frac{t \xrightarrow{\alpha^?} t'}{u \blacktriangleright t \xrightarrow{\alpha^!} u \blacktriangleright t'}$$

$$\text{R5}\ \frac{t \xrightarrow{\alpha^!} t'}{t \blacktriangleright u \xrightarrow{\alpha^!} t' \blacktriangleright u} \qquad \text{R6}\ \frac{t \xrightarrow{\alpha^?} t'}{t \blacktriangleright u \xrightarrow{\alpha^?} t' \blacktriangleright u}\ \text{if } State(u) = \emptyset$$

The intuition is that a step $t \xrightarrow{\alpha^x} u$ in T_G encodes a step $s_t \xrightarrow{\alpha} s_u$ in $\mathcal{M}_G$ (where s_t is the hierarchical state denoted by t). The extra label $x =!$ (resp. $x =?$) means that this step can (resp. cannot) occur on top of active children processes. The label is chosen by rules R1, R2, tested by rules R5, R6, and propagated according to the semantics.

We write $u \xrightarrow{!} v$ (resp. $u \xrightarrow{?} v$) when $u \xrightarrow{\alpha^!} v$ (resp. $u \xrightarrow{\alpha^?} v$) for some α, and $u \rightarrow v$ when $u \xrightarrow{!} v$ or $u \xrightarrow{?} v$. For $n \in \mathbb{N}$, we let "$\xrightarrow{n}$" and "$\xrightarrow{n,!}$" denote respectively the iterated relations $(\rightarrow)^n$ and $\left(\xrightarrow{!}\right)^n$. Also $\rightarrow^*$ denotes the closure $\bigcup_{n\in\mathbb{N}} \xrightarrow{n}$. As usual, "$u \rightarrow$" and "$u \not\rightarrow$" mean respectively that $u \rightarrow v$ for some v (resp. for no v).

3.3 Basic properties of RPA steps

We now list some key lemmas about the transitions between terms. These results aim at explaining how one can decompose a compound step into smaller steps and will be the basis of the construction in section 6.

Lemma 3.1 *If $u \blacktriangleright v \rightarrow w$ then w has the form $u' \blacktriangleright v'$ and either ($u \rightarrow u'$ and $v = v'$) or ($v \rightarrow v'$ and $u = u'$).*

Proof. By case analysis of rules R3–R6. □

Lemma 3.2 *If $u \rightarrow v$ then $|v| > |u|$.*

Proof. By induction on the derivation $u \to v$. The base cases are transitions $q \to q' \blacktriangleright t$. □

Lemma 3.3 *$q \to^* q'$ iff $q = q'$.*

Proof. $q \xrightarrow{n} q'$ entails $n = 0$ (Lemma 3.2). □

The next six lemmas are proved in the Appendix. Lemma 3.5 gives a characterization of $\xrightarrow{!}{}^*$.

Lemma 3.4 *$u \to$ iff $State(u) \neq \emptyset$.*

Lemma 3.5 *$u \xrightarrow{!}{}^* v$ iff for all $t \in T_G$, $u \blacktriangleright t \to^* v \blacktriangleright t$.*

Lemma 3.6 *$v \blacktriangleright t \xrightarrow{!}{}^* v' \blacktriangleright t'$ iff $v \xrightarrow{!}{}^* v'$ and $t \to^* t'$.*

Lemma 3.7 *$v \blacktriangleright t \to^* v' \blacktriangleright t'$ iff $t \to^* t'$ and $\begin{cases} t' \not\to \text{ and } v \to^* v', \\ \text{or } v \xrightarrow{!}{}^* v'. \end{cases}$*

Lemma 3.8 *$q \to^* v \blacktriangleright t$ iff there exist c and u s.t. $(q \to_\Delta c, u)$ is a rule in Δ, $u \to^* t$, and $\begin{cases} t \not\to \text{ and } c \to^* v, \\ \text{or } c \xrightarrow{!}{}^* v. \end{cases}$*

Lemma 3.9 *$q \xrightarrow{!}{}^* v \blacktriangleright t$ iff $q \in Q^!$ and there exist c and u s.t. $q \to_\Delta c, u$ is a rule in Δ, $u \to^* t$ and $c \xrightarrow{!}{}^* v$.*

4 Embedding RPPS schemes into RPA

The behavior of an RPPS scheme G can be faithfully encoded in RPA. We consider a set of rules Δ_G obtained from Λ. For any $q \in Q$,

$$\textbf{action: if } \Lambda(q) = (a, q') \text{ then } \Delta_G \text{ contains } q \xrightarrow{a} q', 0, \tag{Da}$$

$$\textbf{end: if } \Lambda(q) = \texttt{end} \text{ then } \Delta_G \text{ contains } q \xrightarrow{\texttt{end}} 0, 0, \tag{De}$$

$$\textbf{call: if } \Lambda(q) = (\texttt{call}, q', q'') \text{ then } \Delta_G \text{ contains } q \xrightarrow{\texttt{call}} q', q'', \tag{Dc}$$

$$\textbf{wait: if } \Lambda(q) = (\texttt{wait}, q') \text{ then } \Delta_G \text{ contains } q \xrightarrow{\texttt{wait}} q', 0. \tag{Dw}$$

Thus Δ_G can be seen as an application from Q to $\tilde{A} \times Const \times T_G$.

We now associate a hierarchical state $\mathcal{S}(t)$ with any term $t \in T_G$ and, reciprocally, a term $\mathcal{T}(s)$ with any $s \in M(G)$. The aim is to define what hierarchical state is encoded by term t, and what term can be used to encode hierarchical state s.

7

The mappings $\mathcal{S}$ and $\mathcal{T}$ are defined inductively by

$$\mathcal{T}(\{(q_1, s_1), \ldots, (q_n, s_n)\}) \stackrel{\text{def}}{=} 0 \blacktriangleright^n (q_1 \blacktriangleright \mathcal{T}(s_1), \ldots, q_n \blacktriangleright \mathcal{T}(s_n)) \quad \text{(T)}$$

$$\mathcal{S}(0 \blacktriangleright^n (u_1, \ldots, u_n)) \stackrel{\text{def}}{=} \mathcal{S}(u_1) + \cdots + \mathcal{S}(u_n) \quad \text{(S1)}$$

$$\mathcal{S}(q \blacktriangleright^n (u_1, \ldots, u_n)) \stackrel{\text{def}}{=} (q, \mathcal{S}(u_1) + \cdots + \mathcal{S}(u_n)) \quad \text{(S2)}$$

where equation (T) for $\mathcal{T}(s)$ requires that one picks some ordering of the elements of the multiset s.

$\mathcal{S}$ and $\mathcal{T}$ behave like an abstraction-concretization pair:

Lemma 4.1 *For all $s \in M(G)$, $\mathcal{S}(\mathcal{T}(s)) = s$.*

Proof. By structural induction on s, using equations (T,S1,S2). □

$\mathcal{S}$ gives rise to an equivalence between RPA terms: $t \equiv_{\mathcal{S}} u \stackrel{\text{def}}{\Leftrightarrow} \mathcal{S}(t) = \mathcal{S}(u)$. We write $[u]$ for the equivalence class of u w.r.t. $\equiv_{\mathcal{S}}$, and $T_{\equiv_{\mathcal{S}}}$ for the set of the equivalence classes of T_G.

Observe that $\equiv_{\mathcal{S}}$ is not a congruence: $(0 \blacktriangleright u) \equiv_{\mathcal{S}} u$ whereas $(0 \blacktriangleright u) \blacktriangleright v \not\equiv_{\mathcal{S}} u \blacktriangleright v$

It is now possible to state how steps between RPA terms are related to steps between RPPS hierarchical states. This is done by abstracting over the ! or ? extra label that RPA steps carry, and that is only used for a compositional definition of steps. Write $u \xrightarrow{\alpha} t$ when $u \xrightarrow{\alpha^\varepsilon} t$ for some $\varepsilon \in \{!, ?\}$.

Proposition 4.2 *1. For all u, v in T_G and α in $\tilde{A}$, if $u \xrightarrow{\alpha} t$ then $\mathcal{S}(u) \xrightarrow{\alpha} \mathcal{S}(t)$.*
2. For all s, s' in $M(G)$ and α in $\tilde{A}$, if $s \xrightarrow{\alpha} s'$, then $\mathcal{T}(s) \xrightarrow{\alpha} u$ for some $u \in T_G$ such that $\mathcal{S}(u) = s'$.

Proof (Idea). 1. (resp. 2.) is proved by induction on u (resp. s) and a tedious case analysis. □

The meaning of Proposition 4.2 is that, modulo the abstraction mapping from Act to $\tilde{A}$ that sends α^ε to α, $\mathcal{S}$ is a bisimulation between the RPA transition system generated by Δ_G and the transition system $\mathcal{M}_G$ we want to analyze.

5 Tree languages and tree automata

Here we recall the classical tree-automata notions we need. We refer to [6] and [22] for more details.

5.1 Tree languages

Given a finite ranked alphabet $\mathcal{F} = \mathcal{F}_0 \cup \mathcal{F}_1 \cup \ldots \cup \mathcal{F}_m$, $T_{\mathcal{F}}$ denotes the set of finite trees (or terms) built from $\mathcal{F}$: for example, with $\mathcal{F}_0 = \{a, b\}$, $\mathcal{F}_1 = \{g, h\}$

and $\mathcal{F}_2 = \{f\}$, $T_\mathcal{F}$ contains trees like a, $f(a,b)$ and $f(g(f(h(b),a)),b)$. A *tree language* is any subset L of $T_\mathcal{F}$.

5.2 Tree automata

A *tree automaton* is a tuple $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, F, \delta \rangle$ where $\mathcal{F}$ is a finite ranked alphabet, $\mathcal{Q} = \{p, p' \ldots\}$ is a finite set of control states, $F \subset \mathcal{Q}$ is a set of accepting states and $\delta \subseteq \cup_{n\in\mathbb{N}}(\mathcal{Q} \times \mathcal{F}_n \times \mathcal{Q}^n)$ is a finite set of *transition rules*.

We refer to [6] (or [17]) for the classical definition of when a tree t is recognized by state p of $\mathcal{A}$, written $p \overset{*}{\mapsto} t$. For $p \in \mathcal{Q}$, $L(p)$ denotes $\{t \mid p \overset{*}{\mapsto} t\}$. $L(\mathcal{A}) \overset{\text{def}}{=} \bigcup_{p\in F} L(p)$ is the tree language recognized by $\mathcal{A}$.

Example 5.1 Continuing with our previous example, and setting $\mathcal{Q} = \{p_0, p_1\}$, the set of rules describes a top-down tree automaton

$$\begin{array}{lll} p_0 \mapsto a & p_0 \mapsto b & p_1 \mapsto g(p_0) \\ p_0 \mapsto g(p_1) & p_0 \mapsto h(p_1) & p_1 \mapsto h(p_0) \\ p_0 \mapsto f(p_1,p_1) & p_0 \mapsto f(p_0,p_0) & p_1 \mapsto f(p_0,p_1) \\ & & p_1 \mapsto f(p_1,p_0) \end{array}$$

A possible derivation of $f(h(b),a)$ by $\mathcal{A}$ is $p_1 \mapsto f(p_1,p_0) \mapsto f(h(p_0),p_0) \mapsto f(h(p_0),a) \mapsto f(h(b),a)$. So $p_1 \overset{*}{\mapsto} f(h(b),a)$.

5.3 Alternating tree automata

An alternating tree automaton is a tuple $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, F, \delta \rangle$ where now δ is a n-indexed family of maps from $\mathcal{Q} \times \mathcal{F}_n$ to $\mathcal{B}^+(\{1,\ldots,n\} \times \mathcal{Q})$. Here, for a given set X, $\mathcal{B}^+(X)$ is the set of positive Boolean formulas over X (i.e., Boolean formulas built from elements in X using $\wedge$ and $\vee$), where we also allow the formulas *true* and *false*. For example we could have $\delta(p,f) = (1,p_1) \vee ((1,p_2) \wedge (2,p_3) \wedge (2,p_4))$.

We refer to [22] for the classical definition of when a tree t is recognized by state p of some alternating $\mathcal{A}$. It is well-known that standard tree automata can be seen as alternating automata where only disjunctions are used, and that the class of trees languages recognized by alternating tree automata is exactly the class of tree languages recognized by non-alternating tree automata.

5.4 Recognizable relations on trees

We follow [6, Chapter 3] and [16]. A tuple $\langle t_1, \ldots, t_n \rangle$ of n trees from $T_\mathcal{F}$ can be seen as a single tree, denoted $t_1 \times \cdots \times t_n$, on a product alphabet $\mathcal{F}^{\times n} \overset{\text{def}}{=} (\mathcal{F} \cup \{\bot\})^n$ where the arity of $f_1 \ldots f_n$ is the maximum of the arities of the f_i, assuming $\bot$ has arity 0.

For instance the pair $\langle f(a,g(b)), f(f(a,a),b)\rangle$ can also be seen as $ff(af(\bot a, \bot a), gb(b\bot))$.

We say a n-ary relation $R \subseteq T_{\mathcal{F}}^n$ is recognizable iff the set of all $t_1 \times \cdots \times t_n$ for $(t_1, \ldots, t_n) \in R$ is a regular tree language over $\mathcal{F}^{\times n}$.

6 Recognizability of the reachability relation for RPA

The reachability relations $\to^*$ and $\xrightarrow{!}{}^*$ between RPA terms are recognizable:

Lemma 6.1 *The set* $L_{\text{term}} \stackrel{def}{=} \{u \in T_G \mid u \not\to\}$ *of* terminated terms *is recognizable.*

Proof. $u \not\to$ iff $State(u) = \emptyset$ (Lemma 3.4). Thus the automaton with an unique accepting state $p_\downarrow$ and the transition rules

$$\delta(p_\downarrow, 0) = true, \qquad \delta(p_\downarrow, q) = false, \qquad \delta(p_\downarrow, \blacktriangleright) = (1, p_\downarrow) \wedge (2, p_\downarrow) \tag{1}$$

recognizes L_{term}. □

We now consider the alternating automaton $\mathcal{A}_{\xrightarrow{*}}$ whose states are p, $\bar{p}$, $p_\downarrow$ and all p_t and $\bar{p}_t$ for t a subterm of some term appearing in Δ (thus $|\mathcal{Q}|$ is in $O(|\Delta|)$).

$\mathcal{A}_{\xrightarrow{*}}$ recognizes pairs of terms. Here we define the alternating transition function δ with the following assumptions: (1) we omit the rules for $\delta(p_\downarrow, \ldots)$, (2) when $\delta(p', fg)$ is not explicitly defined (for some $p' \in \mathcal{Q}$ and some $f, g \in (\mathcal{F} \cup \{\bot\})$) this means $\delta(p', fg)$ is false, and (3) we quantify over all $q \in Q$, all $c \in Const$, and all $f \in (\mathcal{F} \cup \{\bot\})$.

$$\delta(p, 00) = \delta(\bar{p}, 00) = true \tag{2}$$

$$\delta(p, qq') = \delta(\bar{p}, qq') = \begin{cases} true \text{ if } q = q', \\ false \text{ otherwise} \end{cases} \tag{3}$$

$$\delta(p, \blacktriangleright\blacktriangleright) = (2, p) \wedge \big[(1, \bar{p}) \vee ((2, p_\downarrow) \wedge (1, p))\big] \tag{4}$$

$$\delta(\bar{p}, \blacktriangleright\blacktriangleright) = (1, \bar{p}) \wedge (2, p) \tag{5}$$

$$\delta(p, q\,\blacktriangleright) = \bigvee_{q \to_\Delta c, u} (2, p_u) \wedge \big[(1, \bar{p}_c) \vee ((2, p_\downarrow) \wedge (1, p_c))\big] \tag{6}$$

$$\delta(\bar{p}, q\,\blacktriangleright) = \begin{cases} \displaystyle\bigvee_{q \to_\Delta c, u} (2, p_u) \wedge (1, \bar{p}_c) \text{ if } q \in Q^!, \\ false \text{ otherwise} \end{cases} \tag{7}$$

$$\delta(p_t, f0) = \delta(\bar{p}_t, f0) = \begin{cases} \mathit{true} \text{ if } t = 0, \\ \mathit{false} \text{ otherwise} \end{cases} \tag{8}$$

$$\delta(p_t, fq) = \delta(\bar{p}_t, fq) = \begin{cases} \mathit{true} \text{ if } t = q, \\ \mathit{false} \text{ otherwise} \end{cases} \tag{9}$$

$$\delta(p_{t_1 \blacktriangleright t_2}, f\ \blacktriangleright) = (2, p_{t_2}) \wedge \big[(1, \bar{p}_{t_1}) \vee ((2, p_\downarrow) \wedge (1, p_{t_1}))\big] \tag{10}$$

$$\delta(\bar{p}_{t_1 \blacktriangleright t_2}, f\ \blacktriangleright) = (1, \bar{p}_{t_1}) \wedge (2, p_{t_2}) \tag{11}$$

$$\delta(p_q, f\ \blacktriangleright) = \delta(p, q\ \blacktriangleright) \tag{12}$$

$$\delta(\bar{p}_q, f\ \blacktriangleright) = \delta(\bar{p}, q\ \blacktriangleright) \tag{13}$$

This automaton satisfies the following correctness property:

Lemma 6.2

$$L(p) = \{u \times v \mid u \to^* v\}, \qquad L(\bar{p}) = \{u \times v \mid u \xrightarrow{!}{}^* v\}, \tag{14}$$

$$L(p_t) = \{u \times v \mid t \to^* v\}, \qquad L(\bar{p}_t) = \{u \times v \mid t \xrightarrow{!}{}^* v\}, \tag{15}$$

$$L(p_\downarrow) = \{u \times v \mid v \not\to\}, \tag{16}$$

where u, v are any terms of $T_G \cup \{\bot\}$.

Proof (Sketch). The rules for $\delta(p_\downarrow, \ldots)$ are the obvious modifications of (1) so that they apply to the second element of a pair $u \times v$ while we do not take care of the first element.

The proof is by induction over the derivations $u \to^* v$, ..., for the ($\supseteq$) directions, and by induction over the product term for the ($\subseteq$) directions.

It turns out every transition rule between (2) and (13) is justified by a behavioral property we already proved. For example, Lemma 3.3 accounts for (3) while Lemma 3.4 accounts for all rules $\delta(p_\downarrow, fg)$. Similarly, (5) is a direct transposition of Lemma 3.6. □

We obtain the important corollary:

Theorem 6.3 *The relations $\to^*$ and $\xrightarrow{!}{}^*$ are recognizable. Furthermore, a tree automaton recognizing them only needs $O(|\Delta|)$ states.*

Proof. Our construction used an alternating automaton for clarity (the clauses defining δ mimic lemmas from section 3.3) but it is easy to adapt the construction and get a (non-deterministic bottom up) tree automaton with $O(|\Delta|)$ states. □

7 Applications

Theorem 6.3 immediately leads to decidability results for RPA terms (and RPPS schemes). The nice thing with these results is that they all involve the

11

same smooth and general automata-theoretic reasoning.

Reachability sets. For any recognizable language L, the sets $Pre^*(L) \stackrel{\text{def}}{=} \{u \mid u \stackrel{*}{\to} v \text{ for some } v \in L\}$ and $Post^*(L) \stackrel{\text{def}}{=} \{u \mid v \stackrel{*}{\to} u \text{ for some } v \in L\}$ are recognizable, and the corresponding automata can be obtained in polynomial-time by standard intersection and projection constructs on automata (assuming an automaton for L is known).

Reachability under constraints. These result extend to reachability under constraints, i.e. to the sets $Pre^*{}_C(L) \stackrel{\text{def}}{=} \{u \mid u \stackrel{\sigma}{\to} v \text{ for some } v \in L \text{ and } \sigma \in C\}$ and $Post^*{}_C(L) \stackrel{\text{def}}{=} \{u \mid v \stackrel{\sigma}{\to} u \text{ for some } v \in L \text{ and } \sigma \in C\}$ where $C \subseteq Act^*$ is a constraint on acceptable labels for reachability. Not all regular $C \subseteq Act^*$ can be dealt with in this approach (see [17,16]) but interesting regular constraints, called decomposable constraints, are allowed [21].

Model checking the logic EF. Using Pre^* and standard constructs for intersection and complementation, one can compute for any formula φ of the modal logic EF, the set $Mod(\varphi)$ of all terms that satisfy φ (see [17,19]). Here, EF can even be enriched with decomposable constraints.

Note that since bisimilar processes satisfy the same EF formulas, we have $s \models \varphi$ iff $\mathcal{T}(s) \models \varphi$, so that this approach allows model checking RPPS schemes.

Model checking the transition logic. EF only needs effective recognizability of $Pre^*(L)$ for recognizable L. But with recognizability of $\stackrel{*}{\to}$, we get a simple model checking algorithm for the full transition logic [5], i.e. the first-order logic $FO(\to, \stackrel{*}{\to})$. See [16] for details and applications.

8 Reachability between RPPS markings

Here we reduce the problem of reachability between RPPS markings to reachability questions between RPA terms. As a result, we get a simple automata-theoretic algorithm for RPPS reachability, from which NP-completeness of reachability is easily derived.

Write $u \stackrel{\alpha}{\Rightarrow} v$ when $u \equiv_{\mathcal{S}} u' \stackrel{\alpha}{\to} v' \equiv_{\mathcal{S}} v$ for some u', v'. We adopt the usual extensions $u \stackrel{\sigma}{\Rightarrow} v$ (for $\sigma \in Act^*$) and $u \stackrel{*}{\Rightarrow} v$. Reachability between RPPS markings reduces to $\stackrel{*}{\Rightarrow}$-reachability between RPA terms, in the following formal sense:

Proposition 8.1 *Given two RPPS markings s and s', $s \stackrel{*}{\to} s'$ in $\mathcal{M}_G$ iff $\mathcal{T}(s) \stackrel{*}{\Rightarrow} \mathcal{T}(t)$ in T_G.*

[5] It is difficult to extend this decidability result: by encoding a grid structure into RPA, one can easily show that model checking $MSO(\to)$, the monadic second-order logic with $\to$ as the only predicate, is undecidable over RPA terms.

Proof. Combine Prop. 4.2 and the definition of $\Rightarrow$. □

8.1 Another characterization of $\equiv_{\mathcal{S}}$

Our next task is to obtain a characterization of $\equiv_{\mathcal{S}}$ that is more manageable from a regular tree languages viewpoint. We do this with in several small steps, with the help of some simplification or permutation relations between RPA terms. The basic concepts (confluence, commutations, ...) used in this subsection are standard in the study of reduction systems (see e.g. [12]).

8.1.1 Simplification

The relations $\curvearrowright$ and $\searrow$ are defined inductively by the following erasing rules:

$$0 \blacktriangleright u \curvearrowright u \tag{E1}$$

$$c \blacktriangleright^n (t_1, \ldots, t_{i-1}, 0 \blacktriangleright^m (u_1, \ldots, u_m), t_{i+1}, \ldots, t_n) \searrow \\ c \blacktriangleright^{n+m-1} (t_1, \ldots, t_{i-1}, u_1, \ldots, u_m, t_{i+1}, \ldots, t_n) \tag{E2}$$

$$\text{if } t_i \searrow u, \text{ then } c \blacktriangleright^n (t_1, \ldots, t_n) \searrow c \blacktriangleright^n (t_1, \ldots, t_{i-1}, u, t_{i+1}, \ldots, t_n) \tag{E3}$$

We let $\twoheadrightarrow$ denote $\curvearrowright \cup \searrow$ and will use juxtaposition to denote the composition of relations. Observe that $t \curvearrowright \searrow u$ implies $t \searrow \curvearrowright u$, and that $t \curvearrowright \curvearrowright u$ implies $t \searrow \curvearrowright u$. Thus, writing $\twoheadrightarrow^*$ for the reflexive-transitive closure of $\twoheadrightarrow$, we deduce that $\twoheadrightarrow^*$ coincide with $\searrow^* \curvearrowright^*$ and then with $\searrow \curvearrowright^=$, where $\curvearrowright^=$ denotes $\curvearrowright \cup Id$.

When $t \twoheadrightarrow^* u$, we say that u is a *simplification* of t. We write $\twoheadleftarrow$ and ${}^*\!\twoheadleftarrow$ to denote the reverse relations $(\twoheadrightarrow)^{-1}$ and $(\twoheadrightarrow^*)^{-1}$. Since $u \twoheadrightarrow t$ implies $|u| > |t|$, $\twoheadrightarrow$ is noetherian and $\twoheadrightarrow^*$ is a well-founded partial ordering.

Lemma 8.2 (Confluence) *If $u \twoheadleftarrow v \twoheadrightarrow w$, then $u = w$ or $u \twoheadrightarrow v' \twoheadleftarrow w$ for some v'.*

Proof. By induction on v and case analysis. See Appendix A.7. □

Hence, by Newman's Lemma, $\twoheadrightarrow$ is convergent: we let $t{\downarrow}$ denote the *simplification normal form* of t, i.e. the unique u one obtains by simplifying t as much as possible.

8.1.2 Permutation

The relation $\rightleftarrows$ is defined inductively by the following rules:

$$c \blacktriangleright^n (t_1, \ldots, t_n) \rightleftarrows c \blacktriangleright^n (t_1, \ldots, t_{i-1}, t_{i+1}, t_i, t_{i+2}, \ldots, t_n) \tag{P1}$$

$$\text{if } t_i \rightleftarrows u, \text{ then } c \blacktriangleright^n (t_1, \ldots, t_n) \rightleftarrows c \blacktriangleright^n (t_1, \ldots, t_{i-1}, u, t_{i+1}, \ldots, t_n) \tag{P2}$$

$\rightleftarrows$ is symmetric. We write $\overset{*}{\rightleftarrows}$ to denote the reflexive-transitive closure of $\rightleftarrows$. When $t \overset{*}{\rightleftarrows} u$, we say t and u are *permutationally equivalent*.

The next lemma allows commuting simplification and permutation:

Lemma 8.3 (Commutation) *If $u \rightleftarrows v \twoheadrightarrow w$, then $u \twoheadleftarrow v' \overset{*}{\rightleftarrows} w$ for some v'.*

13

Proof. By induction on u and case analysis. See Appendix A.8. □

By symmetry, $u \swarrow \rightleftarrows w$ entails $u \overset{*}{\rightleftarrows} \swarrow w$.

8.1.3 Convertibility

Finally, we combine simplifications and permutations in $\leftrightsquigarrow$, a relation defined as $\left(\searrow \cup \rightleftarrows \cup \swarrow\right)^*$. When $u \leftrightsquigarrow v$, we say that u *is can be converted in* v.

Lemma 8.4 *The following are equivalent:*

(a) $u \leftrightsquigarrow v$,

(b) *there exist two terms* u' *and* v' *s.t.* $u \searrow^* u' \overset{*}{\rightleftarrows} v' \ {}^*\!\swarrow v$,

(c) $u{\downarrow} \overset{*}{\rightleftarrows} v{\downarrow}$.

Proof. Obviously (c) ⇒ (b) ⇒ (a). One proves (a) ⇒ (b) by a standard "peaks into valleys" normalization: Lemmas 8.2 and 8.3 allow erasing local peaks. Termination is guaranteed because $\overset{*}{\rightleftarrows} \searrow \overset{*}{\rightleftarrows}$ is noetherian, so that the multiset of peaks strictly decreases (in the well-founded multiset ordering obtained from $\overset{*}{\rightleftarrows} \searrow \overset{*}{\rightleftarrows}$) after every local transformation.

Then (b) ⇒ (c) is easy: $u \leftrightsquigarrow v$ entails $u{\downarrow} \ {}^*\!\swarrow u \leftrightsquigarrow v \searrow^* v{\downarrow}$ or shortly $u{\downarrow} \leftrightsquigarrow v{\downarrow}$. Thus $u{\downarrow} \searrow^* \overset{*}{\rightleftarrows} {}^*\!\swarrow v{\downarrow}$ by (a) ⇒ (b). But since $u{\downarrow}$ and $v{\downarrow}$ cannot be simplified further, we get $u{\downarrow} \overset{*}{\rightleftarrows} v{\downarrow}$. □

Proposition 8.5 $u \equiv_{\mathcal{S}} v$ *if and only if* $u \leftrightsquigarrow v$.

Proof. The (⇐) direction is obvious: a simple inspection of the rules show that $u \searrow v$ or $u \curvearrowright v$ or $u \rightleftarrows v$ implies $\mathcal{S}(u) = \mathcal{S}(v)$. The (⇒) direction is proved in Appendix A.9. □

Having decomposed $\equiv_{\mathcal{S}}$ into "permutation" and "simplification" allows a partial answer to the question of "what is the set of terms that belong to some regular set L modulo $\mathcal{S}$-equivalence?".

For a tree language L define

$$[L]_{\rightleftarrows} \overset{\text{def}}{=} \{u \mid \exists t \in L, u \overset{*}{\rightleftarrows} t\}, \qquad [L]_{\swarrow} \overset{\text{def}}{=} \{u \mid \exists t \in L, u \searrow^* t\},$$
$$[L]_{\leftrightsquigarrow} \overset{\text{def}}{=} \{u \mid \exists t \in L, u \leftrightsquigarrow t\}. \qquad [L]_{\searrow} \overset{\text{def}}{=} \{u \mid \exists t \in L, t \searrow^* u\},$$

If L is regular, then $[L]_{\rightleftarrows}$ and $[L]_{\leftrightsquigarrow}$ are not regular in general, while $[L]_{\swarrow}$ and $[L]_{\searrow}$ are. For our purposes, we shall need the following:

Lemma 8.6 *If* L *is regular then* $[L]_{\swarrow}$ *is regular. Furthermore, from a tree automaton* $\mathcal{A}$ *recognizing* L*, one can build in polynomial-time a tree automaton* $\mathcal{A}'$ *for* $[L]_{\swarrow}$ *with* $|\mathcal{A}'| = O(|\mathcal{A}|^2)$.

Proof (Idea). First, for any pair p, q of states of $\mathcal{A}$, we add a state r_p^q and rules such that $t \mapsto^* r_p^q$ iff t is some $0 \blacktriangleright^n (t_1, \ldots, t_n)$ and $p \blacktriangleright^n (t_1, \ldots, t_n) \mapsto^* q$

14

in $\mathcal{A}$. Then, whenever $q \blacktriangleright q' \overset{*}{\mapsto} q''$, we add all rules of the form $r_p^q \blacktriangleright q' \mapsto r_p^{q''}$. With further rules $p \blacktriangleright r_p^q \mapsto q$ and $r_p^q \blacktriangleright r_q^r \mapsto r_p^r$, the resulting automaton has $t \overset{*}{\mapsto} p$ iff $t \searrow^* u$ for some u with $u \overset{*}{\mapsto} p$ in $\mathcal{A}$.

Then the construction is easily completed in view of $\searrow\!\!\!\!\searrow^* = \searrow^* \curvearrowright^=$. □

8.2 Transitions modulo $\equiv_{\mathcal{S}}$

We can now prove that $\equiv_{\mathcal{S}}$ (or equivalently $\leftrightsquigarrow$) respects behaviours in a sense stronger than just being included in the largest bisimulation:

Proposition 8.7 *$\equiv_{\mathcal{S}}$ is a bisimulation relation modulo the abstraction of $\{!, ?\}$ labels, i.e. $u \equiv_{\mathcal{S}} v$ and $u \overset{\alpha}{\to} u'$ implies that $v \overset{\alpha}{\to} v'$ for some v' with $v \equiv_{\mathcal{S}} v'$.*

Proof (Idea). Standard but tedious. One proves that $\rightleftarrows$, $\searrow$ and $\curvearrowright$ are bisimulations up-to $\leftrightsquigarrow$. Prop. 8.5 concludes. □

Proposition 8.8 *For any $\sigma \in Act^*$, $t \overset{\sigma}{\Rightarrow} u$ iff $t \overset{\sigma}{\to} u'$ for some $u' \equiv_{\mathcal{S}} u$.*

Proof. By induction on the length of σ and using Prop. 8.7. □

With Prop. 8.5 and Lemma 8.4, we get

Lemma 8.9 *$u \overset{*}{\Rightarrow} v$ iff $u \overset{*}{\to} w$ for some w s.t. $v{\downarrow} \overset{*}{\rightleftarrows} w{\downarrow}$.*

8.3 A NP-algorithm for $\overset{*}{\Rightarrow}$-reachability

We can now prove the following

Theorem 8.10 *$\overset{*}{\Rightarrow}$-reachability between RPA terms is NP-complete.*

Proof. NP-hardness is well-known already for simpler process algebra like BPP [8].

We now show membership in NP. Given u and v, we compute $v{\downarrow}$ in polynomial-time, guess a w s.t. $v{\downarrow} \overset{*}{\rightleftarrows} w$ (note that $|w| \leq |v|$), build a tree automaton for $L = [w]_{\swarrow\!\!\!\swarrow}$ using Lemma 8.6, and then an automaton for $L' = Pre^*(L) = \{t \mid t \overset{*}{\to} t' \in L\}$ using Theo. 6.3 (these automata can be built in polynomial-time). We answer yes if $u \in L'$. Lemma 8.9 states that this algorithm is correct. □

9 Conclusion

We encoded RPPS systems into RPA, a process rewrite system that combines several features:

- it has an effectively recognizable reachability relation,
- hence an uniform tree automata method can compute the models of any formula written in the transition logic TL,

• which can be used for the reachability analysis of RPPS systems.

The difficulty in that work was to discover a process-algebraic presentation of hierarchical states where transitions are local enough so that the reachability relation is recognizable, which is the sensitive problem. The consequence is that the link between hierarchical states and RPA terms is not direct: $\equiv_{\mathcal{S}}$ is not a congruence, we need to use two notions "$u \xrightarrow{\alpha^!} v$" and "$u \xrightarrow{\alpha^?} v$", etc.

We see this work as more proof of the power of process rewrite systems for the analysis of various kind of of infinite state systems. At the same time, it also shows that tree-automata are a powerful tool for the analysis of such process rewrite systems.

A Appendix

A.1 Proof of Lemma 3.4

($\Rightarrow$): by induction on the derivation $u \to$.
($\Leftarrow$) by induction on u. If $u = 0$ then $State(u) = \emptyset$. If $u = q \in Q$ then we assumed Δ has at least one rule $q \xrightarrow{a} q', v$. If u is some $u_1 \blacktriangleright u_2$, then either $State(u_1) \neq \emptyset$ or $State(u_2) \neq \emptyset$:
1. if $State(u_2) \neq \emptyset$ then $u_2 \to$ by ind. hyp. and then $u \to$ by R3-R4.
2. if $State(u_2) = \emptyset$ then $State(u_1) \neq \emptyset$, $u_1 \to$ by ind. hyp., and then $u \to$ by R5-R6. Observe that the condition on the application of R6 causes no problem.

A.2 Proof of Lemma 3.5

The ($\Rightarrow$) direction is obvious with rule R5.

For the ($\Leftarrow$) direction we pick $q \in Q$ and show by induction on $n \in \mathbb{N}$ that $u \blacktriangleright q \xrightarrow{n} v \blacktriangleright q$ implies $u \xrightarrow{!}{}^* v$:

1. $n = 0$: then $u \blacktriangleright q = v \blacktriangleright q$. It follows that $u = v$ and $u \xrightarrow{!}{}^* v$.

2. $n > 0$: then $u \blacktriangleright q \xrightarrow{n-1} t \to v \blacktriangleright q$. t must be some $t_1 \blacktriangleright t_2$ (Lemma 3.1) and $t_2 \xrightarrow{i} q$ for some $0 \leq i \leq 1$. Necessarily $i = 0$ (Lemma 3.2) and then $t_2 = q$. $t \to v \blacktriangleright q$ is obtained by R5 since $State(q) \neq \emptyset$ rules out R6. Hence $t_1 \xrightarrow{!} v$. We conclude by noting that the ind. hyp. gives $u \xrightarrow{!}{}^* t_1$.

A.3 Proof of Lemma 3.6

($\Leftarrow$): Assuming $v \xrightarrow{!}{}^* v'$ and $t \to^* t'$, we have $v \blacktriangleright t \xrightarrow{!}{}^* v \blacktriangleright t'$ by R3-R4 and $v \blacktriangleright t' \xrightarrow{!}{}^* v' \blacktriangleright t'$ by R5.
($\Rightarrow$): Assume $v \blacktriangleright t \xrightarrow{!} v' \blacktriangleright t'$. This was obtained by R3, R4 or R5, so that ($v \xrightarrow{!} v'$ and $t = t'$), or ($v = v'$ and $t \to t'$). Hence $v \xrightarrow{!}{}^* v'$ and $t \to^* t'$.

If now $v \blacktriangleright t \xrightarrow{n.!} v' \blacktriangleright t'$ for some $n \in \mathbb{N}$, the previous reasoning and an easy induction on n gives $v \xrightarrow{!}{}^* v'$ and $t \to^* t'$.

A.4 Proof of Lemma 3.7

($\Leftarrow$): one gets $v \blacktriangleright t \xrightarrow{!}{}^* v \blacktriangleright t'$ by R3-R4, and follows with $v \blacktriangleright t' \to^* v' \blacktriangleright t'$ by R5 if $v \xrightarrow{!}{}^* v'$, or by R5-R6 if $v \to^* v'$ and $t' \not\to$.
($\Rightarrow$): we have either (a) $v \blacktriangleright t \xrightarrow{!}{}^* v' \blacktriangleright t'$ or (b) $v \blacktriangleright t \to^* v_1 \blacktriangleright t_1 \xrightarrow{?} v_2 \blacktriangleright t_2 \to^* v' \blacktriangleright t'$. In case (a), Lemma 3.6 concludes. In case (b), rule R6 requires $State(t_1) = \emptyset$ so that $t_1 \not\to$. It follows that $t' = t_1$ and $t' \not\to$.

A.5 Proof of Lemma 3.8

($\Rightarrow$): the first step of $q \to^*$ must be some $q \to c \blacktriangleright u$ obtained by R1-R2 via some $q \to_\Delta c, u$ in Δ. Then $c \blacktriangleright u \to^* v \blacktriangleright t$ and Lemma 3.7 concludes.
($\Leftarrow$): this direction is obvious by combining R1-R2 and Lemma 3.7.

A.6 Sketch Proof of Lemma 3.9

This extends Lemma 3.6 exactly like the previous lemma extended Lemma 3.7.

A.7 Proof of Lemma 8.2

We prove the lemma by induction on v. Assume $u \swarrow v \searrow w$ with $u \neq w$, write v under the form $c \blacktriangleright^n (v_1, \ldots, v_n)$, and consider the following cases:

- If $v \searrow u$ using rule (E1), then $v = 0 \blacktriangleright u$ and, since $u \neq w$, $w = 0 \blacktriangleright u'$ with $u \searrow u'$. Then $u \searrow u' \swarrow w$.
- If $v \searrow u$ using rule (E2) on v_i, then if $v \searrow w$ also uses rule (E2) (on v_j with $j \neq i$) it is easy to show $u \searrow\!\swarrow w$. If $v \searrow w$ uses rule (E3), then $u \searrow\!\swarrow w$ is equally obvious.
- If $v \searrow u$ using rule (E3), then $u = c \blacktriangleright^n (v_1, \ldots, v_{i-1}, u_i, v_{i+1}, \ldots, v_n)$ with $v_i \searrow u_i$. The only interesting case for $v \searrow w$ is when $w = c \blacktriangleright^n (v_1, \ldots, v_{i-1}, w_i, v_{i+1}, \ldots, v_n)$ with $v_i \searrow w_i$ (the other cases are mirror images of cases we already considered). Here, since $u_i \neq w_i$, the ind. hyp. gives $u_i \searrow v'' \swarrow w_i$ for some v'' and we deduce $u \searrow\!\swarrow w$.

A.8 Proof of Lemma 8.3

We assume $u \rightleftarrows v \searrow w$ and prove the Lemma by induction on w. Write w under the form $c \blacktriangleright^n (w_1, \ldots, w_n)$. If $n = 0$ then $v = 0 \blacktriangleright c$ and no u exists s.t. $u \rightleftarrows v$. Thus $n > 0$ and we now consider all cases for $v \searrow w$:

- If $v \searrow w$ by rule (E1), then $v = 0 \blacktriangleright w$ and $u = 0 \blacktriangleright w'$ with $w' \rightleftarrows w$. We are done since $u \searrow w'$.

17

- If $v \searrow\!\!\!\!\searrow w$ by rule (E2), then v is some $c \blacktriangleright^{n-m+1} (w_1, \ldots, w_{i-1}, 0 \blacktriangleright^m (w_i, \ldots, w_{i+m-1}), w_{i+m}, \ldots, w_n)$ with m possibly 0. Now there are several cases for $u \rightleftarrows v$:

  If $u \rightleftarrows v$ by rule (P2), or by rule (P1) in a way that does not touch the $0 \blacktriangleright^m (w_i, \ldots, w_{i+m-1})$ subterm of v, then it is easy to see that $u \searrow\!\!\!\!\searrow \rightleftarrows w$.

  Otherwise the $0 \blacktriangleright^m (w_i, \ldots, w_{i+m-1})$ subterm of v is swapped with w_{i-1} or w_{i_m}. In the first case u is $c \blacktriangleright^{n-m+1} (w_1, \ldots, w_{i-2}, 0 \blacktriangleright^m (w_i, \ldots, w_{i+m-1}), w_{i-1}, w_m, \ldots, w_n)$ and $u \searrow\!\!\!\!\searrow v' = c \blacktriangleright^n (w_1, \ldots, w_{i-2}, w_i, \ldots, w_{i+m-1}, w_{i-1}, w_m, \ldots, w_n)$ works since $v' \overset{*}{\rightleftarrows} w$ with m uses of rule (P1). The second case is similar.
- If $v \searrow\!\!\!\!\searrow w$ by rule (E3), v is $c \blacktriangleright^n (w_1, \ldots, w_{i-1}, w'_i, w_{i+1}, \ldots, w_n)$ for some i and w'_i s.t. $w'_i \searrow\!\!\!\!\searrow w_i$. The cases where $u \rightleftarrows v$ by rule (E1), or by rule (E2) on a subterm different from w'_i, are easy to deal with.

  The interesting case is when $u = c \blacktriangleright^n (w_1, \ldots, w_{i-1}, w''_i, w_{i+1}, \ldots, w_n)$ and $w''_i \rightleftarrows w'_i$. Then the induction hypothesis applied on $w''_i \rightleftarrows w'_i \searrow\!\!\!\!\searrow w_i$ yields $w''_i \searrow\!\!\!\!\searrow v'' \overset{*}{\rightleftarrows} w_i$ for some v'', and we deduce $u \searrow\!\!\!\!\searrow v' \overset{*}{\rightleftarrows} w$ with $v' = c \blacktriangleright^n (w_1, \ldots, w_{i-1}, v'', w_{i+1}, \ldots, w_n)$.

A.9 Proof of Proposition 8.5

There only remains to prove the ($\Rightarrow$) direction of Prop. 8.5. We start with the following lemma:

Lemma A.1 *$u \leftrightsquigarrow u'$ implies $c \blacktriangleright^n (\ldots, u, \ldots) \leftrightsquigarrow c \blacktriangleright^n (\ldots, u', \ldots)$.*

Proof. By induction on the length of the derivation $t_i \leftrightsquigarrow u$. For the base case, assume $u \searrow\!\!\!\!\searrow u'$ (resp. $u \curvearrowright u'$, $u \rightleftarrows u'$): one concludes using rule (E3) (resp. (E2), (P2)). □

We are now ready to prove that $\mathcal{S}(u) = \mathcal{S}(v)$ entails $u \leftrightsquigarrow v$. The proof is by induction on $|u| + |v|$. We assume that u and v are resp. $c \blacktriangleright^n (u_1, \ldots, u_n)$ and $c' \blacktriangleright^m (v_1, \ldots, v_m)$ and consider several cases:

- If $c \in Q$ and $c' = 0$, then $\mathcal{S}(u) = (c, \sum_i \mathcal{S}(u_i))$ and $\mathcal{S}(v) = \sum_j \mathcal{S}(v_j)$. Hence there is some k s.t. $\mathcal{S}(v_k) = \mathcal{S}(u)$ and for all $j \neq k$, $\mathcal{S}(v_j) = \emptyset$. By ind. hyp. we have $v_k \leftrightsquigarrow u$ and $v_j \leftrightsquigarrow 0$ for $j \neq k$. Thus $v \leftrightsquigarrow 0 \blacktriangleright^m (0, \ldots, 0, u, 0, \ldots 0)$ by Lemma A.1. Then $v \leftrightsquigarrow 0 \blacktriangleright u$ by (E2) and $v \leftrightsquigarrow u$ by (E1). The case where $c = 0$ and $c' \in Q$ is symmetric.
- If $c = 0 = c'$, then $\mathcal{S}(u) = \sum_i \mathcal{S}(u_i)$ and $\mathcal{S}(v) = \sum_j \mathcal{S}(v_j)$. If $c, c' \in Q$, then $\mathcal{S}(u) = (c, \sum_i \mathcal{S}(u_i))$ and $\mathcal{S}(v) = (c', \sum_j \mathcal{S}(v_j))$. In both cases, $c = c'$ and $\sum_i \mathcal{S}(u_i) = \sum_j \mathcal{S}(v_j)$.

  Now, if each u_i and each v_j has the form $q \blacktriangleright^\alpha (\ldots)$ with $q \in Q$, then $n = m$ and there is a bijective h s.t. $\mathcal{S}(u_i) = \mathcal{S}(v_{h(i)})$. By ind. hyp., $u_i \leftrightsquigarrow v_{h(i)}$, then $u \leftrightsquigarrow c \blacktriangleright^n (v_{h(1)}, \ldots, v_{h(n)})$ by Lemma A.1, then $u \leftrightsquigarrow v$ by (P1).

  Otherwise some u_i or v_j has the form $0 \blacktriangleright^k (w_1, \ldots, w_k)$, we use rule (E2)

18

to flatten the corresponding term in u or v and we repeat the process until no such u_i and v_j exists. Eventually we obtain $u \searrow^* u'$ and $v \searrow^* v'$ with u' and v' having the form of the previous subcase, concluding the proof.

References

[1] Aceto, L., W. J. Fokkink and C. Verhoef, *Structural operational semantics*, in: J. A. Bergstra, A. Ponse and S. A. Smolka, editors, *Handbook of Process Algebra*, Elsevier Science, 2001 pp. 197–292.

[2] Baeten, J. C. M. and W. P. Weijland, "Process Algebra," Cambridge Tracts in Theoretical Computer Science **18**, Cambridge Univ. Press, 1990.

[3] Bouajjani, A., B. Jonsson, M. Nilsson and T. Touili, *Regular model checking*, in: *Proc. 12th Int. Conf. Computer Aided Verification (CAV'2000), Chicago, IL, USA, July 2000*, Lecture Notes in Computer Science **1855** (2000), pp. 403–418.

[4] Bukart, O., D. Caucal, F. Moller and B. Steffen, *Verification on infinite structures*, in: J. A. Bergstra, A. Ponse and S. A. Smolka, editors, *Handbook of Process Algebra*, Elsevier Science, 2001 pp. 545–623.

[5] Caucal, D., *On word rewriting systems having a rational derivation*, in: *Proc. 3rd Int. Conf. Foundations of Software Science and Computation Structures (FOSSACS'2000), Berlin, Germany, Mar.-Apr. 2000*, Lecture Notes in Computer Science **1784**, 2000, pp. 48–62.

[6] Comon, H., M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison and M. Tommasi, *Tree Automata Techniques and Applications* (1997–99), a preliminary version of this electronic book is available at `http://www.grappa.univ-lille3.fr/tata`.

[7] Comon, H. and Y. Jurski, *Timed automata and the theory of real numbers*, in: *Proc. 10th Int. Conf. Concurrency Theory (CONCUR'99), Eindhoven, The Netherlands, Aug. 1999*, Lecture Notes in Computer Science **1664** (1999), pp. 242–257.

[8] Esparza, J., *Petri nets, commutative context-free grammars, and basic parallel processes*, Fundamenta Informaticae **31** (1997), pp. 13–25.

[9] Esparza, J. and J. Knoop, *An automata-theoretic approach to interprocedural data-flow analysis*, in: *Proc. 2nd Int. Conf. Foundations of Software Science and Computation Structures (FOSSACS'99), Amsterdam, The Netherlands, Mar. 1999*, Lecture Notes in Computer Science **1578** (1999), pp. 14–30.

[10] Esparza, J. and A. Podelski, *Efficient algorithms for pre∗ and post∗ on interprocedural parallel flow graphs*, in: *Proc. 27th ACM Symp. Principles of Programming Languages (POPL'2000), Boston, MA, USA, Jan. 2000*, 2000, pp. 1–11.

[11] Finkel, A. and Ph. Schnoebelen, *Well structured transition systems everywhere!*, Theoretical Computer Science **256** (2001), pp. 63–92.

[12] Klop, J. W., *Term rewriting systems*, in: S. Abramsky, D. M. Gabbay and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, vol.2. Background: Computational Structures*, Oxford Univ. Press, 1992 pp. 1–116.

[13] Kouchnarenko, O. and Ph. Schnoebelen, *A model for recursive-parallel programs*, in: *Proc. 1st Int. Workshop on Verification of Infinite State Systems (INFINITY'96), Pisa, Italy, Aug. 1996*, Electronic Notes in Theor. Comp. Sci. **5** (1997), available at `http://www.lsv.ens-cachan.fr/Publis/PAPERS/`.

[14] Kupferman, O. and M. Y. Vardi, *An automata-theoretic approach to reasoning about infinite-state systems*, in: *Proc. 12th Int. Conf. Computer Aided Verification (CAV'2000), Chicago, IL, USA, July 2000*, Lecture Notes in Computer Science **1855** (2000), pp. 36–52.

[15] Kushnarenko, O. and Ph. Schnoebelen, *A formal framework for the analysis of recursive-parallel programs*, in: *Proc. 4th Int. Conf. Parallel Computing Technologies (PaCT'97), Yaroslavl, Russia, Sep. 1997*, Lecture Notes in Computer Science **1277** (1997), pp. 45–59.

[16] Lugiez, D. and Ph. Schnoebelen, *Decidable first-order transition logics for PA-processes*, in: *Proc. 27th Int. Coll. Automata, Languages, and Programming (ICALP'2000), Geneva, Switzerland, July 2000*, Lecture Notes in Computer Science **1853** (2000), pp. 342–353.

[17] Lugiez, D. and Ph. Schnoebelen, *The regular viewpoint on PA-processes* (2000), to appear in *Theor. Comp. Sci.*, available at `http://www.lsv.ens-cachan.fr/Publis/PAPERS/`.

[18] Mayr, R., *Process rewrite systems*, Information and Computation **156** (2000), pp. 264–286.

[19] Mayr, R., *Decidability of model checking with the temporal logic EF*, Theoretical Computer Science **256** (2001), pp. 31–62.

[20] Moller, F., *Infinite results*, in: *Proc. 7th Int. Conf. Concurrency Theory (CONCUR'96), Pisa, Italy, Aug. 1996*, Lecture Notes in Computer Science **1119** (1996), pp. 195–216.

[21] Schnoebelen, Ph., *Decomposable regular languages and the shuffle operator*, EATCS Bull. **67** (1999), pp. 283–289.

[22] Vardi, M. Y., *Alternating automata: Checking truth and validity for temporal logics*, in: *Proc. 14th Int. Conf. Automated Deduction (CADE'97), Townsville, North Queensland, Australia, July 1997*, Lecture Notes in Computer Science **1249** (1997), pp. 191–206.

Electronic Notes in Theoretical Computer Science 52 No. 1 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume52.html 11 pages

Adequate Sets of Temporal Connectives in CTL

Alan Martin [1,2]

Department of Mathematics
University of Ottawa
Ottawa, Ontario, Canada

Abstract

An *adequate set* of temporal connectives for CTL is a subset of the logic's temporal connectives that is sufficient to express equivalents for all CTL formulas.

In this paper, a characterization of all such adequate sets is presented. Specifically, it is shown that a subset of CTL's temporal connectives is adequate if and only if it contains one of $\{\mathrm{AX}, \mathrm{EX}\}$, one of $\{\mathrm{EG}, \mathrm{AF}, \mathrm{AU}\}$, and EU.

The proof requires, among other things, the analysis of a certain class of models, the reflexive models. These models have the desirable property that several connectives become redundant, thus simplifying the analysis.

1 Introduction

Recall the definition of an adequate set of connectives:

Definition 1.1 An *adequate set* of connectives for a logic is a subset S of its connectives such that every formula of the logic is equivalent to some formula of the sublogic generated by S.

Why should we be interested in adequate sets? There are several reasons. One is the tradeoff between expressive power and difficulty of proof and implementation. For instance, it is certainly easier to express typical propositional sentences using $\neg$, $\wedge$, $\vee$, and $\rightarrow$ rather than using a single adequate connective such as the Sheffer stroke (NAND). But inductive proofs and computer implementations will be simpler if there are fewer connectives. So it is of interest to have ways to express formulas using a larger set of connectives in terms of a smaller set.

[1] Research supported in part by NSERC and the University of Ottawa.
[2] Email: martin@igs.net

Adequate sets of connectives are also connected to the problem of classifying sublogics generated by subsets of connectives. Determining which of these are distinct up to equivalence of formulas is equivalent to describing the adequate sets of connectives for each of the sublogics. In practice it is usually only necessary to consider the original logic and a few of its sublogics.

Most reasonable logics have a substitution theorem, which states that uniform replacement of an atom with an arbitrary formula preserves validity and equivalences. If this is the case then an adequate set of connectives can be used to express equivalents not only for formulas, but also for formula schemes.

In this paper, we are particularly interested in Computation Tree Logic, or CTL, a branching-time temporal logic due to Clarke and Emerson [1]. Some earlier work on branching-time temporal logics [4,7] allowed the application of path quantifiers to linear-time temporal logic (LTL) formulas. A very general such logic is CTL* (see, for example, [6]), a logic with good expressive power but an NP-complete model-checking problem [1]. CTL was developed as a logic that provides much expressive power, but unlike CTL*, admits an efficient model-checking algorithm.

CTL has been used as the basis for model checking systems in practice. The model checker SMV [9] is based on CTL, and it has been used to verify properties of various systems, for example a cache coherence protocol for an IEEE bus architecture standard (see [2]).

In this paper, we will define the formulas of CTL using the BNF:

$$\phi ::= \mathrm{Atom} \mid \top \mid \bot \mid \neg\,\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \to \phi \mid \mathrm{A}\,\alpha \mid \mathrm{E}\,\alpha$$
$$\alpha ::= \mathrm{X}\,\phi \mid \mathrm{F}\,\phi \mid \mathrm{G}\,\phi \mid \phi\,\mathrm{U}\,\phi$$

Here ϕ defines the *state formulas*, while α defines the *path formulas*. This mutually recursive definition is derived from CTL*, of which CTL is a sublogic; only the state formulas are considered to be CTL formulas, but the use of path formulas simplifies the semantics. Accordingly, the temporal connectives of CTL are considered to be the combinations of path quantifiers A and E with modal connectives X, F, G, and U. That is, there are unary connectives $\{\mathrm{AX}, \mathrm{AF}, \mathrm{AG}, \mathrm{EX}, \mathrm{EF}, \mathrm{EG}\}$ and binary connectives $\{\mathrm{AU}, \mathrm{EU}\}$.

A model $\mathcal{M}$ for CTL consists of a set S of *states*, a *labelling function* $L : S \to \mathrm{Atoms}$, and a *transition relation* $\mathrm{Tran} \subset S \times S$, such that for every $s \in S$, there is $s' \in S$ such that $(s, s') \in \mathrm{Tran}$.

The *satisfaction relation* $\models$ is defined by mutual recursion over state formulas and path formulas, according to the following rules, where $\mathcal{M}$ is a model, $s \in S$, and $\pi = (s_0 \to s_1 \to s_2 \to \cdots)$ is a path in $\mathcal{M}$:

- $\mathcal{M}, s \models \mathrm{A}\,\alpha$ if and only if for all paths π' in $\mathcal{M}$ starting at s, $\mathcal{M}, \pi' \models \alpha$.
- $\mathcal{M}, s \models \mathrm{E}\,\alpha$ if and only if there exists a path π' in $\mathcal{M}$ starting at s such that $\mathcal{M}, \pi' \models \alpha$.
- $\mathcal{M}, \pi \models \mathrm{X}\,\varphi$ if and only if $\mathcal{M}, s_1 \models \varphi$.
- $\mathcal{M}, \pi \models \mathrm{F}\,\varphi$ if and only if there is $i \geq 0$ such that $\mathcal{M}, s_i \models \varphi$.

- $\mathcal{M}, \pi \models \mathrm{G}\,\varphi$ if and only if for all $i \geq 0$, $\mathcal{M}, s_i \models \varphi$.
- $\mathcal{M}, \pi \models \varphi\,\mathrm{U}\,\psi$ if and only if there exists $i \geq 0$ such that $\mathcal{M}, s_i \models \psi$ and for all j such that $0 \leq j < i$, $\mathcal{M}, s_j \models \varphi$.

The usual rules apply for propositional connectives and constants, and an atom p is satisfied at those states s with $p \in L(s)$.

This paper presents a complete characterization of the adequate sets of temporal connectives for CTL.

After this paper had been written, the author learned that the main result of Lemma 3.5 had been previously proven [8].

1.1 Acknowledgements

This paper was written as part of the author's M.Sc. thesis at the University of Ottawa under the supervision of Dr. Richard Blute. The author would like to thank Dr. Blute and the University of Ottawa, as well as Dr. Prakash Panangaden of McGill University who read an earlier version.

2 Adequacy Theorem

Theorem 2.1 *The adequate sets of temporal connectives for CTL are exactly those sets of temporal connectives containing at least one of* $\{\mathrm{AX}, \mathrm{EX}\}$*, at least one of* $\{\mathrm{EG}, \mathrm{AF}, \mathrm{AU}\}$*, and* EU*.*

The requirement for EU may seem surprising at first. However, the fact that there is no such requirement for AU does not give a contradiction, since AU and EU are not dual. If we allow boolean combinations of path formulas with the usual semantics (as in CTL*), we can write the path formula equivalence $\neg(\varphi \mathrm{U}\, \psi) \equiv ((\neg\, \psi)\mathrm{U}\,(\neg\, \varphi \wedge \neg\, \psi)) \vee \mathrm{G} \neg\, \psi$ (see, for example, [6]). However, when we apply a path quantifier to both sides of this equivalence, we get a CTL equivalence only if that quantifier distributes over $\vee$, that is, only for E (in which case we get equivalence (6), below), not for A.

One direction of the proof is straightforward: to prove that every such set of temporal connectives is in fact adequate.

Lemma 2.2 *The sets of temporal connectives specified in Theorem 2.1 are adequate.*

Proof. We claim that the following are equivalences in CTL:

(1) $\mathrm{AX}\,\varphi \equiv \neg\, \mathrm{EX} \neg\, \varphi$
(2) $\mathrm{AF}\,\varphi \equiv \neg\, \mathrm{EG} \neg\, \varphi$
(3) $\mathrm{AG}\,\varphi \equiv \neg\, \mathrm{EF} \neg\, \varphi$
(4) $\mathrm{AF}\,\varphi \equiv \mathrm{A}[\top\ \mathrm{U}\ \varphi]$
(5) $\mathrm{EF}\,\varphi \equiv \mathrm{E}[\top\ \mathrm{U}\ \varphi]$
(6) $\mathrm{A}[\varphi\ \mathrm{U}\ \psi] \equiv \neg\, \mathrm{E}[\neg\, \varphi\ \mathrm{U}\ (\neg\, \varphi \wedge \neg\, \psi)] \wedge \mathrm{AF}\,\psi$

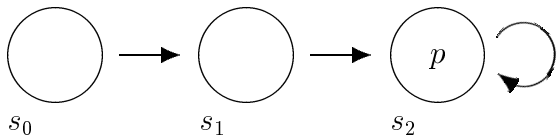


Fig. 1. Model $\mathcal{M}$ demonstrating inadequacy of S_1

For proofs of these and similar equivalences, see, for example, [6].

Given any set of connectives S as specified in Theorem 2.1 and any CTL formula φ, we can use equation (1) to write whichever of AX and EX does not occur in S in terms of the other one. Then, if one of EG or AF occurs in S, we can use equations (2) and (6) to write all of EG, AF, and AU in terms of EU and whichever of them occurs in S. Otherwise, AU occurs in S and we can use equations (2) and (4) to write EG and AF in terms of AU. Finally, we can use equations (3) and (5) to write AG and EF in terms of EU. In all cases we have found a formula φ' of the sublogic generated by S that is equivalent to the given formula φ. □

The other direction is more difficult. We must show that any set of temporal connectives not meeting the conditions of Theorem 2.1 is not adequate. Since a superset of an adequate set is adequate, it is enough to consider only three sets of temporal connectives:

$$S_1 := C \setminus \{\mathrm{AX}, \mathrm{EX}\}$$
$$S_2 := C \setminus \{\mathrm{EG}, \mathrm{AF}, \mathrm{AU}\}$$
$$S_3 := C \setminus \{\mathrm{EU}\}$$

where $C := \{\mathrm{AX}, \mathrm{AF}, \mathrm{AG}, \mathrm{AU}, \mathrm{EX}, \mathrm{EF}, \mathrm{EG}, \mathrm{EU}\}$ is the set of all CTL temporal connectives.

Lemma 2.3 *The set of temporal connectives $S_1 := C \setminus \{\mathrm{AX}, \mathrm{EX}\}$ is not adequate.*

Proof. We consider the CTL formula $\mathrm{EX}\,p$ and the model $\mathcal{M}$ shown in Figure 1.

There is only one path, $\pi = (s_0 \to s_1 \to s_2 \to s_2 \to \cdots)$, starting at s_0, and similarly there is only one path $\pi' = (s_1 \to s_2 \to s_2 \to \cdots)$ starting at s_1.

We have $\mathcal{M}, s_0 \not\models \mathrm{EX}\,p$ and $\mathcal{M}, s_1 \models \mathrm{EX}\,p$. But we claim that if φ is a formula not using AX or EX, then $\mathcal{M}, s_0 \models \varphi$ if and only if $\mathcal{M}, s_1 \models \varphi$, so in particular φ cannot be equivalent to $\mathrm{EX}\,p$. This will show that S_1 is not an adequate set.

We will prove this claim by structural induction on φ. We may assume by equivalences (1)–(6) that φ uses only the temporal connectives EG and EU.

The base case is if φ is an atom, $\top$, or $\bot$. In this case the claim is obvious, since s_0 and s_1 are identically labeled.

If φ has a propositional connective as its principal connective, then the claim follows trivially from the induction hypothesis.

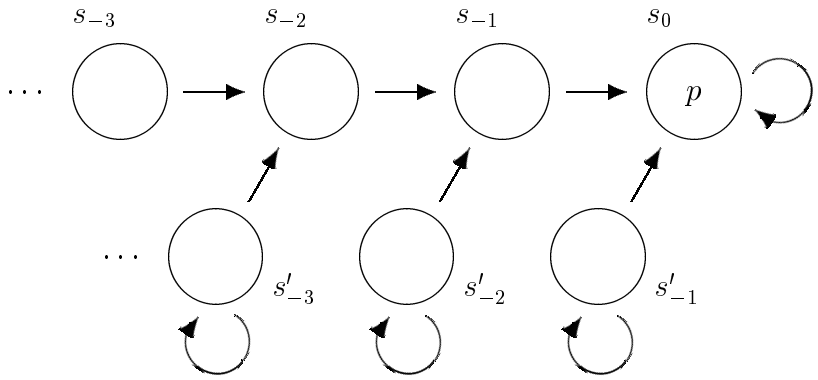


Fig. 2. Model $\mathcal{M}$ demonstrating inadequacy of S_2

If $\varphi = \mathrm{EG}\,\psi$, then $\mathcal{M}, s_0 \models \varphi$ if and only if $\mathcal{M}, s_i \models \psi$ for $i \in \{0, 1, 2\}$, and $\mathcal{M}, s_1 \models \varphi$ if and only if $\mathcal{M}, s_i \models \psi$ for $i \in \{1, 2\}$. But by the induction hypothesis, $\mathcal{M}, s_0 \models \psi$ if and only if $\mathcal{M}, s_1 \models \psi$; so we can conclude that $\mathcal{M}, s_0 \models \varphi$ if and only if $\mathcal{M}, s_1 \models \varphi$.

If $\varphi = \mathrm{E}[\psi\ \mathrm{U}\ \psi']$, then $\mathcal{M}, s_0 \models \varphi$ if and only if there exists $i \geq 0$ such that $\mathcal{M}, \pi_i \models \psi'$, and for all j with $0 \leq j < i$, $\mathcal{M}, \pi_j \models \psi$. (Here π_i denotes the i^{th} state of π, numbered from zero.) Using the induction hypothesis, ψ and ψ' each have the same truth value at s_0 and s_1. We consider cases: if $i < 2$, then $\mathcal{M}, s_0 \models \psi'$, and if $i \geq 2$, then $\mathcal{M}, s_0 \models \psi$ and $\mathcal{M}, s_2 \models \psi'$. Each of these implies $\mathcal{M}, s_0 \models \varphi$, so we can conclude that $\mathcal{M}, s_0 \models \varphi$ if and only if one of these conditions holds. We can do a similar case analysis and find that $\mathcal{M}, s_1 \models \varphi$ if and only if either $\mathcal{M}, s_1 \models \psi'$, or $\mathcal{M}, s_1 \models \psi$ and $\mathcal{M}, s_2 \models \psi'$. But using the induction hypothesis again, the two sets of conditions are equivalent, so again we conclude that $\mathcal{M}, s_0 \models \varphi$ if and only if $\mathcal{M}, s_1 \models \varphi$. This completes the induction. □

We have seen the first technique that will be needed to prove that a set S is not adequate: Find a CTL formula φ, which is claimed not to be equivalent to any formula in the sublogic generated by S. Then for any given CTL formula ψ with temporal connectives from S, find two states (in the same or different models) such that φ is true at one and false at the other, but ψ has the same truth value at both. (Similar techniques were used by Lamport [7].)

Lemma 2.4 *The set of temporal connectives* $S_2 := C \setminus \{\mathrm{EG}, \mathrm{AF}, \mathrm{AU}\}$ *is not adequate.*

Proof. In this case we will consider an infinite model $\mathcal{M}$. Its states are s_i for $i \leq 0$ and s'_i for $i < 0$. All states are labeled with $\emptyset$ except for s_0 which is labeled with $\{p\}$. There are transitions from s_i to s_{i+1}, from s'_i to s_{i+1}, from s'_i to s'_i for $i < 0$, and from s_0 to s_0. Part of this model is shown in Figure 2.

There is a unique path starting at any state s_i, which eventually reaches s_0 which satisfies p. So for all $i \leq 0$, $\mathcal{M}, s_i \models \mathrm{AF}\,p$. But starting at any state s'_i, there are infinitely many paths that may remain at s'_i for some time

but eventually transition to s_{i+1} and continue on to s_0, as well as a constant path remaining at s'_i forever. The latter path does not satisfy $\mathrm{F}\,p$, so for all $i < 0$, $\mathcal{M}, s'_i \not\models \mathrm{AF}\,p$. But we claim that for any formula φ not using EG, AF, or AU, there exists $n_\varphi < 0$ such that φ has the same truth value at all states s_i and s'_i with $i \leq n_\varphi$, so in particular φ cannot be equivalent to $\mathrm{AF}\,p$. This will show that S_2 is not an adequate set.

Again we proceed by structural induction on φ. We may assume, using the equivalences (1)–(6), that φ uses only the temporal connectives EX and EU. The base case, where φ is an atom, $\top$, or $\bot$, is trivial. If the principal connective of φ is propositional, we can take n_φ to be the minimum of the n values for the operand(s) of this connective and the claim clearly holds.

If $\varphi = \mathrm{EX}\,\psi$, let $n_\varphi := n_\psi - 1$. For $i \leq n_\varphi$, $\mathcal{M}, s_i \models \varphi$ if and only if $\mathcal{M}, s_{i+1} \models \psi$, and $\mathcal{M}, s'_i \models \varphi$ if and only if either $\mathcal{M}, s'_i \models \psi$ or $\mathcal{M}, s_{i+1} \models \psi$. But by the induction hypothesis, ψ has a constant truth value on s_i and s'_i for $i \leq n_\psi$, and $i \leq n_\varphi$ implies $i + 1 \leq n_\psi$. So we can conclude that φ has the same truth value on s_i and s'_i for $i \leq n_\varphi$ as ψ has on s_i and s'_i for $i \leq n_\psi$, and the claim holds.

If $\varphi = \mathrm{E}[\psi\,\mathrm{U}\,\psi']$, then by the induction hypothesis, ψ has a constant truth value on s_i and s'_i for $i \leq n_\psi$, and ψ' has a constant truth value on s_i and s'_i for $i \leq n_{\psi'}$. Let $n_\varphi := \min(n_\psi, n_{\psi'})$. If ψ' is true at the states s_i and s'_i for $i \leq n_\varphi$, then so is φ and the claim holds. Otherwise we may assume ψ' is false at all of these states, since it has a constant truth value on them. In this case, we consider ψ. If it is false at the states s_i and s'_i for $i \leq n_\varphi$, then so is φ and the claim holds. So we may also assume that ψ is true at all of these states. Now if any state s_i or s'_i for $i \leq n_\varphi$ satisfies φ, then the path from that state satisfying $\psi\,\mathrm{U}\,\psi'$ must go through $s_{n_\varphi+1}$, and we must have $\mathcal{M}, s_{n_\varphi+1} \models \varphi$. But every state s_i or s'_i with $i \leq n_\varphi$ has a path to $s_{n_\varphi+1}$ along which every state satisfies ψ; then all of these states satisfy φ. So either all of these states satisfy φ or none of them do, and in either case the conclusion holds. This completes the induction. □

This time we needed a slightly different approach. For any fixed finite n, every CTL formula is equivalent on models with at most n states to some formula using only the next-time connectives AX and EX. (This can be shown using fixed-point techniques; see, for example, [6].) So when proving that a set of temporal connectives that includes AX or EX is not adequate, it is necessary to consider arbitrarily large models. (Again, similar techniques were used in [7].)

Showing that S_3 is not adequate will use both of the techniques we have seen already. But first we need some results on reflexive models.

3 Reflexive models

Definition 3.1 A model $\mathcal{M}$ is *reflexive* if its transition relation is reflexive, i.e. if there is a loop on every state.

Reflexive models are of interest for a variety of reasons. In modal logics, reflexiveness corresponds to a meaningful axiom, and in models of computation, reflexiveness corresponds to allowing idling. But our interest in this property is simply that it will eliminate many of the difficulties involved in a proof that S_3 is not an adequate set.

When considering a subclass of models, it is natural to ask about equivalence (and validity) relative to this subclass. Since equivalence means having equal truth values in all models, a smaller class of models yields a coarser equivalence relation.

Definition 3.2 Let φ and ψ be CTL formulas. Suppose that for all reflexive models $\mathcal{M}$ and states s of $\mathcal{M}$, $\mathcal{M}, s \models \varphi$ if and only if $\mathcal{M}, s \models \psi$. Then we will say that φ and ψ are *equivalent on reflexive models*, and we will write $\varphi \equiv_r \psi$.

There is, as usual, a substitution theorem:

Theorem 3.3 *If $\varphi \equiv_r \varphi'$, $\psi \equiv_r \psi'$, and p is any atom, then $\varphi[\psi/p] \equiv_r \varphi'[\psi'/p]$.*

Proposition 3.4 *The following equivalences hold on reflexive models, where φ and ψ are arbitrary CTL formulas:*

(7) $\quad \mathrm{EG}\,\varphi \equiv_r \varphi$
(8) $\quad \mathrm{AF}\,\varphi \equiv_r \varphi$
(9) $\quad \mathrm{A}[\varphi\,\mathrm{U}\,\psi] \equiv_r \psi$

Proof. Suppose $\mathcal{M}$ is a reflexive model and s is a state in $\mathcal{M}$. Then $\mathcal{M}, s \models \mathrm{A}[\varphi\,\mathrm{U}\,\psi]$ if and only if for every path π starting at s, $\mathcal{M}, \pi \models \varphi\,\mathrm{U}\,\psi$. In particular, the constant path at s (which must exist since $\mathcal{M}$ is reflexive) satisfies $\varphi\mathrm{U}\psi$, so $\mathcal{M}, s \models \psi$. Conversely, if $\mathcal{M}, s \models \psi$, clearly $\mathcal{M}, s \models \mathrm{A}[\varphi\mathrm{U}\psi]$. So we have proved (9). The other cases are similar. □

We have shown that the connectives EG, AF, and AU are redundant on reflexive models. This will make the job of proving that S_3 is not an adequate set much easier.

On the other hand, the connectives AG, EF, and EU behave exactly the same on any model $\mathcal{M}$ and its reflexive closure $\mathcal{M}'$ (which has the same states as $\mathcal{M}$). That is, any CTL formula using only these three temporal connectives has the same truth values on corresponding states of $\mathcal{M}$ and $\mathcal{M}'$. This can be proved by induction; the base case and the cases AG and EF are obvious. The case EU is more interesting. Suppose φ and ψ have the same truth values on corresponding states of $\mathcal{M}$ and $\mathcal{M}'$. Clearly, for any path π in $\mathcal{M}$, $\mathcal{M}, \pi \models \varphi\,\mathrm{U}\,\psi$ if and only if $\mathcal{M}', \pi \models \varphi\,\mathrm{U}\,\psi$. Conversely, if π' is a path in $\mathcal{M}'$ satisfying $\varphi\,\mathrm{U}\,\psi$, define a path π in $\mathcal{M}$ with the same starting state as π' by eliminating loop edges from π' up to the first state satisfying ψ. This gives a finite path in $\mathcal{M}$ since all non-loop edges of $\mathcal{M}'$ are edges of $\mathcal{M}$, and it can

be extended arbitrarily to an infinite path. Clearly we have $\mathcal{M}, \pi \models \varphi \,\mathrm{U}\, \psi$. Using the definition of satisfaction for E, the inductive step for EU follows.

Lemma 3.5 *The set of temporal connectives $S_3 := C \setminus \{\mathrm{EU}\}$ is not adequate.*

Proof. We will consider the formula $\mathrm{E}[p \,\mathrm{U}\, q]$. Suppose φ is an arbitrary formula not using the connective EU; we want to show that it is not equivalent to $\mathrm{E}[p \,\mathrm{U}\, q]$. This will show that S_3 is not an adequate set and thus complete the proof of Theorem 2.1.

We may assume, using our earlier equivalences (1)–(6), that φ uses only the connectives EX, EF, and AU. (We cannot eliminate EF using equivalence (5) because it would leave us with EU, nor can we replace AU with another connective using equivalence (6) for the same reason.)

Now using equivalence (9) and Theorem 3.3, we can replace each occurrence of AU in φ with its second operand to obtain a formula φ' using only the connectives EX and EF with $\varphi \equiv_r \varphi'$. We may further assume that none of the subformulas of φ' with principal connective EF are equivalent in reflexive models to $\bot$, without loss of generality since we could replace any such subformulas with $\bot$ and still have $\varphi \equiv_r \varphi'$.

Let $\varphi_1, \varphi_2, \ldots \varphi_m$ be the subformulas of φ' with principal connective EF. By assumption, none of them is equivalent to $\bot$ in reflexive models, so let $\mathcal{M}_i$ be a reflexive model and t_i a state of that model such that $\mathcal{M}_i, t_i \models \varphi_i$, for $1 \leq i \leq m$. Now we will define two reflexive models $\mathcal{M}$ and $\mathcal{M}'$ as follows: $\mathcal{M}$ will consist of the disjoint union of the states of the $\mathcal{M}_i$, together with states r_i for $0 \leq i \leq m$ and s_i for $i \leq 0$. The labelling of the states from the $\mathcal{M}_i$ will be unchanged, the labelling of the states r_i will be $\emptyset$, and the labelling of the states s_i will be $\{p\}$. The transitions of the $\mathcal{M}_i$ will be retained, and there will also be transitions from r_i to t_i for $1 \leq i \leq m$, from s_i to s_{i+1} for $i < 0$, from s_0 to r_i for $0 \leq i \leq m$, and from every state to itself. This model is shown in Figure 3. $\mathcal{M}'$ will be identical to $\mathcal{M}$, with its states distinguished by primes, except that the labelling of r'_0 will be $\{q\}$.

Clearly $\mathcal{M}, s_i \not\models \mathrm{E}[p \,\mathrm{U}\, q]$ and $\mathcal{M}', s'_i \models \mathrm{E}[p \,\mathrm{U}\, q]$ for all $i \leq 0$. Indeed, in the former case every path from s_i must either remain among the s_j, or eventually reach one of the r_j, and in either case it does not satisfy $p \,\mathrm{U}\, q$. In the latter case, the path $s'_i \to s'_{i+1} \to \cdots \to s'_0 \to r'_0 \to r'_0 \to \cdots$ starting at s'_i satisfies $p \,\mathrm{U}\, q$.

But we claim that for any subformula ψ of φ', there exists $n_\psi \leq 0$ such that ψ has a constant truth value on all states s_i and s'_i with $i \leq n_\psi$. We again use induction. The base case of atoms, $\top$, or $\bot$ is trivial. Also, as in the proof of Lemma 2.4, if ψ has a propositional principal connective, we can take n_ψ to be the minimum of the n values of this connective's operand(s). So only the temporal cases are left.

If $\psi = \mathrm{EF}\,\psi'$, then $\psi = \varphi_k$ for some k. In this case $\mathcal{M}, t_k \models \psi$ and $\mathcal{M}', t'_k \models \psi$. But this means that there are states s and s' reachable from t_k and t'_k respectively with $\mathcal{M}, s \models \psi'$ and $\mathcal{M}', s' \models \psi'$. But by construction, for

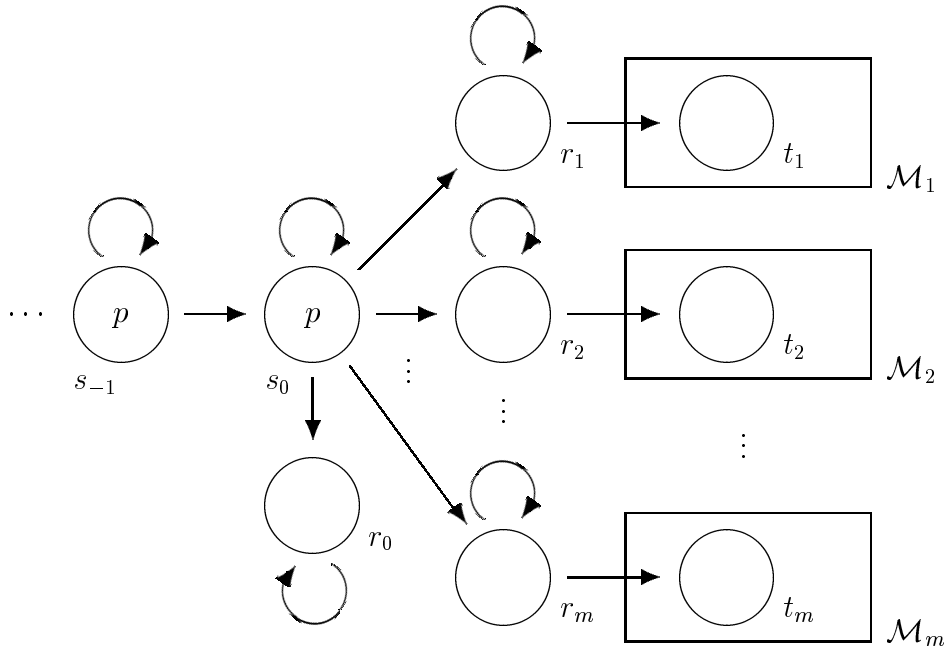


Fig. 3. Model $\mathcal{M}$ demonstrating inadequacy of S_3

any $i \leq 0$, t_k is reachable from s_i by the path $s_i \to s_{i+1} \to \cdots \to s_0 \to r_k \to t_k$, so by concatenating paths, s is reachable from s_i; similarly s' is reachable from s'_i. So $\mathcal{M}, s_i \models \psi$ and $\mathcal{M}', s'_i \models \psi$ for all $i \leq 0$; so we can take $n_\psi := 0$ and the claim is satisfied.

Finally we have the case $\psi = \mathrm{EX}\,\psi'$. But this case is similar to the EX case in the proof of Lemma 2.4, and the same argument holds. This completes the induction.

Since our models $\mathcal{M}$ and $\mathcal{M}'$ are reflexive by construction, we can conclude that $\varphi' \not\equiv_r \mathrm{E}[p\,\mathrm{U}\,q]$. Since $\equiv_r$ is an equivalence relation, this implies $\varphi \not\equiv_r \mathrm{E}[p\,\mathrm{U}\,q]$; *a fortiori*, $\varphi \not\equiv \mathrm{E}[p\,\mathrm{U}\,q]$, as required. □

Thus we have completed the proof of Theorem 2.1.

Laroussinie [8] gives a much more direct proof of Lemma 3.5 using just a single CTL model and without using results on reflexive models. The proof given here was developed independently by the author.

4 Further questions

We have characterized the adequate sets of temporal connectives for CTL. Where do we go from here? There are several possibilities.

- *Characterize the sublogics generated by subsets of connectives.*
  We have identified the subsets of the set C of temporal connectives that are adequate for CTL; but what about the sublogics generated by subsets that are not adequate? Some of them are equivalent in terms of expressive power, others are not. The equivalences (1)–(6), together with Theorem 2.1,

go a long way toward characterizing these sublogics. The key fact is that if $S, S' \subset C$ are two sets of connectives generating equivalent sublogics, then $S \cup S''$ and $S' \cup S''$ generate equivalent sublogics for any $S'' \subset C$. This means that the pairs of nonequivalent sublogics consisting of CTL itself together with the inadequate sets of Lemmas 2.3, 2.4, and 3.5 induce additional pairs of nonequivalent sublogics generated by smaller sets of connectives. However, there is still some more work to be done, in particular to determine whether S and $S \cup \{\mathrm{AU}\}$ can generate nonequivalent sublogics for S containing EG or AF but not EU.

- *Consider other types of sublogics.*

  There is also the question of sublogics not generated by subsets of connectives. For example, some recent research on this topic by Etessami and Wilke is [3], in which sublogics of LTL are defined by limiting the nesting depth of certain connectives. The possibilities for defining such sublogics appear endless. Some of them are interesting in their own right, as in [3], and it may also be an interesting question whether or not such sublogics can be treated systematically.

- *Consider combinations of temporal logic and linear logic.*

  Another possibility is to consider the effect of changing the base logic, specifically using Girard's linear logic [5] instead of classical logic. A CTL-like semantics can be defined using trees whose nodes are labeled with models of linear logic, such as phase spaces. The question of expressiveness and adequate subsets of connectives applies in this setting as well, and it may be an interesting question given the unique properties of linear logic such as resource sensitivity.

References

[1] Clarke, E. M., and E. A. Emerson, *Design and synthesis of synchronization skeletons using branching time temporal logic*, "Logics of Programs Workshop," Lecture Notes in Computer Science **131**, Springer Verlag, 1981, 52–71.

[2] Clarke, E. M., O. Grumberg, and D. A. Peled, "Model Checking," The MIT Press, Cambridge, Massachusetts, 1999.

[3] Etessami, K., and T. Wilke, *An until hierarchy and other applications of an EF game for temporal logic*, preprint, 1998, 26pp.

[4] Gabbay, D., A. Pnueli, et al., *The temporal analysis of fairness*, 7$^{\mathrm{th}}$ Annual ACM Symposium on Principles of Programming Languages, 1980.

[5] Girard, J.-Y., *Linear Logic*, Theoretical Computer Science **50** (1987), 1–102.

[6] Huth, M., and M. Ryan, "Logic in Computer Science: Modelling and reasoning about systems," Cambridge University Press, Cambridge, UK, 2000, 387pp. With associated web site `http://www.cs.bham.ac.uk/research/lics/`.

[7] Lamport, L., *"Sometime" is sometimes "Not Never,"* 7th Annual ACM Symposium on Principles of Programming Languages, 1980.

[8] Laroussinie, F., *About the expressive power of CTL combinators*, Information Processing Letters **54** (1995), 343–345.

[9] McMillan, K. L., "Symbolic Model Checking: An Approach to the State Explosion Problem," Kluwer Academic Publishers, 1993.

On logical and concurrent equivalences

J. C. Bradfield and S. B. Fröschle

Laboratory for Foundations of Computer Science, Division of Informatics, King's Bldgs, University of Edinburgh, Edinburgh EH9 3JZ, UK.
{jcb,sib}@dcs.ed.ac.uk

Abstract

We consider modal analogues of Hintikka et al.'s 'independence-friendly first-order logic', and discuss their relationship to equivalences previously studied in concurrency theory.

Key words: Independence, concurrency, branching quantifiers, modal logic.

1 Introduction

In [1], Alur, Henzinger and Kupfermann introduced Alternating Temporal Logic, based on certain imperfect information games, in which independent 'teams' synchronize. In [3], the first author proposed the application of logics based on Henkin quantifiers to modal logic in computer science; such logics include ATL, but also allow more powerful forms of expression. In that paper, we argued that making sense of such logics required some notion of locality in processes. After establishing some basic facts about such logics, we left open the obvious question of how such logics relate to established notions of independence and concurrency in computer science.

In this paper, we first interpret Henkin modal logics in a setting without locality (at least, without explicit locality), and then relate them to some of the natural true concurrent notions in the literature. The results here are preliminary, but, we believe, go some way towards a satisfactory explanation, and open up many further questions.

2 Henkin quantifiers and independence-friendly logic

We give a brief summary of the notions of Henkin quantifier and independence-friendly logic.

A *branching quantifier* Q is a set $\{x_1, \ldots, x_m, y_1, \ldots, y_n\}$ of variables, carrying a partial order $\prec$; the x_i are universal, the y_i existential. The semantics of $Q\phi$ is defined to be that of $\exists f_1 \ldots f_n . \forall x_1 \ldots x_m . \phi[f_i(y_i\downarrow)/y_i]$, where $y_i\downarrow$ is the list of variables $\prec y_i$, and $[\cdot/\cdot]$ denotes syntactic substitution: thus f_i is a Skolem function for y_i, but it refers only to variables preceding y_i in the partial order.

In particular, the *Henkin quantifier* $\begin{smallmatrix}\forall & \exists \\ \forall & \exists\end{smallmatrix} = \{x_1, x_2, y_1, y_2\}$ with $x_i \prec y_i$ is written $\begin{smallmatrix}\forall x_1 & \exists y_1 \\ \forall x_2 & \exists y_2\end{smallmatrix}$; thus $\begin{smallmatrix}\forall x & \exists y \\ \forall u & \exists v\end{smallmatrix}\phi(x, y, u, v)$ is equivalent by definition to $\exists f, g . \forall x, u . \phi(x, f(x), u, g(u))$.

Henkin quantifiers turn out to have existential second-order power, and are thus a strong operator to add to one's logic.

An alternative way of giving semantics to branching quantifiers is via games. Recall the Hintikka model-checking game for first-order logic (in positive form): given a formula ψ and a structure M, a position is a subformula $\phi(\vec{x})$ of ψ together with a *deal* for ϕ, that is, an assignment of values $\vec{v}$ to its free variables $\vec{x}$. At a position $(\forall x . \phi_1, \vec{v})$, Abelard chooses a value v for x, and play moves to the position $(\phi_1, \vec{v} \cdot v)$; similarly Eloise moves at $\exists x . \phi$. At $(\phi_1 \wedge \phi_2, \vec{v})$, Abelard chooses a conjunct ϕ_i, and play moves to $(\phi_i(\vec{x}'), \vec{v}')$, where $\vec{x}', \vec{v}'$ are $\vec{x}, \vec{v}$ restricted to the free variables of ϕ_i; and at $(\phi_1 \vee \phi_2, \vec{v})$, Eloise similarly chooses a disjunct. A play of the game terminates at (negated) atoms $(P(\vec{x}), \vec{v})$ (resp. $(\neg P(\vec{x}), \vec{v})$), and is won by Eloise (resp. Abelard) iff $P(\vec{v})$ is true. Then it is standard that $M \models \phi$ exactly if Eloise has a winning strategy in this game, where a strategy is a function from sequences of legal positions to moves.

These games have *perfect information*; both players know everything that has happened, and in particular when one player makes a choice, they know the other player's previous choices. Game semantics for the Henkin quantifiers, following [8], use games of imperfect information: in the game for $\begin{smallmatrix}\forall x & \exists y \\ \forall u & \exists v\end{smallmatrix}\phi$, when Eloise chooses for v, she does not know what Abelard chose for x. To make this explicit, the logic is written with a more general syntax which is linear rather than two dimensional. A full account of the appropriate logic requires several new constructs, some of which raise subtle issues [9]; we shall work with a restricted version which is sufficient to express all Henkin quantifiers.

In addition to the usual first-order syntax, we also have *independent quantification*: If ϕ is a formula, x a variable, and W a finite set of variables, then $\forall x/W . \phi$ and $\exists x/W . \phi$ are formulae. The intention is that W is the set of independent variables, whose values the player is not allowed to know at this choice point: thus the Henkin quantifier $\begin{smallmatrix}\forall x & \exists y \\ \forall u & \exists v\end{smallmatrix}$ can be written as $\forall x/\varnothing . \exists y/\varnothing . \forall u/\{x, y\} . \exists v/\{x, y\}$. If one then plays the usual model-checking game with this additional condition, which can be formalized by requiring strategies to be uniform in the 'unknown' variables, one gets a game semantics which characterizes the Skolem function semantics in the sense that Eloise has a winning

strategy iff the formula is true.

This logic is called by Hintikka 'independence-friendly' logic. Study of this particular formalism has been mostly carried out by Hintikka and colleagues; but there has been over the last thirty years a continued interest in branching quantification in natural language semantics, increased now by the current popularity of 'Game Theoretical Semantics'. (The recent thesis [12] contains a most useful account of this area.) However, there has been little interest in the computer science temporal logic community.

3 Independence-friendly modal logic

One reason for this is that at first sight, independence-friendly modal logic makes little sense. Suppose that we extend the usual syntax of modal logic with the Hintikka slash; we will also need to assign a tag to each modality, so that we can refer back to it after a slash.

Definition 3.1 *The syntax of independence-friendly modal logic (IFML) is given as follows. Let* $\alpha, \beta, \ldots$ *range over a countable set of* tags, $a, b, \ldots$ *over a set of* labels. tt *and* ff *are IFML formulae. If* Φ_1 *and* Φ_2 *are IFML formulae, so are* $\Phi_1 \vee \Phi_2$ *and* $\Phi_1 \wedge \Phi_2$*; and so are* $\langle a \rangle_{\alpha/\beta_1,\ldots,\beta_m} \Phi_1$ *and* $[a]_{\alpha/\beta_1,\ldots,\beta_m} \Phi_1$.

Certain syntactic conditions may be imposed:

Definition 3.2 *An IFML formula* Φ *is* well-formed *if*

(a) *in every subformula* $\langle a \rangle_{\alpha/\beta_1,\ldots,\beta_m} \Psi$*, the* bound tag α *is uniquely bound in* Φ*;*

(b) *every* independent tag β_i of α *is bound in some higher modality in* Φ*. It is moreover* good *if*

(c) *the* dependency relation *on tags given by* $\alpha \prec \beta$ *if* β *is not an independent tag of* α*, is transitive.*

We will for this paper restrict ourselves to good formulae.

Of the well-formedness requirements, (a) is a convenience to avoid renaming, but (b) is more controversial: it implies, for example, that a subformula of a well-formed formula is not in general well-formed. This is an issue related to questions of compositional semantics; see [9] for a discussion.

The 'goodness' requirement is a restriction largely for technical convenience. If the dependency relation is not transitive, one can have a phenomenon called 'signalling' [9], whereby intendedly independent choices can be made dependent. Although this is interesting in certain linguistic applications, in 'normal' mathematics, and arguably in logics for concurrency, it is undesirable.

Obviously, the intended semantics of an independence-friendly modal logic is that the existential choice in the $\langle a \rangle_{\alpha/\beta_1,\ldots,\beta_m}$ must be made independently

of the choices made in the modalities tagged by β_i. However, in a standard transition system semantics for modal logic, the choices available at a modality are determined by the choices made in earlier modalities, and thus in general it makes no sense to ask for an independent choice.

This problem is removed if the events referred to in the modalities are 'independent' in some sense. For example, in a system comprising two parallel, non-communicating, components, two independent modalities can reasonably refer to choices made in different components. Moreover, the two independent local choices may result in only a single action at a global system level, as when in CCS two actions synchronize; it is this situation that gives the new expressive power in the ATL of [1], and in the 'Henkin modal logic' of [3]. This observation then naturally raises the question of the relationship between independence in the meaning of Hintikka, and independence in semantic models for concurrency.

To examine this question, we shall revert from models with independence implicitly given by locality, to a model with explicit independence. Of the many possibilities, let us choose *transition systems with independence*; these are perhaps the nearest model to ordinary labelled transition systems, and have been used by Nielsen and others to study branching-time logics of (concurrent) independence.

First, we banish a confusing clash of terminology. In 'transition systems with independence', the independence is concurrency, in the model; we wish to relate this to Hintikka-style logical 'independence'. Therefore, *henceforth, concurrent model independence will be called 'concurrency'; 'independence' will be used only to refer to logical independence.* We stress that 'concurrency' is here being used as an *ad hoc* term to distinguish model independence from logic independence. In the literature, 'concurrency' is a distinct concept from model 'independence'; because we will make restrictions on our classes of models, the distinction does not occur in our setting. (We welcome suggestions for better terminology.)

Definition 3.3 *A* coherent transition system with concurrency *(TSC) is a labelled transition system with states S, labels L, and transition relation $\rightarrow \subseteq S \times L \times S$, together with a relation $\mathcal{C} \subseteq \rightarrow \times \rightarrow$ and an initial state s_0. Two transitions $t_1 = (s_1 \xrightarrow{a_1} s_1')$ and $t_2 = (s_2 \xrightarrow{a_2} s_2')$ are* concurrent *if $(t_1, t_2) \in \mathcal{C}$. A relation $\prec$ between transitions with the same label is defined by*

$$s_1 \xrightarrow{a} s_1' \prec s_2 \xrightarrow{a} s_2' \Leftrightarrow \exists b.\, (s_1' \xrightarrow{b} s_2')\ \mathcal{C}\ (s_1 \xrightarrow{a} s_1')\ \mathcal{C}\ (s_1 \xrightarrow{b} s_2)\ \mathcal{C}\ (s_2 \xrightarrow{a} s_2')$$

(i.e., the two a transitions form a diamond with two b transitions independent of a; notionally, the two a transitions are the same a 'event', and the two b transitions are the same b 'event'); $\sim$ is the reflexive, symmetric and transitive closure of $\prec$, and it groups transitions into events. *In addition, the relation*

$\mathcal{C}$ is required to satisfy four natural axioms which ensure that an event has a unique outcome at a given state, that concurrent transitions may occur in either order, that concurrency respects events, and that two concurrent events can occur one after the other:

1. $s \xrightarrow{a} s_1 \sim s \xrightarrow{a} s_2 \Rightarrow s_1 = s_2$
2. $s \xrightarrow{a} s_1 \ \mathcal{C} \ s_1 \xrightarrow{b} u \Rightarrow \exists s_2.\, s \xrightarrow{a} s_1 \ \mathcal{C} \ s \xrightarrow{b} s_2 \ \mathcal{C} \ s_2 \xrightarrow{a} u$
3. $s \xrightarrow{a} s_1 \prec s_2 \xrightarrow{a} u \ \mathcal{C} \ w \xrightarrow{b} w' \Rightarrow s \xrightarrow{a} s_1 \ \mathcal{C} \ w \xrightarrow{b} w'$

   and $w \xrightarrow{b} w' \ \mathcal{C} \ s \xrightarrow{a} s_1 \prec s_2 \xrightarrow{a} u \Rightarrow w \xrightarrow{b} w' \ \mathcal{C} \ s_2 \xrightarrow{a} u$
4. $s \xrightarrow{a} s_1 \ \mathcal{C} \ s \xrightarrow{b} s_2 \Rightarrow \exists u.\, s_1 \xrightarrow{b} u \ \mathcal{C} \ s \xrightarrow{a} s_1$

(a plain TSC need not satisfy axiom 4, the coherence axiom; however, most reasonable models and classes of models are coherent, and we need it for Theorem 6.10, so we adopt it as a standard requirement). Consequently, a firing sequence of transitions gives rise to a partial order of events, which can be linearized into several different transition sequences, in the usual way of partial order semantics. (Note: in the literature, I is used rather than $\mathcal{C}$, as TSCs are called TSIs.)

In graphical depictions of TSCs, concurrent transitions are denoted by putting the symbol $\mathcal{C}$ inside the commutative square, and the initial state is marked by a circle (when it is not obvious).

We can now define a semantics for IFML, given *à la* Hintikka, by defining its model-checking game as a game of imperfect information. A consequence of this is that the semantics is not defined on states, but requires some history to be kept.

Definition 3.4 *A* tagged run *of a TSC is a sequence $s_0 \xrightarrow[\alpha_0]{a_0} \ldots \xrightarrow[\alpha_{n-1}]{a_{n-1}} s_n$, where the α_i are distinct* tags*; we shall also use the tag α_i to refer to the transition $s_i \xrightarrow{a_i} s_{i+1}$. We let ρ, σ etc. range over tagged runs, and use obvious notations for extensions of runs.*

A position *of the model-checking game for an IFML formula Φ on a TSC is a pair of a tagged run and a subformula, written $\rho \vdash \Psi$.*

The initial position is $s_0 \vdash \Phi$.

The rules of the game are as follows:

- *At a position $\rho \vdash \mathrm{tt}$, Eloise wins; at $\rho \vdash \mathrm{ff}$, Abelard wins.*
- *At $\rho \vdash \Phi_1 \vee \Phi_2$ (resp. $\vdash \Phi_1 \wedge \Phi_2$), Eloise (resp. Abelard) chooses a new position $\rho \vdash \Phi_i$.*
- *At $\rho = s_0 \xrightarrow{a_0} \ldots \xrightarrow{a_{n-1}} s_n \vdash \langle b \rangle_{\beta/\alpha_{i_1},\alpha_{i_2},\ldots,\alpha_{i_m}} \Psi$ (resp. $\vdash [b]_{\beta/\alpha_{i_1},\alpha_{i_2},\ldots,\alpha_{i_m}} \Psi$), Eloise (resp. Abelard) chooses a transition $s_n \xrightarrow{b} t$ that is concurrent with all the transitions α_{i_j}, and the new position is $\rho \xrightarrow[\beta]{b} t \vdash \Psi$.*

Tags are, of course, merely syntactic sugar; it suffices to identify the ith transition by i. However, tags are convenient to match the definition of IFML.

As usual, a strategy for Eloise is a function from her positions to choices. Imperfect information games are handled by imposing additional conditions on strategies.

Definition 3.5 *An Eloise strategy σ is* uniform *if the choice at a $\langle\rangle$ position is uniform in the specified independent earlier choices, in the following sense:*

Let $\rho \vdash \langle b\rangle_{\beta/\alpha_{i_1},\alpha_{i_2},\ldots,\alpha_{i_m}} \Psi$ be as above. The strategy σ must choose $s_n \xrightarrow{b} t$ such that if $s_0 = s'_0 \xrightarrow{a_0} \ldots \xrightarrow{a_{n-1}} s'_n \vdash \langle b\rangle_{\beta/\alpha_{i_1},\alpha_{i_2},\ldots,\alpha_{i_m}} \Psi$ is any other position such that $j \notin \{i_1,\ldots,i_m\} \Rightarrow \alpha_j \sim \alpha'_j$, σ chooses a transition $s'_n \xrightarrow{b} t' \sim s_n \xrightarrow{b} t$. (In words, σ must choose the same event regardless of the events chosen in the independent modalities. If no such event can be chosen, there is no uniform strategy.) Abelard uniform strategies are defined similarly.

Definition 3.6 *An IFML formula Φ is* true *in a given TSC, written $s_0 \models \Phi$, iff Eloise has a uniform winning strategy for the model-checking game $s_0 \vdash \Phi$.*

Φ is false *iff Abelard has a uniform winning strategy.*

Φ is determined *iff it is either false or true.*

The non-determinacy in general of the model-checking game is a characteristic feature of independence-friendly logic. For a simple example, consider the TSC generated by the CCS process $((a.c + a.\overline{c}) \mid (b.c + b.\overline{c}))\backslash c$ (in which the a transitions are independent of the b transitions), and the formula $[a]_\alpha\langle b\rangle_{\beta/\alpha}\langle\tau\rangle\mathrm{tt}$. This formula is not true, since Eloise cannot choose a b transition so as to synchronize unless she knows which a transition was chosen; but it is also not false, since Abelard has no strategy for falsifying it. For practical purposes, we may consider untruth to be falsehood.

4 IFML equivalence

One of the first questions about any logic is, what is the induced equivalence? In the case of IFML (or indeed the simpler Henkin modal logic of [3]), the definition of equivalence itself is problematic, because of the non-determinacy. We take the weaker (practical) definition, and say

Definition 4.1 *Two TSCs S and T are IFML-equivalent, $S \sim_{\mathrm{IFML}} T$, if for every IFML formula Φ, $S \models \Phi \Leftrightarrow T \models \Phi$.*

Logically induced equivalences are typically characterized by a game naturally related to the satisfaction game: for modal logic, we have bisimulation games and model-checking games, for first-order logic we have Ehrenfeucht–Fraïssé games and Hintikka games. For IF logics, the outscoping nature of the / makes such a formulation harder, and to our knowledge none has been

presented. We will consider E–F games for independence logics in a later article; here we study IFML equivalence by relation to known equivalences in true concurrency.

5 Restrictions on models

For the remainder of this paper, we will consider restricted classes of models. Analysing the effect of removing the restrictions is left to later work.

Firstly, *all TSCs will be image-finite*: that is, for any state s and label a, there are only finitely many a-successors of s. This is a standard restriction required to obtain an exact match between finitary modal logic and bisimulation.

Secondly, *all TSCs will be acyclic*: that is, no state is reachable from itself. This restriction avoids the necessity of distinguishing between models and their unfoldings, which in turn avoids the necessity to distinguish multiple occurrences of the 'same' event.

6 Equivalences for concurrency

There are numerous equivalences for concurrency, but there is one spectrum of particularly natural equivalences that appears promising: the spectrum from bisimulation through to coherent hereditary history preserving bisimulation. These equivalences have several characterizations; we will define them in the style of classical bisimulation, and also give the game characterizations, which will be useful in our results.

The weakest equivalence is ordinary 'strong bisimulation'; this is well known to be too weak for true concurrent properties, but we define it just to help clarify the other definitions. In particular, we will define it on runs, rather than states.

Definition 6.1 *A relation R on pairs of runs of two TSCs S and T is a* (strong) bisimulation *if*

A $(s_0, t_0) \in R$

B *if $(\sigma, \tau) \in R$ and $\sigma' = \sigma \xrightarrow{a} s$ is a run, then there is t such that $\tau' = \tau \xrightarrow{a} t$ and $(\sigma', \tau') \in R$; and symmetrically.*

Systems S and T are (strongly) bisimilar*), $S \sim_{\mathrm{b}} T$, if there is a strong bisimulation between them.*

Bisimulation makes no use of the history of a run, and ignores the concurrency, and thus is definable on states of the TSCs, as is usually done. The definition can also be cast in game-theoretic terms:

Definition 6.2 *The* bisimulation game *played between Duplicator and Spoiler*

on two TSCs S and T is played as follows. Positions are pairs (σ, τ) of runs from S and T. The initial position is (s_0, t_0). The two players alternate, with Spoiler starting. The rules are:

I *Spoiler chooses one of S or T, say S, and chooses a transition $s_n \xrightarrow{a_n} s_{n+1}$. Duplicator must respond in the other system with a transition $t_n \xrightarrow{a_n} t_{n+1}$ extending τ, or else she loses.*

II *If either player cannot move, the other wins; if play continues for ever, Duplicator wins.*

S and T are bisimilar iff Duplicator has a winning strategy for the bisimulation game iff Duplicator has a history-free winning strategy.

Since modal logic characterizes bisimulation, and IFML includes modal logic, it is immediate that $\sim_{\mathrm{IFML}}$ implies $\sim_{\mathrm{b}}$.

A stronger notion of equivalence is obtained [7,13] by requiring the equivalence to preserve the concurrency relation between matching events. The following formulation is not the original definition, but is equivalent in our framework:

Definition 6.3 *R is a* history-preserving bisimulation (hpb) *if*

A *$(s_0, t_0) \in R$*

C *if $(\sigma, \tau) \in R$ and $\sigma' = \sigma \xrightarrow{a} s$ is a run, then there is t such that $\tau' = \tau \xrightarrow{a} t$, and transitions i and j in σ' are concurrent iff transitions i and j in τ' are concurrent, and $(\sigma', \tau') \in R$; and symmetrically.*

and we write $S \sim_{\mathrm{hpb}} T$ if there is an hbp between S and T.

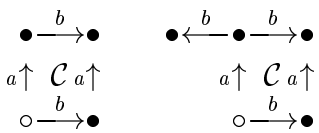
and there is the obvious analogous game characterization.

Hpb detects at least some true concurrent features; for example, it distinguishes $a.b + b.a$ from $a|b$. However, it has been argued [6,5] that hpb and similar relations such as local/global cause equivalence are really about causality, not about concurrency, and that true concurrency is more correctly captured by the stronger equivalences. The development in this paper will provide further backing to such a view.

The first, initially discouraging, result is that hpb can make distinctions that IFML cannot.

Theorem 6.4 $\sim_{\mathrm{IFML}} \not\subseteq \sim_{\mathrm{hpb}}$

Proof. Consider the following systems:



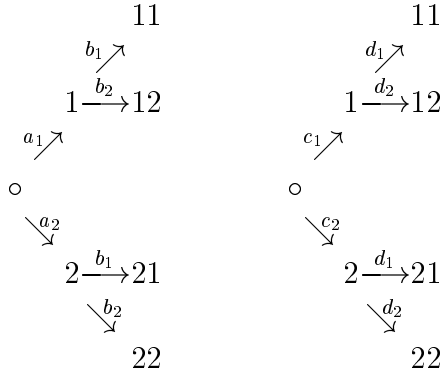
These systems are not hpb, but it may be verified by exhaustive checking that no IFML formula distinguishes them. □

This example will suggest later a possible modification to the definition of IFML; for the present, we continue with the investigation.

It would be surprising if hpb were finer than IFML-equivalence, and indeed it is not, although this is not quite so easy to demonstrate.

Theorem 6.5 $\sim_{\mathrm{hpb}} \not\subseteq \sim_{\mathrm{IFML}}$

Proof. The simplest counter-example we have at present is rather complex to draw in full, so we will give a combined graphical and syntactic description. Let A and C be the two systems



and let P be their concurrent composition, which is a pyramid with 16 distinct final states on the square face. The systems S and T are formed by adding an e transition to some of these final states, as indicated by the following matrix in which the columns are the A states $11, 12, 21, 22$, the rows are the C states $11, 12, 21, 22$, and the entries indicate the presence of an e transition in the given systems.

$$\begin{array}{cccc} - & ST & - & S \\ ST & - & S & T \\ S & T & ST & - \\ - & S & - & ST \end{array}$$

It may be verified (and has been checked with the Edinburgh Concurrency Workbench!) that S and T are strongly bisimilar, and since the concurrency relations are the same, they are also history-preserving bisimilar. However, the following IFML formula is true of S but not of T:

$$[a]_{\alpha}\langle b\rangle_{\beta}[c]_{\gamma/\alpha\beta}\langle d\rangle_{\delta/\alpha\beta}\langle e\rangle \mathrm{tt}.$$

(This is because in S, Eloise can choose b_1 after Abelard's a_1 and b_1 after Abelard's a_2; then she can choose d_2 after c_1 and d_1 after c_2, without depending on a, and she ends up in a state with an e transition. In T, on the other hand, no such uniform choice of d exists.) □

A stronger equivalence from concurrency theory is *hereditary (or strong) history-preserving bisimulation (hhpb)* [2,10]. Its relational characterization is

Definition 6.6 *R is a* hereditary history-preserving bisimulation (hhpb) *if*

A $(s_0, t_0) \in R$

B *if $(\sigma, \tau) \in R$ and $\sigma' = \sigma \xrightarrow{a} s$ is a run, then there is t such that $\tau' = \tau \xrightarrow{a} t$ and $(\sigma', \tau') \in R$; and symmetrically;*

D *if $(\sigma = s_0 \xrightarrow[\alpha_0]{a_0} \ldots s_n, \tau = t_0 \xrightarrow[\beta_0]{a_0} \ldots t_n) \in R$, and transition α_i is* backwards enabled *in σ, meaning that α_i is concurrent with every later α_j, then β_i is backwards enabled in τ and $(\sigma', \tau') \in R$, where σ' is obtained from σ by using the TSC diamond axioms to push α_i to the end, and then deleting α_i, and similarly τ' is obtained from τ by likewise 'backtracking' β_i; and symmetrically.*

The rather complex looking clause D is nothing more than undoing the latest action in some concurrent component; viewing a run as a partial order, rather than a sequence, it is simply the deletion of a maximal element.

It is easy to see that clauses B and D imply that hhpb also satisfies clause C of the hpb definition, and so hhpb is finer (and indeed strictly finer) than hpb. The natural game characterization [11] of hhpb is

Definition 6.7 *The* hhpb *game played between Duplicator and Spoiler on two TSCs S and T is played as follows. Positions are pairs (σ, τ) of runs from S and T. The initial position is (s_0, t_0). The two players alternate, with Spoiler starting. Spoiler may move in two ways, to which Duplicator must respond.*

(i) Spoiler chooses one of S or T, say S, and chooses a transition $s_n \xrightarrow{a_n} s_{n+1}$. Duplicator must respond in the other system with a transition $t_n \xrightarrow{a_n} t_{n+1}$ extending τ, or else she loses.

(ii) Alternatively, Spoiler chooses S or T (say S), and a transition $s_i \xrightarrow{a_i} s_{i+1}$ in σ which is backward-enabled. He then 'backtracks' along this transition, as in the relational definition. Duplicator must then respond by backtracking the ith transition in the other system; if this transition is not backwards enabled, she cannot move.

(iii) If either player cannot move, the other wins; if play continues for ever, Duplicator wins.

Hhpb looks like a good candidate for comparison with IFML. For the same reasons as hpb, hhpb can distinguish systems that IFML cannot; but one might wonder whether hhpb is finer than IFML-equivalence (for our restricted models). We have a counter-example for infinite-branching models, but for image-finite models we have not so far constructed a counter-example (or proved the assertion). We make the

Conjecture 6.8 $\sim_{\text{hhpb}} \not\subseteq \sim_{\text{IFML}}$

(As an illustration of how hhpb is stronger than hpb, and how it is intuitively related to IFML, note that the two systems of Theorem 6.5 are distinguished by the formula

$$[a]\langle b\rangle[c]\langle d\rangle \textcircled{b}\textcircled{@}[a]\langle b\rangle\langle e\rangle \mathrm{tt}$$

of the characteristic logic [11] for hhpb (where @ is the modality of backtracking an a action). We shall discuss in a later article the nature of the relationship between this formula and the IFML formula.)

Fortunately, the concurrency literature contains a yet stronger equivalence than hhpb, which is even more naturally related to IFML, and is easily shown to be finer than IFML. This is the equivalence called *strong coherent history-preserving bisimulation* in [4]; we shall call it *coherent hereditary history-preserving bisimulation (chhpb).*

Chhpb is most simply defined in a partial order setting:

Definition 6.9 *A hpb R is* coherent hereditary *if, for any pair of partial order runs (σ,τ) with $\sigma = \bigcup_{i\in I}\sigma_i$, $(\sigma,\tau)\in R$ iff $\tau = \bigcup_{i\in I}\tau_i$ for some τ_i, and every $(\sigma_i,\tau_i)\in R$.*

In other words, in a chhpb, any partial order can be matched by putting together the matchings for its 'concurrent components'; and in particular, the matching of a given event can be chosen independently of any concurrent events. This is exactly the property required to prove easily:

Theorem 6.10 *If S and T are chhpb, then they are IFML-equivalent.*

Proof. (Sketch) Let $S \sim_{\mathrm{chhpb}} T$ be two TSCs, and let Φ be an IFML formula such that $S \models \Phi$. We shall use the chhpb relation and Eloise's winning uniform strategy for $S \vdash \Phi$ to allow her to win $T \vdash \Phi$.

Suppose that in the model-checking games we have reached positions $\sigma \vdash \Psi$ and $\tau \vdash \Psi$. If it is Abelard's turn to move in T, Eloise copies his move to S using the chhpb. If it is Eloise's turn to move, her move in T is given by taking her move in S and mapping it to T via the chhpb. Since the chhpb maps a given S event to a T event that depends only on the causal predecessors, if Eloise's choice is uniform in S, it is also uniform in T. □

Chhpb can be defined in terms of (linear) runs thus:

Definition 6.11 (Alternative to Defn 6.9) *R is a* coherent hereditary history-preserving bisimulation (chhpb) *if*

The clauses of hhpb, together with

E *if $(\sigma \xrightarrow[\alpha]{a} s, \tau \xrightarrow[\beta]{a} t)\in R$ and $(\sigma \xrightarrow[\alpha']{a'} s', \tau \xrightarrow[\beta']{a'} t')\in R$ and $\alpha\ \mathcal{C}\ \beta$, then $\alpha'\ \mathcal{C}\ \beta'$ and $(\sigma \xrightarrow{a} s \xrightarrow{a'} s'', \tau \xrightarrow{a} t \xrightarrow{a'} t'')\in R$.*

Lemma 6.12 *Definitions 6.9 and 6.11 are equivalent.* □

Using this definition, a more direct proof of the preceding theorem can be given without invoking explicitly the partial order structure.

7 Alternatives to IFML?

The fact that all the concurrent equivalences (apart from bisimulation itself) distinguish systems that IFML does not, is unsatisfactory. Upon inspection of the counter-example of Theorem 6.5, one can see that this is due to a rather simple mismatch between the expressivity of the concurrent logics and IFML: the concurrent logics can express 'a followed by a concurrent b', 'a followed by a dependent b', and also 'a followed by choice of concurrent and dependent b'. IFML, on the other hand, can express 'a followed by a concurrent b', and 'a followed by a dependent b and no concurrent b', but cannot distinguish the case where there is a dependent b as well as a concurrent b.

It is possible to make a small change to the semantics of IFML which addresses this issue. Let us call the result IFMLd (IFML with explicit dependence), defined by the following change to the model-checking game of Defn 3.4:

Definition 7.1 *The IFMLd game is as for IFML except that:*

- *At* $\rho = s_0 \xrightarrow{a_0} \ldots \xrightarrow{a_{n-1}} s_n \vdash \langle b \rangle_{\beta/\alpha_{i_1},\alpha_{i_2},\ldots,\alpha_{i_m}} \Psi$ *(resp.* $\vdash [b]_{\beta/\alpha_{i_1},\alpha_{i_2},\ldots,\alpha_{i_m}} \Psi$*), Eloise (resp. Abelard) chooses a transition* $s_n \xrightarrow{b} t$ *that is concurrent with all the transitions* α_{i_j} and not concurrent with any other transition α_k, *and the new position is* $\rho \xrightarrow[\beta]{b} t \vdash \Psi$.

That is, choices in modalities are required to be concurrent with previous choices if and only if they are logically independent, rather than just if.

For this modification to make sense, we must require the dependency relations in the models to be transitive; this is formally, but not actually, a further restriction, since events that are formally concurrent but actually causally dependent can be made formally non-concurrent without change to the model.

This is superficially attractive, and certainly deals with the example of Theorem 6.5, and we

Conjecture 7.2 $\sim_{\mathrm{IFMLd}} \subseteq \sim_{\mathrm{hpb}}$

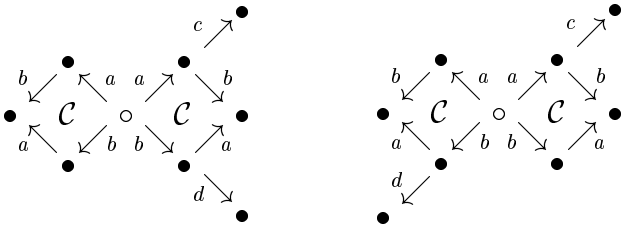
but have not established this conjecture.

It is also very tempting to conjecture that $\sim_{\mathrm{IFMLd}} \subseteq \sim_{\mathrm{hhpb}}$ and even to make the highly desirable conjecture $\sim_{\mathrm{IFMLd}} = \sim_{\mathrm{chhpb}}$ which would give the first logical characterization of the coherent equivalence. Unfortunately, these conjectures fail.

Theorem 7.3 $\sim_{\mathrm{IFMLd}} \not\subseteq \sim_{\mathrm{hhpb}}$

Proof. The following is a notorious example [11] of two systems that are not

hhbp (although they are hbp):



It may be verified by exhaustive (and in this case somewhat exhausting) checking that neither IFML nor IFMLd can distinguish them.

It should, however, be pointed out that despite the naturalness of IFMLd, there are some unpleasant consequences of adopting it. In particular, it becomes impossible to express the ordinary modal logic formula $[a]\langle b\rangle\Phi$, where the choice of b *may* depend on a, if a and b happen to be concurrent. (It is for this reason that Conjecture 7.2 is not the simple result one would like.)

8 Conclusion

We have shown that it is possible to define a modal version of the Hintikka–Sandu independence-friendly logic, and that such a logic naturally requires true concurrent models. We have looked at the relationship between the induced equivalence and the equivalences associated with true concurrent models. The results so far indicate that although there is a natural connection, it is not as clean as one would like; however, we are hopeful that further work will throw more light on this. We expect in the full version of this paper to settle all the issues explicitly labelled as conjectures; but we think it will take a more substantial effort to complete the analysis. There are intriguing questions about the exact relationship between backtracking (as used in hhbp), and uniformity (as used in chhpb and in IFML), and we suspect that these questions may provide a useful notion of Ehrenfeucht–Fraïssé game for independence logics. (To coin a slogan, the art of independence is in doing second-order things without appearing to do so.) In turn, independence logics may give new insight into the complexity of the concurrent equivalences.

9 Acknowledgements

The first author is supported by EPSRC Advanced Research Fellowship AF/100690; the second author is supported by EPSRC Research Grant GR/M84763 'Dimensions of Concurrency'. We thank the referees for helpful comments.

References

[1] R. Alur, T. Henzinger and O. Kupferman, Alternating-time temporal logic, in *Proc. 38th FOCS (1997)*, 100–109.

[2] M. A. Bednarczyk, Hereditary history preserving bisimulations, manuscript (1991).

[3] J. C. Bradfield, Independence: logics and concurrency, *Proc. CSL 2000*, LNCS **1862** 247–261 (2000).

[4] A. Cheng, *Reasoning About Concurrent Computational Systems*, Ph.D. thesis, BRICS DS–96–2, Univ. Aarhus (1996).

[5] S. Fröschle, *Decidability and coincidence of equivalences for concurrency.* Ph.D. thesis, Univ Edinburgh, forthcoming.

[6] S. Fröschle and T. Hildebrandt, On plain and hereditary history-preserving bisimulation. *Proc. MFCS '99*, LNCS **1672**, 354–365 (1999).

[7] R. van Glabbeek and U. Goltz, Equivalence notions for concurrent systems and refinement of actions, *Proc. MFCS '89*, LNCS **379** (1989).

[8] J. Hintikka and G. Sandu, A revolution in logic?, *Nordic J. Philos. Logic* **1**(2) 169–183 (1996).

[9] W. Hodges, Compositional semantics for a language of imperfect information, *Int. J. IGPL* **5**(4), 539–563.

[10] A. Joyal, M. Nielsen and G. Winskel, Bisimulation from open maps, *Inform. and Comput.*, **127**(2) 164–185 (1996).

[11] M. Nielsen and C. Clausen, Bisimulations, games and logic, in Karhumaki, Maurer and Rozenberg (eds), *Results and Trends in Theoretical Computer Science: Colloquium in Honour of Arto Salomaa*, LNCS **812** 289-305 (1994). Also as BRICS report RS-94-6.

[12] A. Pietarinen, *Games logic plays. Informational independence in game-theoretic semantics.* D.Phil. thesis, Univ Sussex (2000).

[13] A. Rabinovitch and B. Trakhtenbrot, Behaviour structures and nets. *Fund. Inf.* **11**(4) (1988).

Rewrite Systems with Constraints

Jan Strejček [1,2]

Faculty of Informatics, Masaryk University
Botanická 68a, 60200 Brno, Czech Republic

Abstract

We extend a widely used concept of rewrite systems with a unit holding a kind of global information which can influence and can be influenced by rewriting. The unit is similar to the *store* used in concurrent constraint programming, and can be also seen as a special (weak) state unit. We present how this extension changes the expressive power of rewrite systems classes which are included in Mayr's PRS hierarchy [8]. The new classes (fcBPA, fcBPP, fcPA, fcPAD, fcPAN, fcPRS) are described and inserted into the hierarchy.

1 Introduction

The cornerstone of concurrency theory is the notion of labelled transition system. Caucal [4] presents an elegant classification of transition systems using families of sequential rewrite systems related to the Chomsky hierarchy. Caucal's classification has been generalised by Moller [11] to both parallel and sequential rewrite systems. Moller's approach was further generalised by Mayr [8], who defines the dynamics for rewrite systems using sequential and parallel composition together. The resulting model is called *process rewrite systems (PRS)*.

Concurrent constraint programming (CCP) [14] is one of the most successful applications of the ideas of concurrency and computing with partial information. In CCP processes work concurrently with a shared *store*, which is seen as a constraint on the values that variables can represent. In any state of the computation, the store is given by the constraint established until that moment. CCP provides two operations to deal with the store, *tell* and *ask*. The *tell* monotonically updates the store by adding a constraint (provided the store remains *consistent*). The *ask* is a test on the store – it can be executed only if the current store is strong enough to *entail* a specified constraint. If this

[1] This work has been partially supported by the Grant Agency of Czech Republic, grant No. 201/00/0400.

[2] Email: xstrejc@fi.muni.cz

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

is not the case, then the process suspends (waiting for the store to accumulate more information by contributions of the other processes).

We transfer some principles of CCP to process rewrite systems. Previously, we have introduced an analogous modification of purely sequential and purely parallel rewrite systems in [15]. In both cases, the aim is to characterise the changes of expressive power of these systems. The mechanism of PRS is extended with the store, which can contain some global (monotonically evolving) information. We add two constraints to every rewrite rule. A rule can be applied only if the actual store is strong enough to entail the first constraint; the second constraint is added to the store when the extended rule is used (the rule is applicable if the store is kept consistent). Extended process rewrite systems are called *process rewrite systems with finite constraint systems (fcPRS).* [3]

We obtain some interesting results by studying which labelled transition systems (up to bisimulation) can be denoted by specific classes of PRS systems (accordant with well-known formalisms like finite state systems, basic process algebra (BPA), basic parallel processes (BPP), process algebra (PA), pushdown processes, Petri nets, etc.) and by corresponding fcPRS classes. The expressive power of finite state systems, pushdown processes, and Petri nets keeps unchanged by adding the store. This does not hold in the case of BPA, BPP, PA, PAD, and PAN class when the expressive power strictly increases, thus some new classes are obtained in this way. Hence this new framework can be used to solve some interesting open problems, e.g. to examine the decidability border within the process hierarchies already maintained (in case of bisimilarity it is known that the border line goes between BPP and its "state-extended" version MSA continuing between (normed) PA and its "state-extended" version etc.); our new process classes are situated in this grey area.

2 Basic definitions

In this section we recall the notions of labelled transitions systems, language generated by such system, and bisimulation equivalence.

Definition 2.1 A *labelled transition system (LTS)* $\mathcal{L}$ is a tuple $(S, Act, \longrightarrow, \alpha_0)$, where S is a set of *states* or *processes*, Act is a set of *atomic actions* or *labels*, $\longrightarrow \subseteq S \times Act \times S$ is a *transition relation* (written $\alpha \stackrel{a}{\longrightarrow} \beta$ instead of $(\alpha, a, \beta) \in \longrightarrow$), $\alpha_0 \in S$ is a distinguished *initial state*. A state $\alpha \in S$ is *terminal* (or *deadlocked*, written $\alpha \not\longrightarrow$) if there is no $a \in Act$ and $\beta \in S$ such that $\alpha \stackrel{a}{\longrightarrow} \beta$.

The transition relation $\longrightarrow$ can be homomorphically extended to finite

[3] Note that rules of fcPRS systems can be also seen as a new special "format" of SOS rules (in the sense of [6]) with side conditions referring to a global (monotonic) store. However this viewpoint is not examined in this paper.

sequences of actions $\sigma \in Act^*$ so as to write $\alpha \xrightarrow{\varepsilon} \alpha$ and $\alpha \xrightarrow{a\sigma} \beta$ whenever $\alpha \xrightarrow{a} \gamma \xrightarrow{\sigma} \beta$ for some state γ. The set of states α such that $\alpha_0 \xrightarrow{\sigma} \alpha$ for the initial state α_0 and some $\sigma \in Act^*$ is called the set of *reachable* states.

Definition 2.2 The *language* generated by the labelled transition system $\mathcal{L}$ is the set $L(\mathcal{L}) = L(\alpha_0)$, where $L(\alpha) = \{w \in Act^* \mid \exists \beta : \alpha \xrightarrow{w} \beta \not\rightarrow\}$. States α and β of the system $\mathcal{L}$ are *language equivalent*, written $\alpha \sim_L \beta$, iff they generate the same language, i.e. $L(\alpha) = L(\beta)$.

Language equivalence is generally taken to be too coarse in the framework of concurrency theory. The second presented equivalence, *bisimulation equivalence*, is perhaps the finest behavioural equivalence studied. Bisimulation equivalence was defined by Park [13] and used by Milner [9,10] in his work on CCS.

Definition 2.3 A binary relation $\mathcal{R}$ on states of labelled transition system is a *bisimulation* iff whenever $(\alpha, \beta) \in \mathcal{R}$ we have that

- if $\alpha \xrightarrow{a} \alpha'$ then $\beta \xrightarrow{a} \beta'$ for some β' with $(\alpha', \beta') \in \mathcal{R}$,
- if $\beta \xrightarrow{a} \beta'$ then $\alpha \xrightarrow{a} \alpha'$ for some α' with $(\alpha', \beta') \in \mathcal{R}$.

α and β are *bisimulation equivalent* or *bisimilar*, written $\alpha \sim \beta$, iff $(\alpha, \beta) \in \mathcal{R}$ for some bisimulation $\mathcal{R}$.

3 Process rewrite systems (PRS)

This section summarise the first part of Mayr's paper titled "Process Rewrite Systems" [8].

The *process rewrite systems (PRS)* developed by Mayr represent a very general term rewriting formalism offering a way for finite description of possibly infinite transition systems. The formalism covers many widely known models like finite-state processes (FS), basic parallel processes (BPP), context-free processes (BPA), pushdown processes (PDA), process algebras (PA), Petri nets (PN), and provides a unified view of these models. The definition of PRS is more general than the definitions of rewrite system given by Caucal [4] (only with sequential composition) and by Moller [11] (only purely sequential and purely parallel rewrite systems).

Let $Const = \{X, Y, Z, \cdots\}$ be a countably infinite set of *process constants*. The set $\mathcal{T}$ of *process terms* is defined by the abstract syntax

$$t = \varepsilon \mid X \mid t_1.t_2 \mid t_1 \| t_2,$$

where ε is the empty term, $X \in Const$ is a process constant (used as an atomic process), "$\|$" means parallel and "." means sequential compositions respectively.

We always work with equivalence classes of terms modulo commutativity and associativity of parallel composition and modulo associativity of sequential composition. Also we define $\varepsilon.t = t = t.\varepsilon$ and $t \| \varepsilon = t$.

The set $Const(t)$ is the set of all constants occurring in a process term t. We distinguish four *classes of process terms.*

"1" Terms consisting of a single process constant like X.
"S" *Sequential* terms - without parallel composition. For example $X.Y.Z$.
"P" *Parallel* terms - without sequential composition. For example $X \| Y \| Z$.
"G" *General* terms with arbitrarily nested sequential and parallel compositions like $(X.(Y \| Z)) \| W$.

We also let $\varepsilon \in S, P, G$, but $\varepsilon \notin 1$.

Definition 3.1 Let $Act = \{a, b, \cdots\}$ be a countably infinite set of *atomic actions*, $\alpha, \beta \in \{1, S, P, G\}$ such that $\alpha \subseteq \beta$. An *(α, β)-PRS (process rewrite system)* Δ is a pair (R, t_0), where

- R is a finite set of *rewrite rules* of the form $t_1 \stackrel{a}{\longrightarrow} t_2$, where $t_1 \in \alpha$, $t_1 \neq \varepsilon$, $t_2 \in \beta$ are process terms and $a \in Act$ is an atomic action,
- $t_0 \in \beta$ is an *initial state.*

A (G, G)-PRS is simply called PRS.

We write $(t_1 \stackrel{a}{\longrightarrow} t_2) \in \Delta$ instead of $(t_1 \stackrel{a}{\longrightarrow} t_2) \in R$, where $\Delta = (R, t_0)$.

For a given Δ we define $Const(\Delta)$ as the set of all constants that occur in rewrite rules or initial state, and $Act(\Delta)$ as the set of all actions that occur in rewrite rules of Δ. The sets $Const(\Delta)$ and $Act(\Delta)$ are both finite.

Each process rewrite system denotes a labelled transition system (LTS) that represents its dynamics. Let $\Delta = (R, t_0)$ be an (α, β)-PRS. The LTS $\mathcal{L}$ denoted by Δ is a tuple $(S, Act(\Delta), \longrightarrow, t_0)$, where $S = \{t \in \beta \mid Const(t) \subseteq Const(\Delta)\}$ is the set of states, t_0 is the initial state and transition relation $\longrightarrow$ is the least relation that satisfies the inference rules [4]

$$\frac{(t_1 \stackrel{a}{\longrightarrow} t_2) \in \Delta}{t_1 \stackrel{a}{\longrightarrow} t_2}, \qquad \frac{t_1 \stackrel{a}{\longrightarrow} t_1'}{t_1 \| t_2 \stackrel{a}{\longrightarrow} t_1' \| t_2}, \qquad \frac{t_1 \stackrel{a}{\longrightarrow} t_1'}{t_1.t_2 \stackrel{a}{\longrightarrow} t_1'.t_2},$$

where $t_1, t_2, t_1' \in \mathcal{T}$.

We speak about "process rewrite system" meaning "labelled transition system generated by process rewrite system".

Obviously, it can be assumed (w.l.o.g.) the initial state t_0 of a (α, β)-PRS is a single constant as there are only finitely many terms t_i such that $t_0 \stackrel{a_i}{\longrightarrow} t_i$.

Figure 1 shows a graphical description of the hierarchy of (α, β)-PRS, simply called *PRS-hierarchy.* Some classes included in the hierarchy correspond to widely known models:

- $(1, 1)$-PRS are equivalent to finite-state systems (FS). Every process constant corresponds to a state and the state space is bounded by $|Const(\Delta)|$. Every finite-state system can be encoded as a $(1, 1)$-PRS.

[4] Note that parallel composition is commutative and, thus, the inference rule for parallel composition also holds with t_1 and t_2 exchanged.

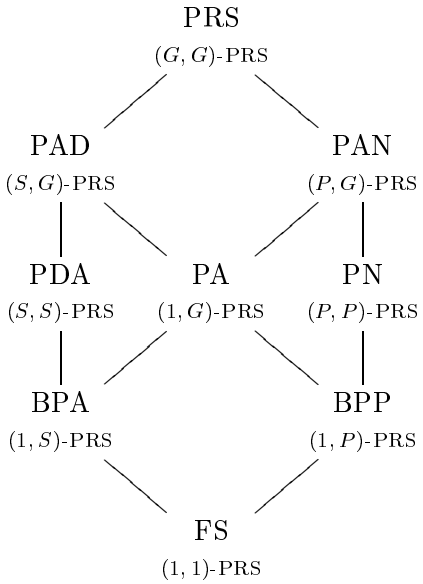


Fig. 1. The PRS-hierarchy

- $(1, S)$-PRS are equivalent to Basic Process Algebra processes (BPA) defined in [1], which are the transition systems associated with Greibach normal form (GNF) context-free grammars in which only left-most derivations are allowed.
- $(1, P)$-PRS are equivalent to communication-free nets, the subclass of Petri nets where every transition has exactly one place in its preset [3]. This class of Petri nets is equivalent to Basic Parallel Processes (BPP) [5].
- $(1, G)$-PRS are equivalent to PA-processes, Process Algebras with sequential and parallel composition, but no communication (see [1] for details).
- It is easy to see that pushdown automata can be encoded as a subclass of (S, S)-PRS (with at most two constants on the left-hand side of rules). Caucal [4] showed that any unrestricted (S, S)-PRS can be presented as a pushdown automaton (PDA), in the sense that the transition systems are isomorphic up to the labelling of states. Thus (S, S)-PRS are equivalent to pushdown processes (which are the processes described by pushdown automata).
- (P, P)-PRS are equivalent to Petri nets (PN). Every constant corresponds to a place in the net and the number of occurrences of a constant in a term corresponds to the number of tokens in this place. This is because we work with classes of terms modulo commutativity of parallel composition. Every rule in Δ corresponds to a transition in the net.
- (S, G)-PRS is the smallest common generalisation of pushdown processes and PA-processes. They are called *PAD* (PA + PDA) in [8].
- (P, G)-PRS are called *PAN-processes* in [7]. It is the smallest common generalisation of Petri nets and PA-processes and it strictly subsumes both of them (e.g., PAN can describe all Chomsky-2 languages while Petri nets cannot).

- The most general case is (G, G)-PRS (here simply called PRS). PRS have been introduced in [8]. They subsume all of the previously mentioned classes.

The hierarchy is not strict w.r.t. language equivalence. For example, both BPA and PDA define exactly the (ε-free) context-free languages. The strictness of the hierarchy w.r.t. bisimulation equivalence follows from previous results [2,11] and the proof, that there is a PDA system (described in Example 3.2) which is not bisimilar to any PAN system and a Petri net (described in Example 3.3) which is not bisimilar to any PAD process.

Example 3.2 Let us consider the following PDA system with initial state $U.X$.

$$\begin{array}{lll} U.X \xrightarrow{a} U.A.X & U.A \xrightarrow{a} U.A.A & U.B \xrightarrow{a} U.A.B \\ U.X \xrightarrow{b} U.B.X & U.A \xrightarrow{b} U.B.A & U.B \xrightarrow{b} U.B.B \\ U.X \xrightarrow{c} V.X & U.A \xrightarrow{c} V.A & U.B \xrightarrow{c} V.B \\ U.X \xrightarrow{d} W.X & U.A \xrightarrow{d} W.A & U.B \xrightarrow{d} W.B \\ V.X \xrightarrow{e} V & V.A \xrightarrow{a} V & V.B \xrightarrow{b} V \\ W.X \xrightarrow{f} W & W.A \xrightarrow{a} W & W.B \xrightarrow{b} W \end{array}$$

Example 3.3 Consider following Petri net given as (P, P)-PRS with initial state $X \| A \| B$.

$$\begin{array}{ll} X \xrightarrow{g} X \| A \| B & Y \| A \xrightarrow{a} Y \\ X \xrightarrow{c} Y & Y \| B \xrightarrow{b} Y \\ X \| A \xrightarrow{d} Z & Y \| A \xrightarrow{d} Z \\ X \| B \xrightarrow{d} Z & Y \| B \xrightarrow{d} Z \end{array}$$

4 PRS with finite constraint systems (fcPRS)

In this section we extend the PRS formalism with a unit (called *store*) able to keep a sort of global information which is accessible to all parallel threads of the term. It is quite surprising that this unit (which is not as powerful as a general finite-state control unit which gives Turing power even to the PA class) increases the expressive power of classes like PAN and PAD.

The state space and possible evolution of the store used by PRS with finite constraint system are described by a constraint system, i.e. a set of constraints with a structure of an algebraic lattice.

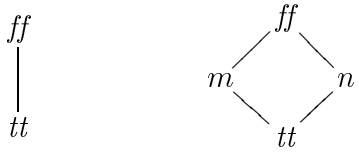
Definition 4.1 A *constraint system* is a bounded lattice $(C, \vdash, \wedge, tt, ff)$, where C is the set of *constraints*, $\vdash$ (called *entailment*) is an ordering on this set, $\wedge$ is the lub operation, and tt (*true*), ff (*false*) are the least and the greatest

elements of C respectively ($ff \vdash tt$ and $tt \neq ff$).

In algebra, the symbol $\wedge$ usually denotes the *glb* operation, while *lub* operation is rather marked with symbol $\vee$. Our notation of lub operation corresponds to logical conjunction (as in CCP).

We say that a constraint m is *consistent* with a constraint n iff $m \wedge n \neq ff$. The state of the store cannot be ff as we require the consistency of the store initialised to tt. We use $C^\diamond$ to denote $C \setminus \{ff\}$.

Example 4.2 Let $\mathcal{C}_\varepsilon$ be the trivial constraint system $(\{tt, ff\}, \vdash, \wedge, tt, ff)$, where $\vdash = \{(ff, tt), (tt, tt), (ff, ff)\}$, and $\mathcal{C}_{mn}$ the constraint system $\mathcal{C}_{mn} = (\{tt, m, n, ff\}, \vdash, \wedge, tt, ff)$, where $\vdash = \{(ff, tt), (m, tt), (n, tt), (ff, m), (ff, n)\} \cup \{(o, o) \mid o \in C\}$. These constraint systems are depicted below.



Definition 4.3 Let $\alpha, \beta \in \{1, S, P, G\}$ such that $\alpha \subseteq \beta$. An *(α, β)-fcPRS (PRS with finite constraint system)* Δ is a tuple $(\mathcal{C}, R, t_0)$, where

- $\mathcal{C} = (C, \vdash, \wedge, tt, ff)$ is a finite constraint system describing the *store*; the elements of C represent the *states of the store*,
- R is a finite set of *rewrite rules* of the form $(t_1 \xrightarrow{a} t_2, m, n)$, where $t_1 \in \alpha$, $t_1 \neq \varepsilon$, $t_2 \in \beta$ are process terms, $a \in Act$ is an atomic action, and $m, n \in C^\diamond$ are constraints,
- $t_0 \in \beta$ is a distinguished *initial process term*.

A (G, G)-fcPRS is simply called fcPRS.

We use human-readable abbreviations fcFS, fcBPA, fcBPP, fcPA, fcPDA, fcPN, fcPAD, fcPAN, and fcPRS for classes $(1,1)$-fcPRS, $(1,S)$-fcPRS, $(1,P)$-fcPRS, $(1,G)$-fcPRS, (S,S)-fcPRS, (P,P)-fcPRS, (S,G)-fcPRS, (P,G)-fcPRS, and (G,G)-fcPRS respectively.

Again, instead of $(t_1 \xrightarrow{a} t_2, m, n) \in R$ where $\Delta = (\mathcal{C}, R, t_0)$, we usually write $(t_1 \xrightarrow{a} t_2, m, n) \in \Delta$. The meaning of sets $Const(\Delta)$ (process constants used in rewrite rules) and $Act(\Delta)$ (actions occurring in rewrite rules) for a given fcPRS Δ is the same as in PRS case. Again, it can be assumed the initial term t_0 of an (α, β)-fcPRS is a single constant.

Every PRS with finite constraint system denotes a labelled transition system. Let $\Delta = (\mathcal{C}, R, t_0)$ be an (α, β)-fcPRS. The LTS $\mathcal{L}$ denoted by Δ has the form $(S, Act(\Delta), \longrightarrow, (t_0, tt))$, where $S = \{t \in \beta \mid Const(t) \subseteq Const(\Delta)\} \times C^\diamond$ is the set of states, (t_0, tt) is the initial state and transition relation $\longrightarrow$ is defined as the least relation that satisfies the inference rules

$$\frac{(t_1 \xrightarrow{a} t_2, m, n) \in \Delta}{(t_1, o) \xrightarrow{a} (t_2, o \wedge n)} \quad \text{if } o \vdash m \text{ and } o \wedge n \neq ff,$$

$$\frac{(t_1, o) \xrightarrow{a} (t_1', p)}{(t_1 \| t_2, o) \xrightarrow{a} (t_1' \| t_2, p)}, \qquad \frac{(t_1, o) \xrightarrow{a} (t_1', p)}{(t_1 . t_2, o) \xrightarrow{a} (t_1' . t_2, p)},$$

where $t_1, t_2, t_1' \in \mathcal{T}$ and $m, n, o, p \in C^\diamond$.

The two side conditions in the first inference rule are very close to principles used in CCP. The first one ($o \vdash m$) ensures the rule $(t_1 \xrightarrow{a} t_2, m, n) \in \Delta$ can be used only if the current state of the store o *entails* m (it is similar to *ask(m)* in CCP). The second condition ($o \wedge n \neq ff$) guarantees that the store stays *consistent* after application of the rule (analogous to the consistency requirement when processing *tell(n)* in CCP).

An important observation is that the state of the store (starting at tt) can move in a lattice $\mathcal{C}$ only in one direction, from tt upwards. This can be easily seen from the fact that the actual state of the store o can be changed only by applying some rewrite rule $(t_1 \xrightarrow{a} t_2, m, n) \in \Delta$ and after this application the new state of the store $o \wedge n$ always entails o. Intuitively, the partial information can only be added to the store, not retracted. We say the store is *monotonic*.

Note that when the system (with o on the store) executes a transition generated by a rule $(t_1 \xrightarrow{a} t_2, m, n) \in \Delta$ then for every subsequent state of the store p conditions $p \vdash m$ and $p \wedge n \neq ff$ are satisfied. The first condition $p \vdash m$ comes from the monotonic behaviour of the store. The second condition comes from the facts that the constraint n in the rule can change the store only in the first application of the rule and that for each subsequent state p of the store $p \wedge n = p$ holds.

On the other hand, the fact that some rule is applicable (hence entailment and consistency are satisfiable) does not imply that this rule is applicable forever. The insidious point is the consistency requirement. The store can evolve to a state inconsistent with the second constraint from the rule.

The first information about the relationship between fcPRS and PRS is provided by the following lemma.

Lemma 4.4 *Let $\alpha, \beta \in \{1, P, S, G\}$. The systems (α, β)-PRS $\Delta' = (R', t_0)$ and (α, β)-fcPRS $\Delta = (\mathcal{C}_\varepsilon, R, t_0)$ are isomorphic on the assumption that $R' = \{t_1 \xrightarrow{a} t_2 \mid (t_1 \xrightarrow{a} t_2, tt, tt) \in R\}$.*

Proof. It is easy to check that if we remove tt from the states of LTS generated by fcPRS Δ, we get an isomorphic system which corresponds to the PRS Δ'.□

The lemma above says that PRS classes can be seen as fcPRS classes with a trivial constraint system. The lemma can be used in both directions, to show that any fcPRS of the specified form has an equivalent PRS as well as for constructing an fcPRS equivalent to a given PRS.

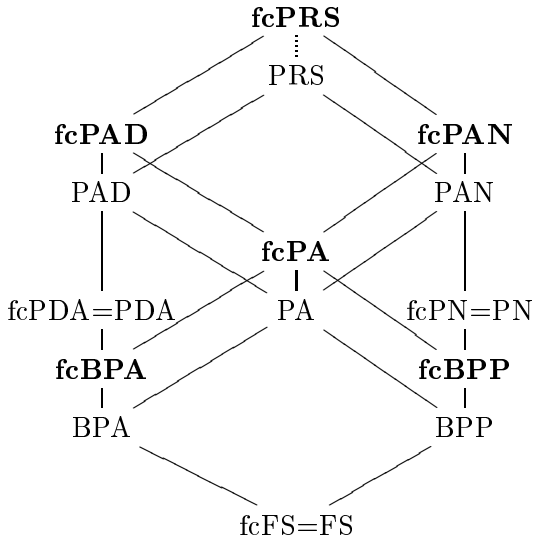


Fig. 2. The fcPRS-hierarchy

5 The fcPRS-hierarchy

Figure 2 shows the hierarchy of PRS and fcPRS classes, simply called *fcPRS-hierarchy*. The relations depicted in the hierarchy partly result from the definition of classes and Lemma 4.4. The rest of the paper is dedicated to three equalities (fcFS = FS, fcPDA = PDA, and fcPN = PN) and the strictness of the hierarchy.

Theorem 5.1 *(i) Let Δ be an fcFS. There exists FS Δ' denoting a labelled transition system isomorphic to the one given by Δ.*
(ii) Let Δ be an fcPDA. There exists PDA Δ' denoting a labelled transition system isomorphic to the one given by Δ.
(iii) Let Δ be an fcPN. There exists PN Δ' denoting a labelled transition system isomorphic to the one given by Δ.

Proof. (i) The construction is obvious, every state (X, m) of Δ is transformed into state $X^{(m)}$ of Δ'.

(ii) The idea of the proof is based on the fact that we can add special process constants corresponding to the actual states of the store, one to each state of fcPDA. Then the content of the store will be represented by such special constants.

Let $\Delta = (\mathcal{C}, R, t_0)$, where $\mathcal{C} = (C, \vdash, \wedge, tt, ff)$. Let $S = \{S^{(m)} \mid m \in C^{\diamond}\}$ be the set of special process constants. A PDA Δ' is constructed as $(R', S^{(tt)}.t_0)$, where $S^{(tt)}.t_0$ is the initial term with the special constant holding the initial state of the store. We replace every rule

$$(t_1 \xrightarrow{a} t_2, m, n) \in R$$

by the set of rules

$$(S^{(o)}.t_1 \xrightarrow{a} S^{(o \wedge n)}.t_2) \in R'$$

for every $o \in C^\diamond$ which satisfies the entailment condition $o \vdash m$ and the consistency condition $o \wedge n \neq ff$. The new rules are constructed to abide by the entailment and consistency conditions connected with the original rules. The isomorphism of Δ and Δ' is obvious as every state $S^{(m)}.t$ of Δ' corresponds exactly to the state (t, m) of the system Δ.

(iii) The proof is the same as for (ii) if we replace every sequential composition by the parallel composition. □

As the PRS-hierarchy is not strict w.r.t. the language equivalence, the fcPRS-hierarchy cannot also be strict on the language expressibility level. However, the fcPRS-hierarchy is strict w.r.t. the bisimulation equivalence with possibly one exception: the relation between PRS and fcPRS (this case will be discussed later). To prove that each of the classes fcBPA, fcBPP, fcPA, fcPAD, and fcPAN differs from the corresponding standard class, we present two fcPRS systems. The first one is an fcBPA system which is not bisimilar to any PAN system. The second system will be an fcBPP which is not bisimilar to any PAD system.

Example 5.2 Let us consider an fcBPA system with the constraint system $\mathcal{C}_{mn}$ from Example 4.2 and the initial process term $U.X$.

$$\begin{array}{ll} (U \xrightarrow{a} U.A, tt, tt) & (A \xrightarrow{a} \varepsilon, tt, tt) \\ (U \xrightarrow{b} U.B, tt, tt) & (B \xrightarrow{b} \varepsilon, tt, tt) \\ (U \xrightarrow{c} \varepsilon, tt, m) & (X \xrightarrow{e} \varepsilon, m, tt) \\ (U \xrightarrow{d} \varepsilon, tt, n) & (X \xrightarrow{f} \varepsilon, n, tt) \end{array}$$

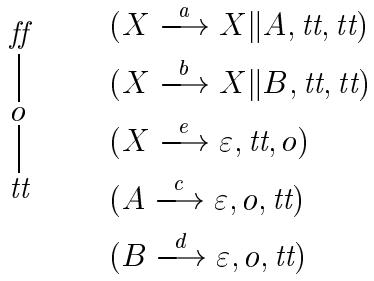
The fcBPA above is bisimilar to the PDA system described in 3.2 which is not bisimilar to any PAN system and thus also the considered fcBPA process is not bisimilar to any PAN system. Hence we obtain a following corollary, where $X \subsetneq Y$ means that X is a strict subclass of Y and $X \not\subseteq Y$ means that X is not a subclass of Y.

Corollary 5.3 *BPA $\subsetneq$ fcBPA, PA $\subsetneq$ fcPA, PAN $\subsetneq$ fcPAN and fcBPA $\not\subseteq$ PA, fcBPA $\not\subseteq$ PAN, fcPA $\not\subseteq$ PAN.*

Proof. Directly from the definition of BPA and PAN classes and from it follows that the BPA class is a subclass of the PAN class. Lemma 4.4 implies that the BPA class is a subclass of the fcBPA class. We know that there exists an fcBPA system which is not bisimilar to any PAN system and thus also to any BPA system. Hence we know that BPA is strict subclass of fcBPA. The proofs of the other relations are similar. □

Example 5.4 Let us consider an fcBPP system with the constraint system

depicted below and the initial state (X, tt).



Lemma 5.5 *If there is a PAD system bisimilar to the fcBPP system from Example 5.4, then there is also a PDA system bisimilar to this fcBPP.*

Proof. Let Δ be a PAD with the initial state Q (w.l.o.g.) such that Q is bisimilar to the initial state (X, tt) of considered fcBPP system. As on the left-hand side of rewrite rules Δ only sequential composition can occur, some part of parallel composition $t_1\|t_2$ can influence the behaviour of such system only if there is a reachable state of the form $(t_1\|t_2).t_3$ where t_3 can be ε. If there is no such a state, we can remove all parallel compositions from the rules and we get a PDA system bisimilar to Δ and thus also bisimilar to the considered fcBPP process.

Another situation arises if there is a reachable state of Δ of the form $(t_1\|t_2).t_3$, where t_3 can be ε. Let us assume that during the derivation of the state $(t_1\|t_2).t_3$ from Q there is no other state of the form $(t'_1\|t'_2).t'_3$ (t_3 can be ε). As Q is a single process constant and any parallel composition $s_1\|s_2$ in a term $p.(s_1\|s_2).p'$ cannot be changed by any rewriting until p is ε, there must be some rewrite rule $(t \xrightarrow{x} l.(t_1\|t_2).r) \in \Delta$ (l, r can be ε, $x \in \{a, b, c, d, e\}$) such that $t_1\|t_2$ is the mentioned parallel composition. There are two cases.

(i) The state $(t_1\|t_2).t_3$ was derived from Q under a word $w \in \{a, b\}^*$. We show that t_1 or t_2 is then deadlocked. With respect to the definition of PAD, which does not provide any form of communication or synchronisation between processes in a parallel composition, just one component of $t_1\|t_2$ can enable the action e, let us assume that it is t_2. Then t_1 is deadlocked – it cannot do neither the actions a or b (as these actions are disabled after the action e) nor the actions c or d (as these actions are disabled before e). Nevertheless, the term $t_1.t'$ is not necessarily deadlocked for some term t'. Hence, the parallel composition $t_1\|t_2$ in the rule $(t \xrightarrow{x} l.(t_1\|t_2).r) \in \Delta$ can be changed to the sequential composition $t_2.t_1$. We should insert some separator between t_2 and t_1 (resp. l and t_2) to keep the impossibility of communication between parts of parallel composition (resp. between l and part of the following parallel composition). Thus we replace the rule $(t \xrightarrow{x} l.(t_1\|t_2).r) \in \Delta$ by the rule $t \xrightarrow{x} l.X.t_2.X.t_1.r$ (resp. $t \xrightarrow{x} t_2.X.t_1.r$ if $l = \varepsilon$), where $X \notin Const(\Delta)$ is a new constant, and we add new rewrite rule $X.s \xrightarrow{x} s'$ to Δ for every rewrite rule $s \xrightarrow{x} s' \in \Delta$ (if we already have the rules of the form $X.s \xrightarrow{x} s'$

in modified Δ, we do not need to add them again in the future). These changes do not affect the behaviour of Δ.

(ii) The action e occurs during the derivation of the state $(t_1 \| t_2).t_3$ from Q. The state $(t_1 \| t_2).t_3$ is then bisimilar to a state $(A^n \| B^m, o)$ [5] of considered fcBPP and thus every possible sequence of actions performed by the process $(t_1 \| t_2).t_3$ is finite, as well as every possible sequence performed by the term $t_1 \| t_2$. We construct a finite labelled (acyclic) transition graph where the vertices are processes reachable from the parallel composition $t_1 \| t_2$ (which is the root of the graph) and edges naturally correspond to actions (resp. applications of rewrite rules). Now we assign a fresh process constant to each vertex of the graph which has some parallel composition inside (the vertices without any parallel composition keep unchanged). We replace the rule $(t \xrightarrow{x} l.(t_1 \| t_2).r) \in \Delta$ by the rule $t \xrightarrow{x} l.Z.r$, where $Z \notin Const(\Delta)$ is a process constant assigned to $t_1 \| t_2$. For every edge of the graph from the vertex A (where A is a fresh constant) to the vertex v we add a rule $A \xrightarrow{x} v$ (where x is the label of the edge) to Δ. The behaviour of Δ is still unchanged thanks to the fact that if $(t_1 \| t_2).t_3 \xrightarrow{*} t'.t_3$ then the term t_3 can be changed by the following transition only if there is no parallel composition in t', and the fact that the vertices without any parallel composition are unchanged.

In both cases, the number of parallel compositions in rewrite rules has decreased (with one exception – when we add rules of the form $X.s \xrightarrow{x} s'$, then the number of parallel compositions can be doubled, but it does not matter as we make it only once). If there is still a reachable state of the form $(t_1 \| t_2).t_3$ in modified Δ, we can use the same method again. As the number of parallel compositions in rewrite rules is finite, after finite number of steps we get a PAD system without any reachable state of the form $(t_1 \| t_2).t_3$, which is the situation discussed at the beginning of this proof. □

The class of context-free languages (i.e. the class of languages generated by PDA processes) is closed under intersection with regular languages. The language L generated by the fcBPP system from Example 5.4 is not context-free, as $L \cap a^*b^*ec^*d^* = \{a^n b^m e c^n d^m \mid m, n \geq 0\}$ which is not context-free. Thus there is no PDA process bisimilar to fcBPP from Example 5.4 and from Lemma 5.5 it follows that there is no PAD process bisimilar to the fcBPP presented above. Hence we get:

Corollary 5.6 *BPP $\subsetneq$ fcBPP, PAD $\subsetneq$ fcPAD and fcBPP $\not\subseteq$ PA, fcPA $\not\subseteq$ PAD, fcBPP $\not\subseteq$ PAD.*

The fcBPP class differs from PN even w.r.t. language equivalence. The language $L = \{a^n b c^n d e^n f \mid n \geq 0\}$ generated by PN from Example 5.7 is an

[5] The expression A^n is an abbreviation for n copies of process constant A in parallel composition. The abbreviation B^m has an analogous meaning.

instance of a language generated by PN, which cannot be described by any fcBPP due to the following Pumping Lemma.

Example 5.7 Let $\Delta = (R, W)$ be a Petri net with rewrite rules as below.

$$W \xrightarrow{a} W\|A\|B \qquad Y\|B \xrightarrow{e} Y$$
$$W \xrightarrow{b} X \qquad Y \xrightarrow{f} Z$$
$$X\|A \xrightarrow{c} X \qquad Z\|A \xrightarrow{z} Z\|A$$
$$X \xrightarrow{d} Y \qquad Z\|B \xrightarrow{z} Z\|B$$

Lemma 5.8 (Pumping Lemma for fcBPP) *Let L be a language of an fcBPP system Δ. There exists a constant h such that if $u \in L$ and $|u| > h$ then there exist $x, y, z, w \in Act^*$ such that $u = xz$, $|y| > 1$, and $\forall i \geq 0$ it holds that $xy^izw^i \in L$.* [6]

Proof. The proof can be found in Appendix A. □

To prove the strictness of the fcPRS-hierarchy completely we introduce a PDA process which is not bisimilar to any fcPAN process and a PAN process which is not bisimilar to any fcPAD process.

Example 5.9 Let us consider a PDA system described in Example 3.2 with initial state $U.X.Y$ and with the following additional rewrite rules.

$$V.Y \xrightarrow{x} U.X.Y \qquad W.Y \xrightarrow{x} U.X.Y$$
$$V.Y \xrightarrow{z} Z \qquad W.Y \xrightarrow{z} Z$$

This system behaves like that defined in Example 3.2, but when the original system terminates, the enhanced system can choose between termination under the action z and restart under the action x.

Lemma 5.10 *There is no fcPAN system bisimilar to the PDA process given in Example 5.9.*

Proof. We assume the contrary and derive a contradiction. Let Δ be an fcPAN bisimilar to the PDA process defined in Example 5.9. From the finiteness of the constraint system used in Δ follows that there exists a non-terminal reachable state (t, o) of Δ such that every non-terminal state reachable from (t, o) has also o on the store (the contrary implies the infiniteness of the constraint system). As (t, o) is non-terminal, there exist a word $w \in \{a, b, c, d, e, f\}^*$ such that $(t, o) \xrightarrow{w.x} (s, o)$, where (s, o) is bisimilar to the state $U.X.Y$ of the PDA process from Example 5.9. If the rules labelled by actions x, z are removed from Δ and (s, o) is taken as an initial state, we obtain the system whose reachable states all have o as their store, bisimilar to the pushdown process from Example 3.2.

[6] $|u|$ denotes the length of the word u.

Now, let Δ' be a PAN system with the initial state s and with the set of rewrite rules consisting of rules $l \xrightarrow{v} r$, where $(l \xrightarrow{v} r, m, n) \in \Delta$, $o \vdash m$, $o \wedge n = o$ and $v \in \{a, b, c, d, e, f\}$. It is obvious that this PAN system Δ' is bisimilar to the PDA system defined in Example 3.2. This is a contradiction.□

Corollary 5.11 *fcBPA $\subsetneq$ PDA, fcPA $\subsetneq$ fcPAD and fcPAD $\not\subseteq$ fcPAN.*

Example 5.12 Let Δ be a PAN process with the initial state $(X\|A\|B).W$ and the following rewrite rules.

$$\begin{array}{llll} X \xrightarrow{g} X\|A\|B & Y\|A \xrightarrow{a} Y & X \xrightarrow{y} \varepsilon & W \xrightarrow{x} (X\|A\|B).W \\ X \xrightarrow{c} Y & Y\|B \xrightarrow{b} Y & Y \xrightarrow{y} \varepsilon & W \xrightarrow{z} D \\ X\|A \xrightarrow{d} Z & Y\|A \xrightarrow{d} Z & Z \xrightarrow{y} \varepsilon & \\ X\|B \xrightarrow{d} Z & Y\|B \xrightarrow{d} Z & A \xrightarrow{y} \varepsilon & \\ & & B \xrightarrow{y} \varepsilon & \end{array}$$

The first two columns of rewrite rules include the same rules as Petri net given by Example 3.3. This PAN system can behave as mentioned Petri net (it can deviate from the behaviour of PN only under action y). States of PAN corresponding to terminal states of considered PN can perform a sequence of actions y^* to reach the state W and then terminate under action z or restart the system under action x.

Lemma 5.13 *There is no fcPAD system bisimilar to the PAN process from Example 5.12.*

Proof. The proof is similar to the proof of the previous lemma, instead of PDA from Example 3.2 it uses Petri net from Example 3.3. □

Corollary 5.14 *fcPA $\subsetneq$ fcPAN and fcPAN $\not\subseteq$ fcPAD.*

The incomparability of fcPAD and fcPAN implies that these classes are strict subclasses of fcPRS.

The edge between PRS and fcPRS classes in the fcPRS-hierarchy is dotted as we have no proof that the fcPRS class is strictly more expressive (w.r.t. bisimilarity) than the PRS class. It is obvious from the definitions that PRS $\subseteq$ fcPRS, but we can provide only intuition for PRS $\subsetneq$ fcPRS. The conjectured witness of the inequality can be found in the fcPA below.

Example 5.15 Let Δ be an fcPA system with the initial process term $X\|Y$

and the following constraint system and rewrite rules.

$$
\begin{array}{c}
ff \\ | \\ o \\ | \\ p \\ | \\ tt
\end{array}
\qquad
\begin{array}{ll}
(X \xrightarrow{a} X.A, tt, tt) & (A \xrightarrow{a'} \varepsilon, o, tt) \\
(X \xrightarrow{b} X.B, tt, tt) & (B \xrightarrow{b'} \varepsilon, o, tt) \\
(Y \xrightarrow{c} Y \| C, tt, tt) & (C \xrightarrow{c'} \varepsilon, o, tt) \\
(X \xrightarrow{x} \varepsilon, tt, p) & \\
(Y \xrightarrow{y} \varepsilon, p, o) &
\end{array}
$$

We can prove that this fcPA system is not bisimilar to any PAD process and to any Petri net either.

Now we try to explain why we conjecture that there is no PRS process bisimilar to the considered fcPA. The weak point of PRS (or rewrite system in general) is the "local potency" of rewriting. Having a parallel composition with at least one sequential component larger than the left side of any rewrite rule, the rule cannot influence both this large component and the rest of the parallel composition at once. Roughly speaking, communication between large component and other component(s) of parallel composition is not possible in general. Any PRS process bisimilar to the fcPA system under consideration should have such a parallel composition with one component which has a sequential character (as it is necessary to keep the information about the order in which the actions a and b are performed) and it can be arbitrary large. And we need to announce to the term that action y has just been done.

6 Conclusion

We have enriched process rewrite systems with the mechanism related to computing with partial information in the form used in widely studied concurrent constraint programming. In the case of process rewrite systems, this mechanism can be effectively used to provide some information to every part of the process term, thus it can be seen as a unit holding a special kind of global information.

It has been proven that enriching the classes of finite systems, pushdown processes, and Petri nets with a finite constraint system does not change their expressibility even w.r.t. isomorphism of the generated labelled transition systems. On the contrary, the process rewrite systems classes BPA, BPP, PA, PAD, and PAN extended with finite constraint systems establish corresponding new classes fcBPA, fcBPP, fcPA, fcPAD, and fcPAN as the expressive power of such systems increases. This may seem quite surprising in the cases of PAD and PAN classes as the formalism of these classes subsumes the formalism of PDA or PN respectively. However PDA and PN do not increase their expressive power if enriched with a finite constraint system.

The hierarchy of fcPRS classes has been introduced and its strictness w.r.t. the bisimulation equivalence (with the exception in the relation between

PRS and fcPRS classes) has been proven.

The area of process rewrite systems with finite constraint systems still offers some interesting topics for further research. One interesting challenge is to specify the boundary of decidability of the bisimulation equivalence and the weak bisimulation equivalence with finite-state processes in the area of fcBPP class (as both problems are decidable for BPP and undecidable in the case of MSA [7]). Another possible topic for further research is to replace the constraint system with a (finite) state unit, where the evolution of the actual state is determined by a given ordering. A totally different mission is to employ an infinite constraint system.

Acknowledgements: I would like to thank Mojmír Křetínský for valuable discussions, reading the draft, and constant support. My thanks go also to Antonín Kučera for encouragement and inspiring comments. I thank three anonymnous referees for helpful remarks.

References

[1] Bergstra, J. A., and J. W. Klop, *Algebra of communicating processes with abstraction*, Theoretical Computer Science **37** (1985), 77–121.

[2] Burkart, O., D. Caucal, and B. Steffen, *Bisimulation collapse and the process taxonomy*, Proceedings of CONCUR '96, Lecture Notes in Computer Science, vol. 1119, Springer-Verlag, 1996, 247–262.

[3] Burkart, O., and J. Esparza, *More infinite results*, Bulletin of the European Association for Theoretical Computer Science **62** (1997), 138–159.

[4] Caucal, D., *On the regular structure of prefix rewriting*, Theoretical Computer Science **106** (1992), 61–86.

[5] Christensen, S., "Decidability and Decomposition in Process Algebras," Ph.D. thesis, Department of Computer Science, University of Edinburgh, 1993.

[6] Groote, J. F., and F. Vaandrager, *Structured operational semantics and bisimulation as a congruence*, Information and Computation **100(2)** (1992), 202–260.

[7] Mayr, R., *Combining Petri nets and PA-processes*, Proceedings of TACS '97, Lecture Notes in Computer Science, vol. 1281, Springer-Verlag, 1997, 547–561.

[8] Mayr, R., *Process rewrite systems*, Electronic Notes in Theoretical Computer Science **7** (1997).

[7] MSA are in [11] called PPDA. In [12] it was also demonstrated that the class of MSA is a strict subclass of Petri nets. It was proven in [15] that fcBPP is a strict subclass of MSA and that the expressibility of MSA systems is not changed by enriching them with finite constraint systems.

[9] Milner, R., "A calculus on communicating systems," Lecture Notes in Computer Science, vol. 92, Springer-Verlag, 1980.

[10] Milner, R., "Communication and Concurrency," Prentice Hall, 1989.

[11] Moller, F., *Infinite results*, Proceedings of CONCUR '96, Lecture Notes in Computer Science, vol. 1119, Springer-Verlag, 1996, 195–216.

[12] Moller, F., *A taxonomy of infinite state processes*, Electronic Notes in Theoretical Computer Science **18** (1998).

[13] Park, D. M. R., *Concurrency and automata on infinite sequences*, Theoretical Computer Science: 5th GI-Conference, Lecture Notes in Computer Science, vol. 104, Springer-Verlag, 1981, 167–183.

[14] Saraswat, V. A., "Concurrent Constraint Programming Languages," Ph.D. thesis, Computer Science Department, Carnegie Mellon University, 1989.

[15] Strejček, J., *Constrained rewrite transition systems*, Technical Report FIMU-RS-2000-12, Faculty of Informatics, Masaryk University Brno, 2000, URL: `http://www.fi.muni.cz/informatics/reports/`.

A Appendix - Pumping lemma for fcBPP (Lemma 5.8)

The pumping lemma for fcBPP is formulated and proved in this appendix. The proof is similar to the one presented by Christensen for BPP case [5] thanks to the fact that every possible sequence of actions contains a finite number of transitions which change the state of the store due to finiteness of a constraint system.

Let $\Delta = (\mathcal{C}, R, t_0)$ be an fcBPP. For every process constant $X \in Const(\Delta)$ and every constraint $m \in C^\diamond$, let $S_m(X)$ denote the set

$$S_m(X) = \{Y \in Const(\Delta) \mid \exists t \in P : (X, m) \longrightarrow^+ (Y \| t, m)\},$$

i.e. the set of process constants Y which can be derived [8] from (X, m) without changes on the store. We extend this definition to parallel terms in obvious manner:

$$S_m(A_1 \| A_2 \| \ldots \| A_j) = \bigcup_{i \in \{1,2,\ldots,j\}} S_m(A_i)$$

Lemma A.1 *Let $\Delta = (\mathcal{C}, R, t_0)$ be an fcBPP. If there exists some derivation of a word $u = u_1 u_2 \ldots u_k \in L(\Delta)$ of the form*

$$(t_0, tt) = (t_0, m_0) \xrightarrow{u_1} (t_1, m_1) \xrightarrow{u_2} \ldots \xrightarrow{u_k} (t_k, m_k) \not\longrightarrow$$

such that $\forall i \in \{0, 1, 2, \ldots, k\}, \forall X \in t_i$ it holds $X \notin S_{m_i}(X)$, then $|u| \leq h$, where h is a constant depending only on Δ.

[8] The relation $\longrightarrow^+$ (resp. $\longrightarrow^*$) is apprehended as usual, i.e. $(t_1, m) \longrightarrow^+ (t_2, n)$ (resp. $(t_1, m) \longrightarrow^* (t_2, n)$) iff there exists $w \in Act^+$ (resp. $w \in Act^*$) such that $(t_1, m) \xrightarrow{w} (t_2, n)$.

Proof. At first we focus on maximum "flat" parts of the above derivation, which are of the form

$$(t_i, m_i) \xrightarrow{u_{i+1}} (t_{i+1}, m_{i+1}) \xrightarrow{u_{i+2}} \ldots \xrightarrow{u_{i+j}} (t_{i+j}, m_{i+j}),$$

where the state of the store (in following marked as m) keeps unchanged ($m = m_i = m_{i+1} = \ldots = m_{i+j}$), $i = 0$ or $m_{i-1} \neq m$, and $i + j = k$ or $m \neq m_{i+j+1}$. We denote $u' = u_{i+1}u_{i+2}\ldots u_{i+j}$. From this flat part we deduce another derivation sequence

$$(r_0 \| s_0, m) \xrightarrow{v_1} (r_1 \| s_1, m) \xrightarrow{v_2} \ldots \xrightarrow{v_p} (r_p \| s_p, m),$$

where $v_1, v_2, \ldots, v_p \in Act^+$, $r_0 \| s_0 = t_i$, in r_0 there are all constants from t_i which are rewritten in the derivation sequence $(t_i, m) \xrightarrow{u'} (t_{i+j}, m)$, and in s_0 there are constants which do not actively participate in this derivation sequence. Now $r_l \| s_l$ ($l = 1, 2, \ldots, p$) rises from $r_{l-1} \| s_{l-1}$ by one rewriting of each constant from r_{l-1} in the same way as a constant has been rewritten in the original flat derivation sequence (thus $|v_l| = |r_{l-1}|$) and still it holds that r_l contains constants, which are rewritten in the original flat derivation sequence, while s_l contains the other constants (thus $s_{l-1} \subseteq s_l$). We finish rewriting when r_l is empty (thus $r_p = \varepsilon$ and $s_p = t_{i+j}$). It is clear that $v = v_1 v_2 \ldots v_p$ is a permutation of u', especially $|v| = |u'|$. By replacing $(t_i, m) \xrightarrow{u'} (t_{i+j}, m)$ with $(r_0 \| s_0, m) \xrightarrow{v} (r_p \| s_p, m)$ in the original derivation we get a correct derivation of the word $u_1 \ldots u_i v u_{i+j+1} \ldots u_n$ of the length k. Further, for each X in r_l ($l = 0, 1, 2, \ldots, p$) there exists t_z ($i \leq z \leq i + j$) such that $X \in t_z$.

Now we show that $S_m(r_{l-1}) \supsetneq S_m(r_l)$ for each $1 \leq l < p$.

"$\supseteq$" It comes directly from the fact that each constant from r_l has an ancestor in r_{l-1}.

"$\neq$" Let us assume that for some $1 \leq l < p$ we have $S_m(r_{l-1}) = S_m(r_l)$. For each $X \in r_l$ ($r_l \neq \varepsilon$) it holds that $X \in S_m(r_{l-1})$ and thus $X \in S_m(r_l)$. From the premise $X \notin S_m(X)$ follows that there exists some $Y \in r_l$, $Y \neq X$ such that $X \in S_m(Y)$. Analogous reasoning as for X can be done for Y, i.e. from $Y \in r_l$ it follows that $Y \in S_m(r_{l-1}) = S_m(r_l)$ and $Y \notin S_m(Y)$, $Y \notin S_m(X)$. In conclusion we get $Y \in S_m(r_l)$ and $Y \notin S_m(X \| Y)$. Again, there exists $Z \in r_l$, $Z \notin \{X, Y\}$ such that $Y \in S_m(Z)$ and thus also $\{X, Y\} \subseteq S_m(Z)$. We know $Z \in r_l$ and $Z \notin S_m(Z)$, hence we get $Z \in S_m(r_l)$ and $Z \notin S_m(X \| Y \| Z)$. We can continue in this fashion to the point where we have the contradiction $W \in S_m(r_l)$ and $W \notin S_m(r_l)$.

Hence we have

$$|Const(\Delta)| \geq |S_m(r_0)| > |S_m(r_1)| > \ldots > |S_m(r_{p-1})| \geq 0.$$

This implies $|Const(\Delta)| \geq p - 1$. Further, for each $1 \leq l \leq p$ it holds that

$$|v_l| = |r_{l-1}| \leq |r_0| a^{l-1} \leq |r_0| a^{p-1} \leq |r_0| a^{|Const(\Delta)|},$$

where a is a maximum number of constants in right sides of rewrite rules in Δ. Now we restrict the length of u'

$$|u'| = |v| = \sum_{l=1}^{p} |v_l| \leq \sum_{l=1}^{p} |r_0| a^{|Const(\Delta)|} = p|r_0|a^{|Const(\Delta)|},$$
$$|u'| \leq p|r_0|a^{|Const(\Delta)|} \leq (|Const(\Delta)| + 1)|t_i|a^{|Const(\Delta)|}.$$

In conclusion we get the restriction on the length of flat parts of the original derivation

$$|u'| \leq |t_i|b,$$

where $b = (|Const(\Delta)| + 1)a^{|Const(\Delta)|}$.

In general it holds that each sequence of derivation steps consists of non-flat steps and flat derivation sequences. The number of "unflat" steps $(t_i, m_i) \xrightarrow{u_{i+1}} (t_{i+1}, m_{i+1})$, where $m_i \neq m_{i+1}$, is limited by $|C^\diamond| - 1$. The cardinality of the set C also constrains the number of flat parts to $|C^\diamond|$. Therefore

$$|u| \leq |C^\diamond| - 1 + \sum_{j=1}^{|C^\diamond|} |t'_j|b,$$

where (t'_j, m'_j) is the first state of the j-th flat derivation sequence, i.e. m'_j is the j-th different state of the store used in the original derivation and (t'_j, m'_j) is the first state in this derivation with the constraint m'_j in the store. Hence $(t'_1, m'_1) = (t_0, tt)$.

The last step is to restrict the length of t'_j for $j > 1$. We can deduce a restriction

$$|t'_j| \leq |t'_{j-1}| + (a-1)(|t'_{j-1}|b + 1)$$

thanks to the facts that each application of a rewrite rule cannot add more than $a - 1$ constants to the string of constants in the actual state and that the number of these applications is limited by the length of the previous flat string plus one (the unflat derivation step). The previous inequality can be modified in the following way.

$$|t'_j| \leq |t'_{j-1}| + a(|t'_{j-1}|b + 1)$$
$$|t'_j| \leq |t'_{j-1}|(1 + ab + a)$$
$$|t'_j| \leq |t'_1|(1 + ab + a)^{j-1}$$
$$|t'_j| \leq |t_0|(1 + ab + a)^{j-1}$$

By summarisation we get

$$|u| \leq |C^\diamond| - 1 + b|t_0| \sum_{j=1}^{|C^\diamond|} (1 + ab + a)^{j-1},$$

where $b = (|Const(\Delta)|+1)a^{|Const(\Delta)|}$. The sum on the right side of the previous inequality can be modified as it is an geometric progression. The final form of desired h is then

$$h = |C^\diamond| - 1 + b|t_0| \frac{(1 + ab + a)^{|C^\diamond|} - 1}{ab + a},$$

where a is the maximum number of constants in right sides of rewrite rules in Δ and $b = (|Const(\Delta)| + 1)a^{|Const(\Delta)|}$. □

The pumping lemma formulated below is a simple consequence of the previous lemma.

Lemma A.2 (Pumping Lemma for fcBPP) *Let L be a language of an fcBPP system Δ. There exists a constant h such that if u is a word of L and $|u| > h$ then there exist $x, y, z, w \in Act^*$ such that*

- $u = xz$,
- $|y| > 1$,
- $\forall i \geq 0 : xy^i zw^i \in L$.

Proof. We have an fcBPP Δ such that $L = L(\Delta)$. It follows from Lemma A.1 that each derivation

$$(t_0, tt) = (t_0, m_0) \xrightarrow{u_1} (t_1, m_1) \xrightarrow{u_2} \ldots \xrightarrow{u_k} (t_k, m_k) \not\longrightarrow$$

of the word $u = u_1 u_2 \ldots u_k \in L(\Delta)$, $|u| > h$ contains some state $(t_j, m_j) = (X \| t'_j, m_j)$, where $X \in S_{m_j}(X)$. The definition of $S_{m_j}(X)$ says that there exist $t \in P$ and $y \in Act^+$ such that $(X, m_j) \xrightarrow{y} (X \| t, m_j)$. Further, let $w \in Act^*$ be a word in $L((t, m_k))$, i.e. there exists a terminal state (t', n) such that $(t, m_k) \xrightarrow{w} (t', n)$. Now the derivation

$$(t_0, tt) \xrightarrow{u_1 \ldots u_j} (t_j, m_j) \xrightarrow{y^i} (t_j t^i, m_j) \xrightarrow{u_{j+1} \ldots u_k} (t^i, m_k) \xrightarrow{w^i} (t'^i, n) \not\longrightarrow$$

is the correct one for all $i \geq 0$. To make the proof complete we should add that $x = u_1 \ldots u_j$ and $z = u_{j+1} \ldots u_k$. □

On the decidability of fragments of the asynchronous π-calculus

Roberto M. Amadio Charles Meyssonnier [1,2]

Laboratoire d'Informatique Fondamentale de Marseille,
CMI, 39 rue Joliot-Curie, 13453, Marseille, France.

Abstract

We study the decidability of a reachability problem for various fragments of the asynchronous π-calculus. We consider the combination of three main features: *name generation*, *name mobility*, and *unbounded control*. We show that the combination of name generation with either name mobility or unbounded control leads to an undecidable fragment. On the other hand, we prove that name generation without name mobility and with bounded control is decidable by reduction to the coverability problem for Petri Nets.

1 Introduction

We are interested in *properties* of the reduction relation such as reachability, deadlock, liveness,... for process calculi based on the *asynchronous* π-calculus [2,7,1].

We recall that 'asynchronous' here refers to a communication mechanism where messages are put in an unbounded and unordered buffer and that in the process calculus jargon this amounts to disallow the *output prefix*. By opposition, the *synchronous* π-calculus forces a synchronization between the sender and the receiver.

Our interest in the asynchronous π-calculus stems from the observation that the core of concurrent programming languages such as Pict [13], Join [4], or Tyco [17] are based on it and the remark that object-oriented programming languages enjoy a rather direct representation in these formalisms.

In this paper, we will mainly consider a *minimal* asynchronous, polyadic, simply sorted π-calculus *not* including external choice and we will concentrate on three main 'features' of this minimal calculus:

[1] {amadio,meyssonn}@cmi.univ-mrs.fr.

[2] The authors are partially supported by RNRT MARVEL.

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

- Name generation, *i.e.* the possibility of generating fresh names (values, channels,...).
- Name mobility, *i.e.* the possibility of transmitting names.
- Unbounded control, *i.e.* the possibility of dynamically adding new threads of control.

In the absence of name generation, our formalism can be mapped to Petri Nets (see, *e.g.* [15]). This encoding, that basically goes back to early work [5] on the translation of CCS [11] to Petri Nets, settles most interesting decision problems for the fragment *without* name generation. Therefore, the main issue that, in our opinion, remains to be clarified is whether there exist *decidable* fragments that include some form of name generation.

So far, most decidability results we are aware of concern the *synchronous* π-calculus with *bounded* control (see, *e.g.*, [3,12]). In the *asynchronous* case, our main results are as follows:

- The combination of name generation and name mobility leads to an undecidable fragment even assuming the control finite.
- The combination of name generation and unbounded control leads to an undecidable fragment even assuming that no name is transmitted (this refines a well-known undecidability result for CCS).
- Name generation without name mobility and with bounded control is decidable by reduction to Petri Nets. This is our main technical result which is based on an analysis of the use of generated names. The analysis, which appears to be original, distinguishes between 'persistent' and 'temporary' names and provides a method to reuse the same name for generated temporary names which are alive at different times.

We regard these results as a first step towards the systematic introduction of approximated decision methods for languages including name generation. We expect that a fruitful approach is to understand these methods by factoring the approximation through a translation into Petri Nets. Once the behaviour is mapped to a Petri Net further standard approximation techniques are available based, *e.g.*, on semi-linear sets (see, *e.g.*, [16], for an up to date survey).

2 Asynchronous π-calculus

As usual, we assume given a denumerable set of *names*, that we denote $a, b, \ldots$ Vectors of names (possibly empty) are denoted $\vec{a}, \vec{b}, \ldots$ We denote with $[\vec{b}/\vec{a}]$ a substitution on names. If $\vec{a} \equiv a_1, \ldots, a_n$ then we use $(\nu\vec{a})$ as a shorthand for $(\nu a_1) \ldots (\nu a_n)$.

We suppose that every name a has an associated sort $st(a)$ and that names are used consistently with their sort. We will just rely on *simple* sorts as

defined by the following grammar

$$s ::= o \mid Ch(s, \ldots, s) \tag{1}$$

where o is some ground sort.

We consider a polyadic, simply sorted, asynchronous π-calculus with the standard operations of message creation $\overline{a}\vec{b}$, input prefix $a(\vec{b}).P$, parallel composition $P \mid Q$, name generation $(\nu a)P$, and parametric recursive definitions. The latter is preferred to *iteration* because it allows a better control on the creation and termination of parallel threads.

We denote with $A, B, \ldots$ parametric process identifiers. A *process* is presented by a finite system $\mathcal{E}$ of parametric equations $A(\vec{a}) = P$ and an initial configuration where we assume that: (i) every process identifier is defined by exactly one equation, and (ii) the names occurring free in P are included in $\{\vec{a}\}$. It will be convenient to assume that every equation has the following normalised shape:

$$A(\vec{a}) = a(\vec{a'}).(\nu\vec{a''})(\Pi_{i\in I}\overline{a}_i\vec{a}_i \mid \Pi_{j\in J}A_j(\vec{a}_j)) \ . \tag{2}$$

Such an equation specifies a process that inputs a message and then generates new names, sends a number of messages, and runs a number of continuations. The sets I and J are assumed finite (possibly empty, in which case the parallel composition reduces to the terminated process 0). We note that in equation (2) the names $\vec{a}$, $\vec{a'}$, and $\vec{a''}$ are bound. We will assume that they are renamed so that they are all distinct.

Given a finite system of recursive equations as above, a *configuration* is a normalised process of the shape:

$$(\nu\vec{a})(\Pi_{i\in I}\overline{a_i}(\vec{a}_i) \mid \Pi_{j\in J}A_j(\vec{a}_j))$$

where as usual 'Π' stands for the parallel composition. Let P, Q be two configurations. We write $P \equiv Q$ if P is syntactically equal to Q up to renaming of bound names, permutation of name generations, and associativity and commutativity of parallel composition. We denote with $fn(P)$ the set of names occurring free in P.

Next we introduce the reduction relation on configurations. All we want to capture is the usual reduction rule

$$\overline{a}\vec{b} \mid a(\vec{c}).P \to [\vec{b}/\vec{c}]P$$

allowed to take place under name generation and parallel composition, up to a suitable structural equivalence. Our definition of reduction is a bit technical because it has to evaluate the actual parameters, unfold a recursive definition to find an input prefix matching a message, and then bring the name generations, the messages, and the continuations under the input prefix at top level. The advantages of this approach, is that we can then limit the structural rules to the ones stated above, give a compact normal form for configurations, and provide a simple translation to Petri Nets.

Definition 2.1 If the equation associated to the process identifier A is (2)

and

(i) $P \equiv (\nu \vec{b'})(A(\vec{b}) \mid \overline{c}(\vec{c}) \mid Q)$,

(ii) the sets $\{\vec{a}, \vec{a'}, \vec{a''}\}$ and $\{\vec{b'}\} \cup fn(P)$ are mutually disjoint,

(iii) $\sigma \equiv [\vec{b}/\vec{a}, \vec{c}/\vec{a'}]$,

(iv) and $\sigma(a) = c$

then

$$P \to (\nu \vec{b'}, \vec{a''})(\Pi_{i \in I} \sigma(\overline{a}_i \vec{a}_i) \mid \Pi_{j \in J} A_j(\sigma \vec{a}_j) \mid Q) \ . \tag{3}$$

We may wonder whether our normalised configurations can represent all usual processes of the π-calculus, say:

$$p ::= \overline{a}\vec{b} \parallel a(\vec{b}).p \parallel !(a(\vec{b}).p) \mid (\nu a)p \parallel (p \mid p) \ .$$

Indeed, this can be easily checked. We note that, up to structural equivalence, a process p can always be written as:

$$p \equiv (\nu \vec{a})(\Pi_{i \in I} \overline{a}_i \vec{a}_i \mid \Pi_{j \in J} a_j(\vec{a}_j).p_j \mid \Pi_{k \in K} !(a_k(\vec{a}_k).p_k)) \ .$$

We claim that we can build a configuration P and a set of equations $\mathcal{E}$ whose behaviour is equivalent to p's. We proceed by induction on the structure of p to generate the set of equations. For every process $a_j(\vec{a}_j).p_j$ we introduce a fresh process identifier $A_j(\ldots)$ and the equation $A_j(\ldots) = a_j(\vec{a}_j).\ldots$, and we apply inductively the transformation to p_j. Similarly, for every process $!(a_k(\vec{a}_k).p_k))$ we introduce a fresh process identifier $A_k(\ldots)$ and the equation $A_k(\ldots) = a_k(\vec{a}_k).(A_k(\ldots) \mid \ldots)$, and we apply inductively the transformation to p_k.

Reassured about the expressivity of our formalism, we can now formally state the reachability problem we address in this paper.

Definition 2.2 *Given a system of equations $\mathcal{E}$ containing a process identifier A and a related initial configuration P, the reachability problem asks whether P reduces to a configuration containing the process identifier A,* i.e. *$P \to^* (\nu \vec{a})(\ldots \mid A(\vec{b}) \mid \ldots)$, for some $\vec{a}, \vec{b}$.*

In section 3.4, we will relate this problem to the well known coverability problem for Petri Nets.

3 The fragment without name generation reduces to Petri Nets

We consider the fragment where the equation (2) is restricted to having the shape:

$$A(\vec{a}) = a(\vec{b}).(\Pi_{i \in I} \overline{c}_i \vec{d_i} \mid \Pi_{j \in J} A_j(\vec{e_j})) \ . \tag{4}$$

In this fragment no name generation is allowed. Given such a system of equations and an initial configuration P we will recall below the standard construction of a Petri Net that simulates the reduction of the process.

3.1 Parameterless systems of equations

First we recall the notion of *parameterless* system of equations (a notation used, *e.g.*, in the context of CCS [11]). In this case, all names have sort $Ch()$ and an equation has the shape

$$A = \Sigma_{k\in K} a_k.(\Pi_{i\in I_k}\overline{a}_i \mid \Pi_{j\in J_k} A_j) \tag{5}$$

where K is a finite set and Σ stands for the *external choice* (external choice is just used here to represent an intermediate step towards the translation to Petri Nets). If K is empty, we take conventionally the left hand side as the terminated process 0. No renaming is allowed and a process identifier is *literally* replaced by the right hand side of the equation defining it.

3.2 From parameterless systems of equations to Petri Nets

We fix a system of equations without parameters of the shape (5). Let P be an initial configuration. Without loss of generality, we may assume that P contains no name generators ν; otherwise we replace the names bound by ν by fresh names. Let N be the collection of names free in P. Since there is no name generation, these are all the names that can appear in a reachable configuration.

(1) We associate a distinct place to every name $a \in N$ and to every process identifier A. The intended interpretation is that a token at place a corresponds to a message $\overline{a}$ while a token at place A means that the control of a thread is at A. Following this interpretation we determine the initial marking.

(2) To every equation we associate a set of transitions which are connected to the places as follows. If $A = a_1 \ldots + \cdots + a_n \ldots$ then we introduce n transitions $t_1, \ldots, t_n$ and for $k = 1, \ldots, n$ an edge from place A to transition t_k and an edge from place a_k to transition t_k. Moreover, if the continuation of a_k has the shape

$$(\Pi_{i\in I}\overline{a_i} \mid \Pi_{j\in J} A_j)$$

then we add an edge from transition t_k to place a_i for $i \in I$ and from transition t_k to place A_j for $j \in J$.

3.3 From systems without name generation to parameterless systems

We fix a system of parametric equations without name generation of the shape (4). For the sake of notational simplicity we assume that all channels have a recursive sort $s = Ch(s)$, and that all process identifiers depend on k parameters. Then:

- for every pair of channel names $a, b \in N$, we introduce a new channel name a_b of sort $Ch()$.
- for every equation of the shape (4) and for every vector of names $\vec{a'} \in N^k$

we produce an equation

$$A_{\vec{a'}} = \Sigma_{b' \in N}(\sigma(a)_{b'}.(\Pi_{i \in I}\overline{\sigma(c_i)_{\sigma(d_i)}} \mid \Pi_{j \in J} A_{j,\sigma(\vec{e}_j)}))\ .$$

where $\sigma \equiv [\vec{a'}/\vec{a}, b'/b]$.

To summarize, we transform a parametric system into a system without parameters but with external choice, and in turn, we transform the latter system into a Petri Net.

3.4 From reachability to coverability, and back

In terms of Petri Nets, the reachability problem we have formulated in definition 2.2 amounts to checking whether certain places, corresponding to a given process identifier, will contain a token. This is an instance of the *coverability* problem for which Lipton [10] has provided a $2^{O(\sqrt{n})}$ space lower bound and Rackoff [14] a $2^{O(n \log n)}$ space upper bound.

On the other hand, it is easy to see that the coverability problem for Petri Nets can be reduced to the reachability problem 2.2. Given a Petri Net, for every transition t taking, say, one token from places $a_1, \ldots, a_n$ and putting one token in places $b_1, \ldots, b_m$, we introduce the equations (we omit the parameters):

$$A_t = a_1.A_t^1 \quad A_t^1 = a_2.A_t^2 \ldots \quad A_t^{n-1} = a_n.(\overline{b}_1 \mid \cdots \mid \overline{b}_m \mid A_t)$$

Thus a transition of the Petri Net is now simulated by serialising the reading of the tokens. If we want to know, whether, say, the place a will ever contain a token we add the equation $A = a.B$. Then the initial configuration contains the process identifier A_t for every transition t, a number of messages corresponding to the initial marking, and the process identifier A. To determine whether the place a will contain a token it is then enough to check whether the initial configuration reaches one containing the process identifier B.

This reduction is polynomial and it shows that even without mobility and without name generation the reachability problem 2.2 we consider requires exponential space. We expect that our reachability problem could be generalized mimicking what has been done for Petri Nets [18]. On the other hand, the quest for decidability results on the equivalence problem (trace, bisimulation,...) is discouraged by the negative results known for Petri Nets [6,9].

4 The fragment with bounded control is undecidable

We say that a configuration has *bounded control* if there is a natural number that bounds the number of live threads running in parallel in any accessible configuration. One can imagine various syntactic conditions that imply this property and are efficiently checkable. To show our negative results, it will be enough to consider the fragment where the equation (2) is restricted to having

the shape:

$$A(\vec{a}) = a(\vec{b}).(\nu\vec{d})(\Pi_{i\in I}\overline{a}_i\vec{b}_i \mid A'(\vec{c}))$$
$$A(\vec{a}) = A_1(\vec{a}_1) \oplus A_2(\vec{a}_2)\ .$$

where $\oplus$ denotes the *internal choice*. This means that, up to internal choice, every control point has exactly one continuation and thus the control is basically *bounded* by the number of parallel threads present in the initial configuration.

Remark 4.1 It is well known that internal choice is *definable* from parallel composition and name generation. In our case, there is just a little twist to fit the shape of the normalised equations (2). Thus we replace the equation $A(\ldots) = A_1(\ldots) \oplus A_2(\ldots)$ by the equations

$$A(\ldots) \ = t.(\nu c)(A'_1(c,\ldots) \mid A'_2(c,\ldots) \mid \overline{c} \mid \overline{t})$$
$$A'_i(\ldots) = c.A_i(\ldots) \quad \text{for } i = 1,2$$

where t is a 'global' channel provided in the initial configuration with a message $\overline{t}$ (the t channel plays the role of the CCS τ action).

A similar trick applies if we want to define the internal choice of two messages $\overline{a_1} \oplus \overline{a_2}$. Then we introduce an identifier A and the equations:

$$A(\ldots) \ = t.(\nu c)(A'_1(c,\ldots) \mid A'_2(c,\ldots) \mid \overline{c} \mid \overline{t})$$
$$A'_i(\ldots) = c.\overline{a_i} \quad \text{for } i = 1,2\ \ .$$

Proposition 4.2 *The reachability problem for the fragment with bounded control is undecidable.*

Proof. The proof is loosely inspired by the encoding of the computation mechanism of Turing machines into a deduction system for Horn clauses without function symbols, also known as DATALOG. Readers familiar with the latter might find it inspiring to look at an 'existential' Horn clause $\forall\vec{x}\,(a(\vec{x}) \supset \exists\vec{y}\,b(\vec{x},\vec{y}))$ as a recursive process $A = a(\vec{x}).(\nu\vec{y})(\overline{b}(\vec{x},\vec{y}) \mid \overline{a}(\vec{x}) \mid A)$.

We now turn to the technical development. We simulate a 2-counter machine (see, *e.g.* [8]) and reduce the halting problem to the reachability problem 2.2. We assume that the 2-counter machine contains instructions of the form:

$$(1)\ q : C_k := C_k + 1;\ \ goto\ q'$$
$$(2)\ q : (C_k = 0) \to goto\ q', C_k := C_k - 1; goto\ q''$$

where C_1, C_2 denote the two counters. An instruction of type (1) increments the counter k and jumps to another point of the control. An instruction of type (2) tests whether the counter C_k is 0 and if it is the case it jumps to a control point q', otherwise it decrements the counter and jumps to control point q''.

A counter is represented as a stack of cells where the bottom cell contains 0 and all the others contain 1. Thus the value 2 is represented by the stack

011. For every state, we assume a channel q of sort $Ch()$. Moreover, for every counter C_k we assume channels

$$Top_k \text{ of sort } Ch(Ch(Ch(), Ch(), Ch())) \qquad \text{and}$$
$$Adj_k \text{ of sort } Ch(Ch(Ch(), Ch(), Ch()), Ch()) \ .$$

Every cell of the stack is assigned a distinct channel a of sort $Ch(Ch(), Ch(), Ch())$. We associate to every such channel three more distinct channels a_0, a_1, a_t and a message $\overline{a}(a_0, a_1, a_t)$. Moreover:

- If the channel a refers to the bottom cell then we introduce a message $\overline{a_0}$, and otherwise we introduce a message $\overline{a_1}$.
- If the channel a refers to the cell at the top of the stack we introduce a message $\overline{Top_k}a$.
- If the channels a and b refer to two adjacent cells (the first under the second) then we introduce a message $\overline{Adj}_k(a, b_t)$.

For instance, the stack 011 could be represented by the following messages:

$$\overline{a}(a_0, a_1, a_t) \mid \overline{a_0} \mid \overline{Adj}_k(a, b_t) \mid \text{(bottom cell)}$$
$$\overline{b}(b_0, b_1, b_t) \mid \overline{b_1} \mid \overline{Adj}_k(b, c_t) \mid \text{(second cell)}$$
$$\overline{c}(c_0, c_1, c_t) \mid \overline{c_1} \mid \overline{Top_k}c \qquad \text{(top cell)}$$

We now consider the problem of implementing on this data structure the 2-counter machine operations. An instruction of type (1) is translated as:

$$A = q.Top_k(a).(\nu a', a'_0, a'_1, a'_t)(\overline{q}' \mid \overline{Adj}_k(a, a'_t) \mid \overline{Top_k}(a') \mid \overline{a'}(a'_0, a'_1, a'_t) \mid \overline{a'_1} \mid A),$$

and an instruction of type (2) becomes:

$$\begin{aligned} A = {} & q.Top_k(a).a(a_0, a_1, a_t). \\ & (a_0.(\overline{q'} \mid \overline{Top_k}(a) \mid \overline{a}(a_0, a_1, a_t) \mid \overline{a_0} \mid A) \oplus \qquad \text{(if } C_k = 0\text{)} \\ & a_1.Adj_k(b, b_t).(\overline{a_t} \mid b_t.(\overline{q''} \mid \overline{Top_k}(b) \mid A))) \qquad \text{(if } C_k > 0\text{)} \ . \end{aligned}$$

Note that in the equations above we have omitted the parameters (which can be easily inferred) as well as the intermediate process identifiers. The case $(C_k > 0)$ reveals the role of the channel a_t: it is used to simulate via a communication an equality test between a_t and b_t so as to make sure that the received channel b corresponds to the cell preceding a's. □

4.1 Undecidability with generated values and conditional

The encoding above relies on channel mobility and moreover processes may input on received channel names. A frequently used extension of the π-calculus includes a *conditional* on name equality. To formalise this extension, we assume equations may have the shape:

$$A(\vec{a}) = [a = a']A'(\vec{a'}), A''(\vec{a''}) \tag{6}$$

with the expected meaning that we branch on A' if $a \equiv a'$ and on A'' otherwise.

Now if we allow a conditional on names of basic sort o then a simpler encoding is possible where all transmitted names have sort o. We assume additional channels $Cont_k$ to indicate the contents of a cell (values 0 or 1). The sorts are now as follows:

$$Top_k \text{ of sort } Ch(o),\ Adj_k \text{ of sort } Ch(o,o),\ \text{and } Cont_k \text{ of sort } Ch(o,o)\ .$$

An instruction of type (1) is translated as:

$$A = q.Top_k(a).(\nu a')(\overline{q'} \mid \overline{Adj_k}(a,a') \mid \overline{Cont_k}(a',1) \mid \overline{Top_k}(a') \mid A),$$

and an instruction of type (2) is translated as:

$$\begin{aligned} A = q.Top_k(a).Cont_k(a',v).[a' = a] \\ ([v = 0](\overline{q'} \mid \overline{Top_k}(a) \mid \overline{Cont_k}(a,0) \mid A), \\ Adj_k(a',a'').[a'' = a](\overline{q'} \mid \overline{Top_k}(a') \mid A))\ . \end{aligned}$$

5 The fragment without name mobility is undecidable

We consider the fragment where all names have sort $Ch()$, *i.e.*, no name mobility is allowed. Then the equation (2) is restricted to having the shape:

$$A(\vec{a}) = a.(\nu \vec{d})(\Pi_{i \in I} \overline{a}_i \mid \Pi_{j \in J} A_j(\vec{c}_j))\ . \tag{7}$$

In the absence of name mobility, generated names cannot be extruded and therefore name generation is essentially CCS *restriction*. Milner [11] shows that *synchronous* CCS with *restriction*, *relabelling*, and *external choice* is powerful enough to simulate a 2-counter machine. We will show that this simulation can be still carried on while dropping external choice and relabelling and using just *asynchronous* communication. Schematically, we replace (i) synchronous communication by asynchronous communication plus an acknowledgement, (ii) external choice by internal choice (of course, this is possible because we are just looking at a reachability property), and (iii) relabelling by parametric equations.

Proposition 5.1 *The reachability problem for the fragment with name generation and without name mobility is undecidable.*

Proof. Again we simulate a 2-counter machine in the form described in the proof of proposition 4.2 and reduce the halting problem to the reachability problem 2.2. The basic issue is to represent a stack. To this end we define the following system of equations (inspired by [11]). The channel i stands for increment, z for counter is zero, and d for decrement. Each of these channels comes with a corresponding 'acknowledgement' channel i^a, z^a, and d^a which

are kept implicit below.

$$
\begin{aligned}
B(i,z,d) &= B_i(i,z,d) \oplus B_z(i,z,d)\\
B_i(i,z,d) &= i.(\overline{i}^a \mid CB(i,z,d))\\
B_z(i,z,d) &= z.(\overline{z}^a \mid B(i,z,d))
\end{aligned}
$$

$$
\begin{aligned}
C(i,z,d,z',d') &= C_i(i,z,d,z',d') \oplus C_d(i,z,d,z',d')\\
C_i(i,z,d,z',d') &= i.(\overline{i}^a \mid CC(i,z,d,z',d'))\\
C_d(i,z,d,z',d') &= d.((\overline{d'} \oplus \overline{z'}) \mid D(i,z,d,z',d'))
\end{aligned}
$$

$$
\begin{aligned}
D(i,z,d,z',d') &= D_d(i,z,d,z',d') \oplus D_z(i,z,d,z',d')\\
D_d(i,z,d,z',d') &= (d'^a.(\overline{d}^a \mid C(i,z,d,z',d')))\\
D_z(i,z,d,z',d') &= (z'^a.(\overline{d}^a \mid B(i,z,d)))
\end{aligned}
$$

$$
\begin{aligned}
CB(i,z,d) &\equiv (\nu i'',z'',d'')(C(i,z,d,z'',d'') \mid B(i'',z'',d''))\\
CC(i,z,d,z',d') &\equiv (\nu i'',z'',d'')(C(i,z,d,z'',d'') \mid C(i'',z'',d'',z',d'))\ .
\end{aligned}
$$

A process C receives on i,d and sends on z',d'. A process B receives on i,z. When decrementing, a process C sends messages to its neighbour. The message goes on d if the neighbour is C and on z if the neighbour is B. Here is a schematic intuition of what happens:

$$
\begin{aligned}
&DCCCBB \to DDCCBB \to DDDCBB \to DDDDBB \to\\
&DDDBBB \to DDCBBB \to DCCBBB \to CCCBBB\ .
\end{aligned}
$$

The D is propagated towards the right till it meets B and when this happens it becomes B and shortcuts the last B.

Note the peculiar way in which we use the internal choice. If a 'server' can receive requests on two channels then it guesses non-deterministically on which channel the next message is coming. Symmetrically, a 'client' with two requests internally guesses which request is going to be served. If client and server guess consistently we obtain the desired behaviour. Otherwise client and server get stuck.

We translate a program of a 2-counter machine as a 'finite' control process that acts as a client for two counters' processes initialised by:

$$B(i_1,z_1,d_1) \mid B(i_2,z_2,d_2)\ .$$

The instructions of type (1) and (2) are simulated as follows:

$$(1)\ A_q = q.(\overline{i_k} \mid i^a_k.(\overline{q'} \mid A_q))\ ,$$

$$\begin{aligned}(2)\ A_q &= A^z_q \oplus A^d_q \\ A^z_q &= q.(\overline{z}_k \mid z^a_k.(\overline{q'} \mid A_q)) \\ A^d_q &= q.(\overline{d}_k \mid d^a_k.(\overline{q''} \mid A_q))\ .\end{aligned}$$

It is clear that by a suitable selection of internal choices we can simulate the behaviour of the 2-counter machine. On the other hand, suppose an attempted communication gets stuck because of wrong internal choices. This may happen (i) when the control sends a request to a counter, or (ii) when a decrement instruction propagates towards the right in a counter. In both cases the control is stuck. In the first case this is clear, in the second case this happens because the control waits for an acknowledgement which is delivered only after the propagation is completed. □

Remark 5.2 *In all the equations above, an input is followed, up to internal choice, by exactly one output. This implies that the number of messages present in a reachable configuration is* bounded.

6 The fragment without mobility and with bounded control is decidable

We consider the fragment where all names have the sort $Ch()$, and the equation (2) is restricted to the shape:

$$(8) \qquad A(\vec{a}) = a.(\nu\vec{d})(\Pi_{i\in I}\overline{a_i} \mid B(\vec{b}))\ .$$

For the sake of simplicity, we assume that all the equations in a given system depend on k parameters. We note that in systems without name mobility and with bounded control there is a bound on the number of 'live' names appearing in any reachable configuration. Indeed, the only form of name transmission allowed in these systems is *via* the recursion parameters: once a name disappears from the recursion parameters, no input can ever be performed on that name again. Therefore, without loss of generality we suppose that in the equation (8) above $\{\vec{d}\} \subseteq \{\vec{b}\}$.

The basic idea is to generalise the reduction to Petri Nets presented in section 3 and to replace name generation by the reusing of 'dead' names. We will begin by transforming the system into an equivalent parameterless system of equations *with reset* (and without name generation), which in turn we transform into a Petri Net with reset arcs. The latter can be reduced to a standard Petri Net, provided that the number of tokens in resetable places is bounded (in general Petri Nets with reset arcs are undecidable).

In the following, a parameterless system of equations with reset is a variant of the parameterless system presented in section 3.1. In such a system, the equations have the shape

$$A = a.\mathrm{reset}\,\vec{d}.(\Pi_{i\in I}\overline{a_i} \mid B)\ , \tag{9}$$

and the semantics of the *reset* operator is to erase all messages sent on names belonging to its argument.

6.1 Lifetime analysis of names

In order to reduce a Petri Net with reset arcs to a standard Petri Net, we need a bound on the number of tokens in any resetable place. This leads us to distinguishing two kinds of names in the original system: *persistent* names, for which there is no bound, but which never need to be reset, and *temporary* names. We will give a bound on the number of messages sent on any temporary name.

To this end, we introduce the *parameter flow graph* of the system, which is defined as follows.

Definition 6.1 *The* parameter flow graph *of a system $\mathcal{E}$ is a directed graph $\mathcal{G} = (\mathcal{L}, \mapsto)$, where:*

- *The set of nodes $\mathcal{L}$ is given by the* parameter positions $\{A^i \mid A$ *identifier in* $\mathcal{E}$ *and* $i \in [1,k]\}$.
- $A^i \mapsto B^j$ *is an edge of the graph if if the equation associated to A in the system $\mathcal{E}$ is*

$$A(\vec{a}) = a.(\nu\vec{d})(\ldots \mid B(\vec{b}))\ ,$$

and the i-th component of $\vec{a}$ is equal to the j-th component of $\vec{b}$.

Positions leading to a cycle in $\mathcal{G}$ will be referred to as *persistent positions*, while the others will be called *temporary positions*. Accordingly, when a name occurs in $A(\vec{a})$ we will call that name persistent if it is used in at least one persistent position, and temporary if it is used only in temporary positions.

Note the peculiar structure of $\mathcal{G}$: if we consider the class of positions associated to one process identifier, all edges from vertices in this class lead to vertices in a unique class, due to our syntactic definition of finite control. Also, since all names in $\{\vec{a}\}$ are distinct, we cannot have, for $i \neq j$, $A^i \mapsto B^l$ and $A^j \mapsto B^l$. It follows from these observations that the set of vertices reachable from a temporary position is a finite tree. If e is the number of equations in $\mathcal{E}$ then the size of the tree is bounded by $e \cdot k$ which is the number of parameter positions. Moreover, if m is the maximum number of outputs on any parameter in any equation of $\mathcal{E}$, then the number of outputs performed on *any* temporary name is bounded by $e \cdot k \cdot m$, which is polynomial in the size of $\mathcal{E}$.

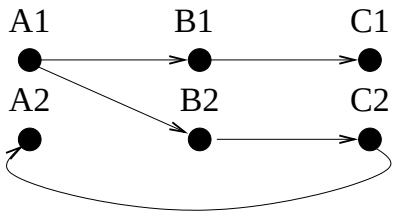


Fig. 1. The parameter flow graph for $\mathcal{E}$

Example 6.2 *Let us consider the system $\mathcal{E}$ defined by the equations*

$$\begin{aligned} A(a,b) &= b.(\overline{a} \mid B(a,a)) \\ B(a,b) &= a.C(a,b) \\ C(a,b) &= b.(\nu c)(\overline{c} \mid A(c,a)) \ , \end{aligned}$$

and the initial configuration

$$P \equiv \overline{a} \mid \overline{a} \mid \overline{b} \mid A(a,b) \ .$$

In this system, all newly generated names are temporary names used in position A^1. Since the tree rooted in A^1 has 6 nodes, and no equation in $\mathcal{E}$ performs more than 1 output on any of its parameters, we can take 6 as a bound on the number of messages sent on any temporary name.

6.2 From systems without mobility and with bounded control to parameterless systems with reset

We fix a system $\mathcal{E}$ of equations of the type (8), and an initial configuration P, which does not contain any generated names. Let N_0 be the set of names free in P, and n the number of process identifiers in P. Without loss of generality, we may suppose that every process identifier in $\mathcal{E}$ relates to a *unique* thread of the initial configuration (if process identifiers are shared among different threads then we can always rename them so as to satisfy this condition).

We will construct a system $\mathcal{E}'$ of equations of the shape (9) and show that the reachability problem for $\mathcal{E}$ and P reduces to a finite number of reachability problems for $\mathcal{E}'$ and a suitable initial configuration P'.

We assume, for every $j \in [1,n]$, pairwise disjoint sets P_j and T_j, of respective cardinalities k and $2k$, which will represent the j-th thread's private name space (P_j is used for persistent names and T_j for the temporary ones). The parameterless system $\mathcal{E}'$ will be defined over the name space $N = N_0 \cup (\cup_{j\in[1,n]} P_j \cup T_j)$.

Definition 6.3 The vector of names $(a_1, \ldots, a_k)$ is compatible with the process identifier A of the j^{th} thread (written $(a_1, \ldots, a_k) \downarrow A, j$) if for all $a \in$

$\{a_1, \ldots, a_k\}$

$$a \in \begin{cases} N_0 \cup P_j \text{ if } \exists\, i \ \ (a_i = a \text{ and } A^i \text{ is a persistent position}) \\ N_0 \cup T_j \text{ otherwise.} \end{cases}$$

Next we define the system $\mathcal{E}'$ associated to $(P, \mathcal{E})$.

Definition 6.4 Fix an equation of the shape (8) in $\mathcal{E}$ relating, say, to the thread j. Then:

(i) for every $\vec{a'}$ such that $\vec{a'} \downarrow A, j$,

(ii) for every *injective* substitution $[\vec{d'}/\vec{d}]$ such that $\{\vec{d'}\} \subseteq P_j \cup T_j$, $\{\vec{d'}\} \cap \{\vec{a'}\} = \emptyset$, $\sigma = [\vec{a'}/\vec{a}, \vec{d'}/\vec{d}]$, and $\sigma\vec{b} \downarrow B, j$

we introduce an equation

$$A_{\vec{a'}} = \sigma(a).\text{reset}\, \vec{r}.(\Pi_{i \in I'} \sigma a_i \mid B_{\sigma\vec{b}})$$

where $\{\vec{r}\} = T_j \backslash \{\sigma\vec{b}\}$ and $I' = \{i \in I \mid \sigma a_i \notin \{\vec{r}\}\}$.

Roughly, we consider all compatible instances $\vec{a'}$ of a process identifier A of a thread j, we replace the generated names $\vec{d}$ by unused names in $P_j \cup T_j$ (a simple cardinality argument show that they exist), and we reset all the channels on temporary names that are not used in the continuation.

Next, we introduce a binary relation $\mathcal{R}$ relating configurations and parameterless configurations.

Definition 6.5 Let $Q \equiv (\nu\vec{d_0})(\nu\vec{d})(\Pi_{i \in I} \overline{a}_i \mid \Pi_{j \in [1,n]} A_j(\vec{a}_j))$ be a configuration where we assume that:

(i) $\{\vec{d_0}\} \cap (\bigcup_{j \in [1,n]} \{\vec{a}_j\}) = \emptyset$,

(ii) the identifier A_j relates to the j^{th} thread, and

(iii) if $d \in \{\vec{d}\}$ then d occurs in *exactly one* set of parameters $\{\vec{a_j}\}$.

Then $Q\ \mathcal{R}\ (\Pi_{i \in I'} \sigma a_i \mid \Pi_{j \in [1,n]} A_{j,\sigma\vec{a}_j})$ whenever:

- $I' = \{i \in I \mid a_i \in N_0 \cup (\bigcup_{j \in [1,n]} \{\vec{a}_j\})\}$ and
- σ is an *injective* substitution from $\vec{d}$ to $\bigcup_{j \in [1..n]} (T_j \cup P_j)$ such that $\sigma\vec{a}_j \downarrow A_j, j$, for $j \in [1, n]$.

Here we follow the same approach as in the previous definition 6.4: we replace the generated names occurring in the parameters of exactly one process identifier by compatible names in the set $T_j \cup P_j$, while removing useless restrictions and messages.

Given an initial configuration P, we can easily compute a P' such that $P\ \mathcal{R}\ P'$. Then we have to check that the relation $\mathcal{R}$ is sufficiently general to keep the two configurations in lockstep.

Lemma 6.6 *If $Q\ \mathcal{R}\ Q'$, we have:*

- *if $Q \to R$ then there is R' such that $R\ \mathcal{R}\ R'$ and $Q' \to R'$.*

- *if* $Q' \to R'$ *then there is* R *such that* $R \mathcal{R} R'$ *and* $Q \to R$.

The proof of this lemma is a simple, although laborious, manipulation of definitions 2.1, 6.4, and 6.5. We can then reduce the reachability problem for $(P, \mathcal{E})$ to a finite number of reachability problems for $(P', \mathcal{E}')$.

Proposition 6.7 *The reachability of the process identifier* A *in* $(P, \mathcal{E})$ *is equivalent to the reachability of one of the (finitely many) parameterless identifiers* $A_{\vec{a}'}$ *in* $(P', \mathcal{E}')$.

Proof. We apply lemma 6.6 inductively on the length of the considered reduction chain and exploit the definition of the relation $\mathcal{R}$. □

6.3 From parameterless systems with reset to Petri Nets with reset arcs

In this section, we show how to extend the reduction from parameterless systems to Petri Nets described in section 3.2, to a reduction from parameterless systems with *reset* to Petri Nets with *reset arcs*.

We suppose given a parameterless system of equations with reset $\mathcal{E}'$, and an initial configuration P' without name generation. Let N be the finite name space over which the system is defined (note that this may be strictly larger than the collection of names free in P').

Like in section 3.2, we build a Petri Net that has one place for each parameterless process identifier in $\mathcal{E}'$, and one place for each name in N (remember that we do not consider mobility). The intended interpretation is still that a token in place a corresponds to a message $\overline{a}$, while a token in place A corresponds to the presence of a parameterless process identifier A in the current configuration.

The transitions are set as in section 3.2, except that we no longer have to care for external choice (*i.e.* there is only one transition per equation), and that if the equation associated to A is $A = \ldots \text{reset}\, \vec{r} \ldots$, then for each $a \in \{\vec{r}\}$ we add a reset arc going from transition t_A to the place a.

Note that, thanks to the analysis performed in subsection 6.1, we can guarantee that all the places pointed to by reset arcs are bounded.

Proposition 6.8 *The reachability of* A *in* $(P', \mathcal{E}')$ *is equivalent to the coverability of place* A *with* 1 *token in the Petri Net with reset arcs described above.*

6.4 From Petri Nets with reset arcs to Petri Nets

Finally, we recall how to simulate a Petri Net with reset arcs $\mathcal{N}$ with a standard Petri Net $\mathcal{N}_0$, provided that all resetable places are bounded (this is a standard result for Petri Nets).

For each resetable place p, we add a complementary place p'. If b is the bound on the number of tokens in place p, in all reachable markings M we will maintain the invariant $M(p) + M(p') = b$.

To this end, we modify the existing transitions so as to add as many outgoing arcs to p' as the number of incoming arcs from p, and as many incoming arcs from p' as the number of outgoing arcs to p.

Then, for any transition t that points a reset arc at p, we replace t by the transitions $t_0, \ldots, t_b$, where, t_i is connected to the places of the net by the same arcs as t, plus an arc of weight i incoming from p, an arc of weight $b - i$ incoming from p', and an arc of weight b outgoing to p'.

Proposition 6.9 *A marking M is reachable in $\mathcal{N}$ if and only if the marking M' is reachable in $\mathcal{N}_0$, where*

- *for any place p of $\mathcal{N}$, $M'(p) = M(p)$,*
- *and for any resetable place p of $\mathcal{N}$, $M'(p') = b - M(p)$.*

To summarize, given a system in the fragment without mobility and with bounded control, by composing the three reductions presented above, we reduce the reachability problem for that system to a finite number of coverability problems for a standard Petri Net.

Theorem 6.10 *The reachability problem for the fragment without mobility and with bounded control is decidable.*

Our decision result could be extended from equations of the shape (8) to equations of the following shape:

$$A(\vec{a}) = a.(\nu\vec{c})(\Pi_{i\in I}\overline{a}_i \mid B(\vec{b}) \mid \Pi_{j\in J}A_j) \tag{10}$$

where A_j are *parameterless* process identifiers that refer to parameterless equations of the shape (5) whose free names do not intersect the generated names $\vec{c}$.

An interesting open problem, concerns the decidability of the fragment with name generation, bounded control, and *weak* forms of name mobility where, *e.g.*, a process cannot receive on received names.

References

[1] R. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations for the asynchronous π-calculus. *Theoretical Computer Science*, 195:291–324, 1998.

[2] G. Boudol. Asynchrony and the π-calculus. Technical report, *RR 1702, INRIA, Sophia-Antipolis*, 1992.

[3] M. Dam. Model checking mobile processes. *Information and Computation*, 1996. Preliminary version appeared in Proc. Concur'93.

[4] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Proc. CONCUR 96, Springer Lect. Notes in Comp. Sci. 1119*, 1996.

[5] U. Golz and A. Mycroft. On the relationship of CCS and Petri Nets. *Proc. ICALP84, Springer Lect. Notes in Comp. Sci. 172:196–208*, 1984.

[6] M. Hack. *Decidability questions for Petri Nets.* Garland publishing Co., 1979.

[7] K. Honda and M. Tokoro. An object calculus for asynchronous communication. *Proc. ECOOP 91, Geneve, Springer Lect. Notes in Comp. Sci. 612*, pages 133–147, 1991.

[8] J. Hopcroft and J. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, 1979.

[9] P. Jancar. Undecidability of bisimilarity for Petri Nets and related problems. *Theoretical Computer Science*, 148:281–301, 1995.

[10] R. Lipton. The reachability problem requires exponential space. Technical Report TR 66, Yale University, 1976.

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

[12] U. Montanari and M. Pistore. Checking bisimilarity for finitary π-calculus. In *CONCUR '95, Springer Lect. Notes in Comp. Sci. 962*, 1995.

[13] B. Pierce and D. Turner. Pict: a programming language based on the π-calculus. University of Cambridge, 1996.

[14] C. Rackoff. The covering and boundedness problem for vector addition systems. *Theoretical Computer Science*, 6:223–231, 1978.

[15] C. Reutenauer. *Aspects mathématiques des réseaux de Petri.* Masson Editeur, 1988. Also available in english: The mathematics of Petri Nets, Prentice-Hall.

[16] G. Sutre. *Abstraction et accéleration de systèmes infinis.* PhD thesis, ENS Cachan, 2000.

[17] V. Vasconcelos and R. Bastos. Core-TyCO, the language definition, version 0.1. Technical report TD98-3, University of Lisbon, 1998.

[18] H. Yen. A unified approach for deciding the existence of certain Petri Nets paths. *Information and Control*, 96(1):119–137, 1992.

Electronic Notes in Theoretical Computer Science 47 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume47.html pp. 1 – 18

Broadcast Calculus Interpreted in CCS upto Bisimulation

K. V. S. Prasad [1]

Department of Computer Science
Chalmers University of Technology
412 96 Gothenburg, Sweden

Abstract

A function M is given that takes any process p in the calculus of broadcasting systems CBS and returns a CCS process $M\,(p)$ with special actions $\{\mathsf{hear?}, \mathsf{heard!}, \mathsf{say?}, \mathsf{said!}\}$ such that a broadcast of w by p is matched by the sequence $\mathsf{say?}\,\tau^*\mathsf{said}\,(w)$ by $M\,(p)$ and a reception of v by $\mathsf{hear}\,(v)\,?\,\tau^*\mathsf{heard!}$. It is shown that $p \sim M\,(p)$, where $\sim$ is a bisimulation equivalence using the above matches, and that $M\,(p)$ has no CCS behaviour not covered by $\sim$. Thus the abstraction of a globally synchronising broadcast can be implemented by sequences of local synchronisations. The criteria of correctness are unusual, and arguably stronger than the usual one of preserving equivalences. Since $\sim$ has meaning only with the above matches, it is a matter of dicussion what the result says about Holmer's (CONCUR'93) conjecture, partially proved by Ene and Muntean (FCT'99), that CCS cannot interpret CBS upto preservation of equivalence.

1 Background

Broadcast communication as in CBS, the calculus of broadcasting systems [14], differs obviously from handshake communication as in CCS [11] in that the former is one-to-many while the latter is one-to-one. But a perhaps more important difference shows up even when just one process observes just one other: actions in CCS are either visible and interactive or invisible and autonomous, while CBS has autonomous output actions. So to get the result of a CCS program, the user has to interact with it, for example after the computation, which itself is silent. In contrast, the user can eavesdrop on a CBS computation while it is running. This gives a very simple way to run a CBS program, expressed using a set of coordination primitives on top of any sequential language, which turn out to be remarkably easy to implement.

[1] Email: prasad@cs.chalmers.se

CBS has been used for experiments in parallel programming and in courses on protocols, etc., and has several implementations.

This paper presents a particular CCS implementation of CBS, fashioned after the top-down sequential randomised interpreter ("TSR") [13,14] for CBS presented in Section 2. The paper is self-contained, and no prior knowledge of CBS is needed, but readers unfamiliar with CCS or process calculus in general are referred to [11]. An informal overview of CCS and CBS is given in Section 2, and the formal syntax and semantics of both calculi in Section 3.

1.1 TSR in CCS

TSR is suitable for this paper because it is very simple. It has been formally shown to be correct (some techniques are reported in [6]) and the proof used to formally prove a sorter correct [2]. Similar uses of CBS are reported in [5]. TSR has also been used for experiments with the grain of parallelism [14], done with a quasi-parallel machine [16].

TSR is cast in CCS via a translation function M that takes a process p in CBS and returns a process $M(p)$ in CCS. The behaviour of this process mimics that of TSR applied to p in the following sense: the call-return sequences in TSR that correspond to CBS actions are encoded as simple CCS protocols. Then $M(p)$ is shown to behave like p when seen through these protocols.

1.2 Motivation: Global synchronisation via local synchronisations

The original motivation for the present work was not to relate CBS to CCS, but to clarify an abstraction that has often been seen as problematic: that the CBS model assumes global synchronisation on every broadcast. And yet the intuition from sequential implementations of CBS, particularly lazy and quasi-parallel ones, is that this global synchronisation does not have to be taken literally, but can in fact be an abstraction from sequences of local synchronisations. The CBS model also assumes that even local hidden broadcasts synchronise with the environment, while in the implementation the environment is undisturbed.

To capture and validate these intuitions about implementations of CBS processes, the author recast them as abstract machines — originally in a low level notation with states and unlabelled transitions. This paper uses CCS notation instead. In so doing, it runs into another thread of research.

1.3 CBS in CCS

Given different calculi of communicating systems, a very natural question that arises is whether any one can be expressed in terms of any other.

Holmer (1993) [8] gives a translation function S from CBS to SCCS [10] such that if $p_1, p_2 \in$ CBS then $p_1 \sim p_2 \iff S(p_1) \sim S(p_2)$, where $\sim$ is strong bisimulation equivalence. The existence of this translation is unsurprising, as

(a variant of) SCCS is known to be universal among process calculi [3], but several facts about it are interesting. First, note that the result proved is what appears to be the standard one in translating from one calculus to another: preservation of equivalences (abbreviated "standard result" below). Next, corresponding to CBS broadcast actions labelled $w!$ and reception actions labelled $v?$, the SCCS translation uses actions with precisely these labels. It also uses many others, but only these survive through to the bisimulation. The question does not arise how SCCS actions be combined into a broadcast; one SCCS action is merely assigned to represent the latter. There is no limitation on how SCCS actions may be combined, and the translation exploits the freedom of interpretation this affords.

Holmer (1993) also conjectures that it is not possible to translate CBS to CCS. His reason is that a broadcast has to be implemented by different numbers of handshakes in different contexts, depending on the number of receivers. This makes it hard to see how $M(p)$ can be independent of the context of p. Holmer does not say in what sense the sequence of handshakes would be required to be equivalent to a broadcast, were a translation possible, but it seems safe to assume he would require a standard result w.r.t. some reasonable equivalences.

Ene and Muntean (1999) [4] prove a version Holmer's conjecture for the case that M is "uniform", a condition examined in Section 5, and fulfilled by neither the translation in this paper nor by Holmer's SCCS translation. They show that there is no uniform encoding of CBS into CCS that preserves a "reasonable semantics". What is relevant here is not the exact nature of their "reasonable semantics", but their remark that they would like the encodings of two terms equivalent under a certain semantics in the first calculus to be equivalent under a related semantics in the second. That is, they require a standard result, but also want the two equivalences be related.

The result in this paper relates the behaviours of p and $M(p)$ directly, and not via equivalences over the two calculi. It is thus non-standard. Section 5 discusses this departure from tradition.

2 Informal Overview

First, an informal summary of the operational semantics of both calculi. [11] or [14] must be consulted for more explanation.

2.1 CCS

The notation $m \xrightarrow{\mathsf{a}(w)?} m'$ says that the process m can receive a value w on the channel a and become the process m'. Similarly, $m \xrightarrow{\mathsf{a}(w)!} m'$ says that m can send w along a and become m'. The arrow notations $\xrightarrow{\mathsf{a}(w)?}$ or $\xrightarrow{\mathsf{a}(w)!}$ need no further decoration to say they are CCS actions as opposed to CBS ones, because the latter, described below, use no channel names. Further down,

CBS experiments are defined over CCS processes. Context should distinguish CCS processes from CBS ones, but as an aid, CCS processes are usually called m (with primes and subscripts) while CBS processes are called p.

The inference rule below for parallel composition $\mid$ says how matching offers of reception and transmission can be combined.

$$\frac{m_1 \xrightarrow{\mathsf{a}(w)?} m_1' \quad m_2 \xrightarrow{\mathsf{a}(w)!} m_2'}{m_1 \mid m_2 \xrightarrow{\tau} m_1' \mid m_2'}$$

The resulting communication action τ is both autonomous and unobservable, and is the only action with either of these properties.

Finally, the type of the values w above is determined uniquely by the channel. Types are needed in this paper only to describe the CBS scoping operator. They are usually clear from the context and dropped.

2.2 CBS

Each CBS process communicates along a lone channel implicitly associated with the process. Processes in parallel share this implicit channel, and synchronise on every broadcast on it. Processes are input-enabled, that is, they are always prepared to receive any broadcast value. It helps to think of broadcasting as akin to speaking and receiving to hearing. Speech is autonomous, while hearing only happens when someone else speaks. Processes speak one at a time, contention between speakers being resolved non-deterministically.

A CBS channel carries data of only one type, say α. Let $w \in \alpha_\tau = \alpha \cup \{\tau\}$, where $\tau \notin \alpha$ is a special value standing for a hidden broadcast. Then $p \xrightarrow{w?} p'$ means p can hear w and become p' as a result, and $p \xrightarrow{w!} p'$ means p can say w and become p' as a result. Note that the implicit lone channel is not named. The characteristic communication rule for CBS is

$$\frac{p_1 \xrightarrow{w?} p_1' \quad p_2 \xrightarrow{w!} p_2'}{p_1 \mid p_2 \xrightarrow{w!} p_1' \mid p_2'}$$

The notation $p \xrightarrow{w!}\!\!\!\!\!/\;$ is used to mean $\not\exists p'.\, p \xrightarrow{w!} p'$, and $p \xrightarrow{w?}\!\!\!\!\!/\;$ to mean $\not\exists p'.\, p \xrightarrow{w?} p'$. It is convenient to define, for any CBS process p, a predicate

$$p \xrightarrow{\delta!} \iff p \xrightarrow{\tau!}\!\!\!\!\!/\; \text{ and } \forall v.\, p \xrightarrow{v!}\!\!\!\!\!/\;$$

and to write $p \xrightarrow{\delta!}$ as $p \xrightarrow{\delta!} p$ so that "p has nothing to say" is written as a pseudo speech action. There is no corresponding receive action: $\forall p.\ p \xrightarrow{\delta?}\!\!\!\!\!/\;$. To take advantage of this convention, w is allowed to range over $\alpha_{\tau\delta} = \alpha \cup \{\tau\} \cup \{\delta\}$, where $\delta \notin \alpha$ is a special value standing for "no broadcast".

Except that $\delta!$ never causes a change of state, it is rather like a clock tick in a timed CBS [15] with maximal progress: it is what happens when no has anything to say.

The following properties will hold by induction for every CBS process p.

(i) $p \xrightarrow{\tau?} p' \Longrightarrow p \equiv p'$

(ii) $\forall v.\, \exists!\, p'.\, p \xrightarrow{v?} p'$

The first is obviously in keeping with the design of the communication model, but there are versions of CBS without the second property, which says that CBS processes are input-deterministic. This simplifying property holds here because a restricted choice operator is used instead of a general + as in CCS.

Hiding, restriction and translation are provided as follows. Processes whose implicit channel carries data of type α are assigned type $\mathrm{Proc}\,(\alpha)$ and can be thought of as speaking the language α. If $f\colon \beta \to \alpha_\tau$, and $g\colon \alpha \to \beta_\tau$, then $\mathsf{T}_g^f p$ is a process that speaks α while p speaks β. The functions f and g translate broadcast values from and to the subsystem, and translation to τ means the value is hidden or restricted, respectively. The functions are extended by setting $f\,(\tau) = g\,(\tau) = \tau$.

$$\frac{p \xrightarrow{g(w)?} p'}{\mathsf{T}_g^f p \xrightarrow{w?} \mathsf{T}_g^f p'} \qquad \frac{p \xrightarrow{w!} p'}{\mathsf{T}_g^f p \xrightarrow{f(w)!} \mathsf{T}_g^f p'}$$

2.3 Interpreting CBS in CCS

This paper finds an interpretation of CBS in CCS. This development is in three steps.

2.3.1 TSR

The starting point is TSR. It represents CBS processes as members of an abstract data type, and provides a function

$$\mathsf{say}\colon \mathrm{Proc}\,(\alpha) \to \mathrm{Int} \to \langle \alpha_{\tau\delta}, \mathrm{Proc}\,(\alpha)\rangle$$

such that

$$\exists r.\, \mathsf{say}\ p\ r = \langle w, p'\rangle \iff p \xrightarrow{w!} p'$$

Here r is a random seed, needed if $p \equiv p_1 \mid p_2$ is a parallel composition. TSR uses r to pick one of p_1, p_2 as speaker. Say it picks p_1. Let $\mathsf{say}\ p_1\ r_1 = \langle w, p_1'\rangle$, where r_1 is a new random number. Then there are three cases.

If $w \in \alpha$, then the other component is made to hear w. That is, $\mathsf{say}\ p\ r = \langle w, (p_1' \mid \mathsf{hear}\ p_2\ w)\rangle$, where

$$\mathsf{hear}\colon \mathrm{Proc}\,(\alpha) \to \alpha \to \mathrm{Proc}\,(\alpha)$$

such that

$$\mathsf{hear}\ p\ v = p' \iff p \xrightarrow{v?} p'$$

A function suffices for this version of CBS because it is input-enabled and -deterministic.

If $w = \tau$, that is, p_1 makes a hidden broadcast, then $\mathsf{say}\ p\ r = \langle w, p_1' \mid p_2 \rangle$. That is, the hear invocation is skipped.

If $w = \delta$, that is, p_1 has nothing to say, then say is applied to p_2 and three similar cases arise. If p_2 too has nothing to say, then p has nothing to say.

The other cases, where the outermost constructor of p is not $|$, are easier. If $\mathsf{say}\ p\ r_1 = \langle w_1, p_1 \rangle$, $\mathsf{say}\ p\ r_2 = \langle w_2, p_2 \rangle$, ..., $\mathsf{say}\ p\ r_i = \langle w_i, p_i \rangle$, then p is said to produce the trace w_1, w_2, ..., w_i. The trace ends at w_i if $w_i = \delta$.

TSR has been formally shown [2,6] to be sound and complete w.r.t. the operational semantics of CBS. That is, every execution sequence produced by TSR can be justified by the semantics of CBS, and every trace derivable from the semantics is produced by a run of TSR for some random seed.

2.3.2 TSR in CCS

Next, for any CBS process $p \in \mathrm{Proc}\,(\alpha)$, the say and hear functions partially applied to p are provided as commands to a CCS process $M\,(p)$, which is the CCS encoding of p.

The CCS equivalent of evaluating $\mathsf{say}\ p\ r$ is to send $M\,(p)$ the message say. No random seed is needed, since $M\,(p)$ can be non-deterministic. If $p \xrightarrow{w!} p'$, $M\,(p)$ responds $\mathsf{said}\,(w)$ and evolves to $M\,(p')$. Since $w \in \alpha_{\tau\delta}$, $M\,(p)$ will always have some response, even if $w = \tau$ or $w = \delta$.

The CCS equivalent of evaluating $\mathsf{hear}\ p\ v$ is to send $M\,(p)$ the message $\mathsf{hear}\,(v)$. If $p \xrightarrow{v?} p'$, $M\,(p)$ responds $\mathsf{heard!}$ and evolves to $M\,(p')$.

The function M is defined in Table 3.

2.3.3 Relating p and M (p)

Lastly, the behaviour of p is related to that of $M\,(p)$ by defining CBS experiments over CCS processes.

Definition 2.1 [CBS experiments over CCS processes] Let m, m' be arbitrary CCS processes. The following *CBS experiments* are defined over CCS processes, for $w \in \alpha_{\tau\delta}$ and $v \in \alpha$.

$$m \xrightarrow{w!} m' \iff m \xrightarrow{\mathsf{say}_\alpha ?} \xrightarrow{\tau}{}^{*} \xrightarrow{\mathsf{said}_\alpha(w)!} m'$$

$$m \xrightarrow{v?} m' \iff m \xrightarrow{\mathsf{hear}_\alpha(v)?} \xrightarrow{\tau}{}^{*} \xrightarrow{\mathsf{heard}_\alpha!} m'$$

$$m \xrightarrow{\tau?} m$$

The subscript α is usually clear from the context and is dropped. "CBS experiment" is usually abbreviated "experiment". Bisimulations over these experiments are called *CBS bisimulations*, usually abbreviated just "bisimulations".

CCS processes of the form $m \equiv M\,(p)$, for some CBS process p, are said to be in *CBS form*. Definition 2.1 applies to any CCS processes m and m',

but of course only those in CBS form are explicitly constructed to have CBS experiments defined over them. A CCS process in CBS form will evolve by CCS moves through processes not of this form, but will eventually always offer complete CBS experiments, as shown by Proposition 4.4. That the result of such a CBS experiment is another CBS form is shown in passing by Proposition 4.5, which establishes a CBS bisimulation between p and $M(p)$ for any CBS process p.

The last clause in the definition above saves the trouble of explicitly providing each $M(p)$ with a sum branch $\mathsf{hear}(\tau)?\,\mathsf{heard}!\,M(p)$. As the reader can guess, the CCS interpreter will optimise by not delivering τ's as messages to be heard.

2.4 Relating one kind of transition to another

Now that a CBS process p and its CCS interpretation $M(p)$ have the same kinds of transitions, it is possible to look for bisimulation or observational equivalence.

Most work in process calculus relates systems that use the same communication model. For example, weak bisimulation is usually formulated as relating single transitions $\xrightarrow{\mu}$ to sequences $\xrightarrow{\tau}{}^{*}\ \xrightarrow{\mu}\ \xrightarrow{\tau}{}^{*}$. Alternatively, the latter sequence can be formulated in terms of a single derived "experiment" transition, also labelled with μ. Weak bisimulation is then just ordinary bisimulation over such experiments. This is the style this paper uses, but in contrast to standard formulations of bisimulation relates a communication of one kind, broadcast, to one of a completely different kind, handshake.

Note that the CBS experiments above put external actions on either side of τs, in contrast to the weak bisimulation experiments.

3 Formal Syntax and Semantics of CBS and CCS

3.1 CBS

This paper is restricted to finite CBS processes. The key idea, how to interpret multi-way communication pairwise, has nothing to do with recursive CBS processes, and the proofs need induction over the structure of CBS processes. If recursive CBS processes are included, "induction over depth of guardedness" would be needed instead, as in [14]. This isn't hard, but seems an unnecessary distraction in this paper, so recursive CBS processes have been dropped for expository reasons. Thus "CBS" in this paper is a subset of that in [14].

Much of the needed notation has already been introduced. Let $x{:}\,\alpha$ be a variable, and $u, v{:}\,\alpha$ be expressions. Let b be a boolean expression. Let $p_\beta{:}\,\mathrm{Proc}(\beta)$, $w_\beta{:}\,\beta_{\tau\delta}$, $f{:}\,\beta \to \alpha_\tau$, and $g{:}\,\alpha \to \beta_\tau$. Note that β is an existential type. Then the processes speaking α are given by

$$p ::= \mathbf{0} \;\Big|\; x?\,p \;\Big|\; v!\,p \;\Big|\; \langle x, p\rangle \& \langle v, p\rangle \;\Big|\; p \mid p \;\Big|\; b \to p, p \;\Big|\; \mathsf{T}_g^f p_\beta$$

$$p \xrightarrow{\tau?} p$$

$$\mathbf{0} \xrightarrow{v?} \mathbf{0}$$

$$x?\,p \xrightarrow{v?} p[v/x]$$

$$u!\,p \xrightarrow{v?} u!\,p \qquad\qquad u!\,p \xrightarrow{u!} p$$

$$\langle x, p\rangle \& \langle u, q\rangle \xrightarrow{v?} p[v/x] \qquad \langle x, p\rangle \& \langle u, q\rangle \xrightarrow{u!} q$$

$$\frac{p \xrightarrow{w?} p' \quad q \xrightarrow{w?} q'}{p \mid q \xrightarrow{w?} p' \mid q'} \qquad \frac{p \xrightarrow{w!} p' \quad q \xrightarrow{w?} q'}{p \mid q \xrightarrow{w!} p' \mid q'}$$

$$\frac{p \xrightarrow{\mathsf{T}_\downarrow(w)?} p'}{\mathsf{T}p \xrightarrow{w?} \mathsf{T}p'} \qquad \frac{p \xrightarrow{w!} p'}{\mathsf{T}p \xrightarrow{\mathsf{T}^\uparrow(w)!} \mathsf{T}p'}$$

$$\frac{p \xrightarrow{v?} p'}{(\mathsf{tt} \to p, q) \xrightarrow{v?} p'} \qquad \frac{p \xrightarrow{w!} p'}{(\mathsf{tt} \to p, q) \xrightarrow{w!} p'}$$

Metavariables: Here v ranges over α, and w over α_τ.
There are symmetric rules for $p \mid q$ where q speaks, and for $(\mathsf{ff} \to p, q)$.
$p \xrightarrow{\delta!} p \iff p \xrightarrow{\tau!}\!\!\!\!\!\!/\;$ and $\forall v.\, p \xrightarrow{v!}\!\!\!\!\!\!/\;$

Table 1
Operational rules for CBS

The notation $b \to p, q$ means "if b then p else q".

The subscripts β are usually dropped. In processes of the form $\mathsf{T}^f_g p$, it is convenient if f is called $\mathsf{T}^\uparrow$ and g is called $\mathsf{T}_\downarrow$. The convention is that $\mathsf{T}^{\mathsf{T}^\uparrow}_{\mathsf{T}_\downarrow} p$ is abbreviated to $\mathsf{T}p$.

No syntax or computation rules are given for α, but the evaluation of expressions is assumed to terminate. Thus closed data expressions merely stand for their values. It must be possible to determine when two elements

$$\mathsf{a}\,(x)\,?\,m \xrightarrow{\mathsf{a}(v)?} m[v/x] \qquad \mathsf{a}\,(u)\,!\,m \xrightarrow{\mathsf{a}(u)!} m$$

$$\frac{m_1 \xrightarrow{\mu} m_1'}{m_1 + m_2 \xrightarrow{\mu} m_1'} \qquad \frac{m_1 \xrightarrow{\mu} m_1'}{(\mathsf{tt} \to m_1, m_2) \xrightarrow{\mu} m_1'}$$

$$\frac{m_1 \xrightarrow{\mu} m_1'}{m_1 \mid m_2 \xrightarrow{\mu} m_1' \mid m_2} \qquad \frac{m_1 \xrightarrow{\mathsf{a}(v)!} m_1' \quad m_2 \xrightarrow{\mathsf{a}(v)?} m_2'}{m_1 \mid m_2 \xrightarrow{\tau} m_1' \mid m_2'}$$

$$\frac{m \xrightarrow{\tau} m'}{m \uparrow I \xrightarrow{\tau} m' \uparrow I} \qquad \frac{m \xrightarrow{\mu} m'}{m \uparrow I \xrightarrow{\mu} m' \uparrow I}\ \mathrm{name}\,(\mu) \in I$$

$$\frac{m[d/z] \xrightarrow{\mu} m'}{A\,(d) \xrightarrow{\mu} m'}\ A\,(z) = m \qquad \frac{m \xrightarrow{\mu} m'}{\phi m \xrightarrow{\phi\mu} \phi m'}$$

There are symmetric rules for $m_1 + m_2$ and $m_1 \mid m_2$ where m_2 acts and sends; also a symmetric conditional.

Table 2
Operational rules for CCS

of α are equal. CBS is first order. That is, α may not itself involve the type $\mathrm{Proc}\,(\beta)$ for any β.

Occurrences of x are *bound* in $x?\,p$ and $\langle x, p\rangle \& \langle v, q\rangle$, and the scope of x here is p. $p[v/x]$ denotes the result of substituting v for x in p. (Data)-open and closed processes are defined in the obvious way; thus $x?\,x!\,\mathbf{0}$ is closed while $x!\,\mathbf{0}$ is open. Only closed processes act. Table 1 shows the inference rules for CBS.

The following rules define $\xrightarrow{\delta!}$ independently of $\xrightarrow{\tau!}$ and $\xrightarrow{v!}$, and agree with the previous definition of $\xrightarrow{\delta!}$.

$$\mathbf{0} \xrightarrow{\delta!} \qquad x?\,p \xrightarrow{\delta!} \qquad \frac{p \xrightarrow{\delta!} \quad q \xrightarrow{\delta!}}{p \mid q \xrightarrow{\delta!}} \qquad \frac{p \xrightarrow{\delta!}}{(\mathsf{tt} \to p, q) \xrightarrow{\delta!}} \qquad \frac{p \xrightarrow{\delta!}}{\mathsf{T}p \xrightarrow{\delta!}}$$

3.2 CCS

The syntax of CCS is given by

$$m ::= \mathbf{0} \mid \mathsf{a}\,(x)\,?\,m \mid \mathsf{a}\,(v)\,!\,m \mid m{+}m \mid m|m \mid b \to m, m \mid m \uparrow I \mid \mathsf{T}_\phi m \mid A\,(d)$$

where a ranges over channel names, I ranges over sets of channel names, and processes are defined in an environment of equations of the form $A(z) = m$, where z is a variable and d a constant of the data type parameterising the equation. The relabelled process $\mathsf{T}_\phi m$, where ϕ is a bijection over channel names is simply written ϕm. If an action $\mu = \mathsf{a}(v)!$ then $\phi(\mathsf{a})(v)!$ is simply written $\phi\mu$, i.e., the relabelling ϕ applies only to the channel name of the action. Table 2 shows the inference rules for CCS.

4 A CCS interpreter for CBS

This is given in Table 3 and follows the sketch in Section 2.

$M(p)$ has the interface

$$I_\alpha = \{\mathsf{hear}_\alpha(\alpha)?, \mathsf{heard}_\alpha!, \mathsf{say}_\alpha?, \mathsf{said}_\alpha(\alpha_{\tau\delta})!\}$$

where the types in parentheses refer to the data carried along the respective channels. The subscripts α are usually clear from the context and are dropped. M should be thought of as polymorphic: applied to $p \in \mathrm{Proc}(\alpha)$, it produces a CCS process that uses channels subscripted by α. Alternatively, M can be thought of as monomorphic, and being subscripted by α; this subscript too is usually dropped.

The first four CBS constructors are simple. Here bisimulation between p and $M(p)$ is immediate in terms of the CBS transitions of Definition 2.1, without even any mention of τ moves.

The conditional combinator is easiest understood by seeing that the b has to be evaluated before any moves can be derived for either of $b \to p, q$ and $b \to M(p), M(q)$. If b is tt, then these two have respectively the behaviours of p and $M(p)$.

The behaviour of $M(p \mid q)$ has already been explained informally. The components $M(p)$ and $M(q)$ are relabelled by ϕ_l and ϕ_r, and cannot interact directly with the environment; only PAR does that. If it gets a $\mathsf{hear}(v)?$ command, it distributes the v to both $M(p)$ and $M(q)$, waiting until each has acknowledged it by a heard, and then sends its own $\mathsf{heard}!$ to the environment. If $M(p \mid q)$ gets a say command, this is offered, suitably relabelled, to both $M(p)$ and $M(q)$. If the first taker says δ, the second gets a chance. If both say δ, that is reported by $\mathsf{said}(\delta)$. If either has a $w \neq \delta$ to say, it is passed up to the environment via said; if $w \neq \tau$, it is first given to the other via a hear. The τ consideration is an optimisation.

The implementation of the scoping construct should be easy to understand.

4.1 CBS forms have no CCS moves other than CBS experiments

Proposition 4.5 establishes $\forall p \in \mathrm{CBS}.\, p \sim M(p)$, and shows that CBS forms have all the necessary CBS experiments and no other. This subsection shows that CBS forms have no irrelevant CCS moves, i.e. that every sequence of

$$
\begin{array}{lll}
M(\mathbf{0}) & = \mathsf{hear}(v)?\,\mathsf{heard!}\,M(\mathbf{0}) & +\mathsf{say?}\,\mathsf{said}(\delta)!\,M(\mathbf{0}) \\
M(x?\,p) & = \mathsf{hear}(v)?\,\mathsf{heard!}\,M(p[v/x]) & +\mathsf{say?}\,\mathsf{said}(\delta)!\,M(x?\,p) \\
M(u!\,p) & = \mathsf{hear}(v)?\,\mathsf{heard!}\,M(u!\,p) & +\mathsf{say?}\,\mathsf{said}(u)!\,M(p) \\
M(\langle x,p_1\rangle\&\langle u,p_2\rangle) & = \mathsf{hear}(v)?\,\mathsf{heard!}\,M(p_1[v/x]) & +\mathsf{say?}\,\mathsf{said}(u)!\,M(p_2) \\
M(b\to p_1,p_2) & = b\to M(p_1),M(p_2) & \\
M(p_1\mid p_2) & = (\phi_1(M(p_1))\mid\phi_2(M(p_2))\mid \mathrm{PAR})\uparrow I & \\
M(\mathsf{T}p) & = \left(\phi_1(M(p))\mid \mathrm{TRANS}\left(\mathsf{T}^{\uparrow},\mathsf{T}_{\downarrow}\right)\right)\uparrow I &
\end{array}
$$

$$I = \{\mathsf{hear}(\alpha)?\,,\mathsf{heard!}\,,\mathsf{say?}\,,\mathsf{said}(\alpha_{\tau\delta})!\,\}$$

$$
\begin{array}{rl}
\text{for } i\in\{1,2\},\ \phi_i = \{ & \mathsf{hear}(\alpha)?\mapsto \mathsf{hear}_i(\alpha)?\,, \\
& \mathsf{heard!}\mapsto \mathsf{heard}_i!\,, \\
& \mathsf{say?}\mapsto \mathsf{say}_i?\,, \\
& \mathsf{said}(\alpha_{\tau\delta})!\mapsto \mathsf{said}_i(\alpha_{\tau\delta})!\ \}
\end{array}
$$

$$
\begin{array}{lll}
\mathrm{PAR} & = \mathsf{hear}(v)? & \mathsf{hear}_1(v)!\,\mathsf{heard}_1? \\
 & & \mathsf{hear}_2(v)!\,\mathsf{heard}_2? \\
 & & \mathsf{heard!}\,\mathrm{PAR} \\
 & +\mathsf{say?} & \mathrm{SAY} \\
\mathrm{SAY} & = \sum_{i=1,2} & \mathsf{say}_i!\,\mathsf{said}_i(w)?\,(w=\delta\to \mathrm{LAST}(1-i),\mathrm{SAID}(1-i,w)) \\
\mathrm{LAST}(i) & = & \mathsf{say}_i!\,\mathsf{said}_i(w)?\,(w=\delta\to \mathsf{said}(\delta)!\,,\quad \mathrm{SAID}(1-i,w)) \\
\mathrm{SAID}(i,w) & = w=\tau\to & \mathsf{said}(\tau)!\,\mathrm{PAR}, \\
 & & \mathsf{hear}_i(w)!\,\mathsf{heard}_i?\,\mathsf{said}(w)!\,\mathrm{PAR}
\end{array}
$$

$$
\begin{array}{lll}
\mathrm{TRANS}(f,g) = \mathsf{hear}(v)?\,g(v)=\tau\to & \mathsf{heard!}\,\mathrm{TRANS}(f,g)\,, \\
 & \mathsf{hear}_1(g(v))!\,\mathsf{heard}_1?\,\mathsf{heard!}\,\mathrm{TRANS}(f,g) \\
+\ \mathsf{say?}\,\mathsf{say}_1!\,\mathsf{said}_1(w)?\,\mathsf{said}\ (f(w))!\,\mathrm{TRANS}(f,g) &
\end{array}
$$

Table 3
CCS interpreter for CBS

CCS moves from a CBS form either includes a CBS experiment as a leading subsequence, or can be extended, and every extension leads to a CBS experiment.

This is done by three lemmas. The first shows that every CBS form is open to any CBS experiment, and has no other initial CCS moves. The next two show that every subsequent CCS evolution completes the experiment that has begun.

All three lemmas use the same structure of proof, induction on the structure of p. The only cases not immediate are $M(p_1 \mid p_2)$ and $M(\mathsf{T}p)$, where the initial moves come not from the components but from PAR and from TRANS respectively.

Lemma 4.1 (Command enabled) *$\forall p \in CBS$, $M(p)$ has a $\mathsf{say}?$ move and $\mathsf{hear}(v)?$ moves for all v, and no other moves.*

Proof. The induction here is only used to show the absence of τ moves from components. □

Lemma 4.2 (Input completion) *$\forall p \in CBS.\ \forall v.$ every sequence of CCS moves starting with $M(p) \xrightarrow{\mathsf{hear}(v)?}$ leads to a $v?$ CBS experiment.*

Proof. Note that there is only one sequence for each v. The induction is used to show that $M(p_1 \mid p_2)$ completes a $v?$ experiment assuming that p_1 and p_2 both do. That these components have such an experiment is guaranteed by Lemma 4.1. □

Lemma 4.3 (Output completion) *$\forall p \in CBS.$ every sequence of CCS moves starting with $M(p) \xrightarrow{\mathsf{say}?}$ leads to a $w!$ CBS experiment.*

Proof. Non-determinism arises here because either component in $M(p_1 \mid p_2)$ might respond to the relayed $\mathsf{say}?$ command. Three cases arise depending on the $w!$ from this component (there will be one by induction), and in the case $w \neq \tau$ and $w \neq \delta$, Lemmas 4.1 and 4.2 guarantee that the other component has a matching $w?$ experiment and will complete it. □

Proposition 4.4 *Every CCS evolution of a CBS form includes a CBS experiment as a leading subsequence.*

Proof. Implied by the preceding lemmas. □

4.2 CBS forms implement CBS processes upto CBS bisimulation

Proposition 4.5 $\forall p.\, p \sim M(p)$

Proof. Directly, by showing that $\{\langle p, M(p)\rangle \mid p \in \mathrm{CBS}\}$ is a CBS bisimulation. It is convenient to let $\natural$ range over $\{!, ?\}$.

Forwards Let $p \xrightarrow{w\natural}_n p'$ mean $p \xrightarrow{w\natural} p'$ can be derived in at most n inferences.

Then it is to be shown that $\forall n \geq 1.\, H(n)$, where $H(n)$ is the hypothesis

that $\forall p, w, \natural, p'.\ p \xrightarrow{w\natural}_n p' \Longrightarrow M(p) \xrightarrow{w\natural} M(p')$. The proof is by induction on n.

The base case is $H(1)$. This arises from the constructors $\mathbf{0}$, $x?\,p$, $v!\,p$ and $\langle x, p\rangle \& \langle u, q\rangle$. They are easy. $H(1)$ also includes the rule $p \xrightarrow{\tau?} p$, which is covered by the clause $m \xrightarrow{\tau?} m$ in Definition 2.1.

For the step, $H(n+1)$ is to be proved from $H(n)$. Here p has three cases, where the constructors are the conditional, parallel composition and scoping. Just three subcases are offered as examples; the others are similar.

Conditional. Suppose $(b \to p, q) \xrightarrow{w\natural}_{n+1} p'$ and b is tt. Then it must be that $p \xrightarrow{w\natural}_n p'$. Then by $H(n)$, $M(p) \xrightarrow{w\natural} M(p')$, so $M(b \to p, q) \xrightarrow{w\natural} M(p')$ and $H(n+1)$ follows.

Parallel composition. Suppose $p \mid q \xrightarrow{v!}_{n+1} p' \mid q'$. One way this can arise is $p \xrightarrow{v!}_n p'$ and $q \xrightarrow{v?}_n q'$. By $H(n)$, $M(p) \xrightarrow{v!} M(p')$, and $M(q) \xrightarrow{v?} M(q')$. Then $M(p \mid q) \xrightarrow{v!} M(p' \mid q')$ follows via several τ's, giving $H(n+1)$.

Scoping. If $\mathsf{T}_{\downarrow}(v) = \tau$, then $\mathsf{T}p \xrightarrow{v?}_2 \mathsf{T}p$, as $p \xrightarrow{\tau?}_1 p$ is given. No induction is involved, for $M(\mathsf{T}p) \xrightarrow{v?} M(\mathsf{T}p)$ is direct. Only TRANS acts, $M(p)$ is not involved. The case where $\mathsf{T}_{\downarrow}(v) \neq \tau$ is no harder than parallel composition.

Backwards Let m, m_1, m_2 range over CCS processes. Let $H(p)$ be the property that $\forall w, \natural, m.\ M(p) \xrightarrow{w\natural} m \Longrightarrow \exists p'.\, m \equiv M(p')$ and $p \xrightarrow{w\natural} p'$. It is to be shown that $\forall p.\, H(p)$. The proof is by structural induction on p.

The base case is $H(\mathbf{0})$ and is easy.

The three prefixes too follow directly without need for induction. For example, $M(x?\,p)$ has only two CBS moves, one via $\xrightarrow{\delta!}$ to $M(x?\,p)$ and one via $\xrightarrow{v?}$ to $M(p[v/x])$, and $H(x?\,p)$ follows without using $H(p)$.

If b is tt, then $M(b \to p_1, p_2)$ has exactly the behaviour of $M(p_1)$, and then $b \to p_1, p_2$ has exactly the behaviour of p_1. So $H(b \to p_1, p_2)$ follows from $H(p)$.

Suppose $H(p_1)$ and $H(p_2)$ and $M(p_1 \mid p_2) \xrightarrow{v!} m$. This can only arise by cases such as $m \equiv (\phi_1(m_1) \mid \phi_2(m_2) \mid \mathrm{PAR}) \uparrow I$ where $M(p_1) \xrightarrow{v!} m_1$ and $M(p_2) \xrightarrow{v?} m_2$, say. Then by $H(p_1)$, $m_1 \equiv M(p_1')$ for some p_1' such that $p_1 \xrightarrow{v!} p_1'$, and by $H(p_2)$, $m_2 \equiv M(p_2')$ for some p_2' such that $p_2 \xrightarrow{v?} p_2'$. Then $p_1 \mid p_2 \xrightarrow{v!} p_1' \mid p_2'$, yielding $H(p_1 \mid p_2)$.

$H(\mathsf{T}p)$ similarly follows from $H(p)$.

□

A simple extension to the interpreter is to change PAR so that after a $\mathsf{hear}(x)?$ it first transmits both $\mathsf{hear}_1(x)!$ and $\mathsf{hear}_2(x)!$ and then waits for $\mathsf{heard}_1?$ and $\mathsf{heard}_2?$ in either order. This introduces some parallelism, but a similar proof goes through.

A more difficult and interesting development would be to drop the heard acknowledgements. They make the present proof easier, by ensuring that the derivative m of a CBS move $M(p) \xrightarrow{v?} m$ is already of the form $M(p')$

rather than a few τ moves away from it. Without heard, both $M(p_1 \mid p_2)$ and $M(\mathsf{T}p)$ would accept a $\mathsf{hear}(v)?$ command and be immediately ready for other commands while the v was still trickling down through the system. Then $p \xrightarrow{v?} p'$ would be matched by just $M(p) \xrightarrow{\mathsf{hear}(v)?} m \sim M(p')$. A proof that $M(p) \sim p$ still holds with this setup would validate a more accurate CCS encoding of a lazy interpreter.

A further interesting development would be to drop the say commands. This would mimic a bottom up interpreter where the components of a parallel composition generate requests to speak rather than respond to a command to speak. Such an interpreter resolves contention between speakers by parallelism rather than by pseudo-random numbers.

5 On the translation of operational semantics

Propositions 4.5 and 4.4 establish a more direct relation between p and $M(p)$ than a standard result would. This new direct relation makes no use of equivalences over CCS processes (though of course it induces one — two CCS processes can be defined to be equivalent if they are equivalent to the same CBS process). Thus the result here is very different from the standard ones, a departure from tradition that calls for comment. The next subsection argues that the present result is stronger than a standard one would be.

5.1 Preserving semantics upto equivalence

First, note that preservation of equivalences by itself is not sufficient to establish the most fundamental requirement of any translation, the preservation of meaning. For example, a translation from English to Swedish that takes each equivalent of "come" to some equivalent of "go" in Swedish preserves semantics upto equivalence, but is nonsensical. If a standard result is augmented by a relation between the languages, it is only the augmentation that actually says something direct about the preservation of meaning! This paper takes an altogether more direct route.

But this problem with the unadorned standard result is usually not an issue. For example, as has been pointed out, the SCCS actions in Holmer's translation carr exactly the labels of the corresponding CBS action. (In fact, more can be said than $p_1 \sim p_2 \iff S(p_1) \sim S(p_2)$, for a kind of bisimulation-upto-bisimulation can be established between p and $S(p)$). Similarly, abstract machines often produce the exact value or action of the interpreted program, so no relation has to be set up between the implemented language and the machine language.

It is a matter of discussion what the present result says about a conjecture that CCS cannot interpret CBS upto any reasonable equivalence, since the equivalence over CCS that is preserved is defined with respect to the CCS protocols for TSR procedures (though not with respect to M itself!), and

therefore means something only in this context. What is clear is that $M(p)$ is independent of the context of p, yet implements a broadcast by different numbers of handshakes in different contexts, and ensures $M(p) \sim p$ where $\sim$ is a new relation, though hopefully meaningful. Holmer's conjecture was inspired by the belief that these were unlikely achievements, and therefore can be said to be disproved in spirit if not in letter.

5.1.1 Uniformity conditions

Ene and Muntean prove that Holmer's conjecture is true if the translation is "uniform", which requires among other things that

$$M(p_1 \mid p_2) \stackrel{\text{def}}{=} M(p_1) \mid M(p_2)$$

As pointed out above, M in this paper is not uniform. The relevant fragment is

$$M(p_1 \mid p_2) \stackrel{\text{def}}{=} (\phi_1(M(p_1)) \mid \phi_2(M(p_2)) \mid \text{PAR}) \uparrow I$$

where ϕ_1 and ϕ_2 are (CCS) relabelling operators, $\uparrow$ is (CCS) restriction, and PAR is a CCS process that distributes broadcasts over the components $M(p_1)$ and $M(p_2)$. Holmer's SCCS translation too uses a similar structure. Indeed the present author finds this uniformity condition rather a strong requirement, and would find interesting any function that satisfies it and yet relates significantly different modes of communication.

Another "uniformity condition", adopted by Ene and Muntean and broken by both the SCCS and CCS interpreters for CBS, says something like

$$M(\phi p) \stackrel{\text{def}}{=} \phi(M(p))$$

This condition is in any case not directly applicable to this paper, as CBS replaces the relabelling and restriction operators of CCS by a single scoping construct. The details are irrelevant here; what is interesting is that this condition too applies to static operators whose definition is an integral part of the communication model, thus strengthening the feeling that a uniform translation is only meant to link two quite similar modes of communication.

6 Conclusions and further work

6.1 Conclusions

Both [8,4] and informal conversations suggest that the general guess has been that CCS cannot interpret CBS. So the first achievement here has to be the positive answer, disproved Holmer's conjecture at least in spirit, however simple it turned out to be. It can be debated whether it shows that CCS can interpret CBS upto a reasonable equivalence, since the preserved equivalence over CCS is defined with respect to the CCS protocols for TSR procedures.

The style of result, linking very different calculi, seems unusual, as does the specific definition of CBS experiments over CCS processes. This author does not know of similar definitions to link an abstraction with an implementation, or of definitions of experiment that bracket a sequence of internal moves by external ones, though both seem very natural. However, these departures from tradition should be treated with caution until they have passed scrutiny.

The critiques of the "preservation of equivalence" criterion for implementation, and of the "uniformity" conditions, may be of some interest.

The present result does not say that CCS has the properties of CBS. It says that the subset of CCS processes of the form $M(p)$, where p is a CBS process, have CBS properties *when viewed through CBS experiments*. In the light of Holmer's conjecture, it does say something new about the expressive power of CCS, by combining the new experiment definition with the well-known facts that CCS can encode procedure calls and that CBS already had a functional interpreter.

The original goal, of showing that the global synchronisation of the CBS model is an abstraction from a sequence of local synchronisations, has been achieved modulo the non-standard correctness requirement. Similar abstractions from local to global synchronisation are of interest in the setting of LUSTRE and other synchronous languages [1]).

6.2 Future work

Several issues vy for immediate attention. Variant implementations have already been mentioned, and are listed below in order of increasing complexity. It remains to be seen whether these can be proved correct.

Getting rid of the heard communications would make for a lazy implementation, which is already available in practice: TSR implemented using a lazy language.

Getting rid of the say communications would make for a distributed implementation. The author has implemented such a bottom up interpreter, where components generate speech requests autonomously. These bubble up the process tree, while heard values and permits to speak travel down, obliterating any requests they meet on the way.

TSR does not feed a τ produced by one component to another, but one can go further, and not regard τ as a value to be passed up along said. Various alternatives are possible; one, discussed in [14], allows τ's to synchronise and hardly changes CBS. Others would induce a new weak equivalence on CBS, which may be related to the barbed bisimulations of [7].

Indeed even the present result induces equivalences on both CCS and CBS and both are worth studying. While that on CCS may be regarded as a curiosity, adding to a very large number of existing CCS equivalences, that on CBS is a physically meaningful equivalence, generated by an implementation.

6.3 Related work

[8,4] are of course the main references, but many others have already been mentioned. There are also several more implementations of CBS than those already mentioned: an object oriented one by Wilhelmi (2000) [17], and other styles of functional CBS implementation by Jones (1993) [9] and Petersson (1994) [12].

6.4 Acknowledgements

I thank the referees for their detailed criticisms. They insisted on Proposition 4.4, which I had not put in even though the equivalent exists in the formal proofs of correctness of TSR, and on a more careful statement of what has actually been achieved.

The abstract machines reported here were first developed long ago and presented at various meetings, though not written up. Andy Gordon suggested encoding these in CCS, making the hoped for equivalence result much more significant. A version of this proof was presented at the IFIP WG 2.2 meeting (1999) and at the Chalmers Winter meeting (2000). Joachim Parrow encouraged a clean up of that proof.

The first version of this paper reported only recent developments, giving an impression that the goal was to disprove Holmer's conjecture. The present version reflects the full picture better, with the main goal being a formalisation of concurrency issues in implementations of CBS.

References

[1] *Crisys project home page*, http://ais.gmd.de/ ap/crisys/.

[2] Andersen, J., E. Harcourt and K. V. S. Prasad, *A machine verified sorting algorithm*, Technical Report RS-96-4, BRICS, Aarhus (1996).

[3] de Simone, R., *Higher level synchronising devices in MEIJE-SCCS*, Theoretical Computer Science **37** (1985), pp. 245–268.

[4] Ene, C. and T. Muntean, *Expressiveness of broadcast communication*, in: G. Ciobanu and G. Paun, editors, *Fundamentals of Computation Theory, 12th International Symposium, FCT '99*, LNCS **1684** (1999), pp. 258–268.

[5] Giménez, E., *An application of co-inductive types in coq: verification of the alternating bit protocol*, in: *Workshop on Types for Proofs and Programs*, LNCS **1158** (1995), pp. 135–152.

[6] Harcourt, E., P. Paczkowski and K. V. S. Prasad, *A framework for representing value-passing parametric processes* (1995), http:// www.cs.chalmers.se / prasad /parametric.html.

[7] Hennessy, M. and J. Rathke, *Bisimulations for a calculus of broadcasting systems*, Theoretical Computer Science **200** (1998), pp. 225–260.

[8] Holmer, U., *Translating broadcast communication into SCCS*, in: E. Best, editor, *CONCUR'93*, LNCS **715** (1993), pp. 188–201.

[9] Jones, S., *Translating CBS to LML*, Technical report, Department of Computer Science, Chalmers University of Technology (1993).

[10] Milner, R., *Calculi for synchrony and asynchrony*, Theoretical Computer Science **25** (1983), pp. 267–310.

[11] Milner, R., "Communication and Concurrency," Prentice Hall, 1989.

[12] Petersson, J., *Tools for a calculus of broadcasting systems*, Licentiate thesis, Department of Computer Science, Chalmers University of Technology (1994).

[13] Prasad, K. V. S., *Programming with broadcasts*, in: E. Best, editor, *CONCUR'93*, LNCS **715** (1993), pp. 173–187.

[14] Prasad, K. V. S., *A calculus of broadcasting systems*, Science of Computer Programming **25** (1995), pp. 285–327.

[15] Prasad, K. V. S., *Broadcasting in time*, in: P. Ciancarini and C. Hankin, editors, *COORDINATION'96*, LNCS **1061** (1996), pp. 321–338.

[16] Runciman, C. and D. Wakeling, *Profiling parallel functional computations*, in: J. T. O'Donnell and K. Hammond, editors, *Glasgow Workshop on Functional Programming*, Springer, 1993 pp. 236–251.

[17] Wilhelmi, S., *A virtual TCBS machine on top of C++*, Web site, Univ. of Karlsruhe (2000), http://goethe.ira.uka.de/ wilhelmi/cbs/.

Encoding Distributed Areas and Local Communication into the π-Calculus

Tom Chothia[1] and Ian Stark

Laboratory for Foundations of Computer Science
Division of Informatics, The University of Edinburgh
Mayfield Road, Edinburgh EH9 3JZ, Scotland, UK
{stark,tpcc}@dcs.ed.ac.uk

Abstract

We show how the π-calculus can express local communications within a distributed system, through an encoding of the *local area π-calculus*, an enriched system that explicitly represents names which are known universally but always refer to local information. Our translation replaces point-to-point communication with a system of shared local ethers; we prove that this preserves and reflects process behaviour. We give an example based on an internet service dæmon, and investigate some limitations of the encoding.

1 Introduction

Part of the power of the π-calculus is that names serve a dual rôle: as well as carriers of communication, they have unique *identity*. By default the scope of these coincide, so that any two processes that know a common name can also use it to communicate. In many distributed systems, however, this is not so natural: widely-known names may be intended to refer always to local information. For example, the standard `finger` service operates on well-known port number 79, but should of course give a different answer on different machines. We can even take this as a defining characteristic of a distributed system: that a single name may refer to different things depending on where it is used.

The *local area π-calculus* ($la\pi$) captures this phenomenon of names which are known universally but always refer to local information. It extends the π-calculus so that a channel name can have within its scope several disjoint *local areas*. Such a channel name may be used for communication within an area, it may be sent between areas, but it cannot itself be used to transmit

[1] Supported by the UK Engineering and Physical Sciences Research Council

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

information from one area to another. Areas are arranged in a hierarchy of *levels*, distinguishing for example between a single application, a machine, or a whole network.

In previous work we introduced the local area π-calculus and showed how a combination of static typing and dynamic checks can give flexible scope for name identity while suitably restricting communication to local areas [8]. In this paper we give a compositional translation of $la\pi$ into the plain π-calculus, and prove that it correctly encodes process behaviour.

The main challenge for representing local areas in the π-calculus is how to prevent communication on a channel between different areas, while still preserving the identity of names. Our solution is to replace communication inside an area with communication on a new channel created just for that area. As well as sending the original data, an encoded output sends the channel name as well. Output action $\bar{a}\langle b\rangle$ becomes $\bar{e}\langle a, b\rangle$ where e is a name that corresponds to the appropriate local area — a shared *ether*. An input action on a channel translates to a process that listens for communication on the relevant ether. When it receives a message it tests the first element against the desired channel name; if they match it accepts the input, otherwise it rebroadcasts the message.

This translation makes explicit the different rôles of names, identity and communication, by mapping them to two distinct sets of names. Names like a and b serve purely as data, for identification; ether names like e are for communication only. The scope of data names manages who knows what, while the scope of ether names handles locality of communication.

The evident motivation for this model is packet communication on an ethernet: instead of sending data directly to its destination, we drop a packet into the ether. Listening processes pick up all packets and sift out the ones they are interested in. In our case the tree of nested areas (applications, machines, networks) gives rise to a hierarchy of ethers, with a process using a different ether for each level of communication (local to an application; within a machine; over the network).

There is a close match between the behaviour of a process in $la\pi$ and its π-calculus translation; we show a form of weak bisimilarity on outputs. There is some loss of information though, in that translated terms may make additional silent moves, as packets pass over the ether, and an ether may indiscriminately accept and rebroadcast packets in which no receiver is interested.

The rest of the paper is arranged as follows. Section 2 presents the local area π-calculus, its type system and operational semantics. Section 3 reviews the version of the π-calculus we use. Section 4 gives the encoding between these, and Section 5 outlines the result that a process and its encoding are weakly bisimilar on output. Section 6 presents an example of the translation at work, and Section 7 concludes.

Related work

Local areas are in some sense a more regular form of CCS *restriction* or CHOCS *blocking*. Vivas and Dam have studied the effect of these on the higher-order π-calculus, and given an encoding into the π-calculus [23]. However, their encoding relies on blocking being carried out explicitly on individual names, and is (in their own words) both complex and indirect (though chiefly because it also handles higher-order operations).

Cardelli, Ghelli and Gordon take a different approach to limiting communication with their notion of name *groups* [5]. Ingeniously, introducing these into the type system allows one to check statically that a process never passes out certain names. In our system, by contrast, names may be passed anywhere — only their action is limited.

There are numerous projects addressing *locations* in the π-calculus [3,4,13], and distributed systems more generally [7,10,18,21]. These overlap with our approach to varying degrees; some look at issues of when locations fail, others limit communication in particular ways. In the extreme, systems like mobile ambients make all interaction local: remote agents must move around to talk to each other [6].

Translating miscellaneous concurrent systems into each other is also a popular sport. Nestmann and Pierce give a good overview of what makes for a good encoding in their work on deconstructing choice [16]. Among many examples, Fournet and others have implemented mobile ambients in the JoCaml languages [9]; this translates one notion of distributed areas into another, whereas we make areas disappear entirely. Moreover, their focus is on providing a basis for a implementation of Mobile Ambients, so much attention is paid to making it run efficiently. Sangiorgi describes in great detail a rather different encoding of locations in order to express non-interleaving semantics [20].

2 A π-calculus with local areas

The local area π-calculus extends a standard π-calculus with nested *local areas* arranged in *levels*. The π-calculus part is unexceptional: it happens to be polyadic (channels carry tuples rather than single values [14]) and asynchronous (output actions always succeed [2]). To illustrate the extensions, we present a brief example, based on a mechanism for selecting internet services.

When a browser contacts a web server to fetch a page, or a person operates `finger` to list the users on another machine, both connect to a numbered "port" on the remote host: port 80 for the web page, port 79 for the finger listing. Of course, this only works if both sides agree; and there is a real-world committee to set this up [12]. Under Unix, the file `/etc/services` holds a list mapping numbers to services. There is also a further level of indirection: most machines run only a general meta-server `inetd`, the Internet dæmon, which listens on all ports. When `inetd` receives a connection, it looks up the

port in `/etc/services`, and then consults a second file which identifies the program to provide that service. The `inetd` starts the program and hands it a connection to the caller. A model of this procedure in the local area π-calculus looks like this (we omit detailed type information).

Client	$Carp = host[\, \nu c.(\overline{pike}\langle finger, c\rangle \,|\, c(x).\overline{print}\langle x\rangle)\,]$
Server	$Pike = host[\, !pike(s,r).\bar{s}\langle r\rangle \mid !finger(y).\bar{y}\langle PikeUsers\rangle$
	$\mid !daytime(z).\bar{z}\langle PikeDate\rangle\,]$
System	$net[Carp \,|\, Pike]$
Names	$pike$ and c operate at net level;
	$finger$, $daytime$ and $print$ operate at $host$ level

This shows a client machine *Carp* that wishes to contact a server *Pike* with a finger request; the $host[-]$ and $net[-]$ markers indicate local areas. The client has two components: the first transmits the request, the second prepares to print the result. Server *Pike* comprises three replicating processes: a general Internet dæmon, a Finger dæmon, and a time-of-day dæmon. Channel *pike* is the internet address of the server machine, while the free names *finger* and *daytime* represent well-known port numbers. In operation, *Carp* sends its request to *Pike* naming the finger service and a reply channel c. The Internet dæmon on *Pike* handles this by retransmitting the contact c over the channel named *finger*. The Finger dæmon collects this and passes information on *PikeUsers* back to the waiting process at *Carp*. Figure 1 gives a graphical representation of the interaction.

In the plain π-calculus, this models leaks: because the names *finger* and *daytime* are visible everywhere, even when the Internet dæmon on *Pike* has collected the request there is no protection against a Finger dæmon on some different server actually handling it. Restricting the scope of *finger* to host *Pike* would be no solution, because then *Carp* could not formulate the request because it has to know the name of the service.

In the local area π-calculus, each channel has an assigned level of operation, which limits how far communication on that channel may travel. In this case, although *finger* is globally known, messages over it remain within a single *host*. This breaks the Catch-22: *Carp* and *Pike* agree on the name for the *finger* service, but different Finger dæmons on separate machines do not interfere with each other.

2.1 Syntax

The calculus is built around two classes of identifiers:

$$\begin{array}{rr} \text{channels} & a, b, c, x, y, query, reply, \ldots \in Chan \\ \text{and levels} & \ell, m, app, host, net, \ldots \in Level. \end{array}$$

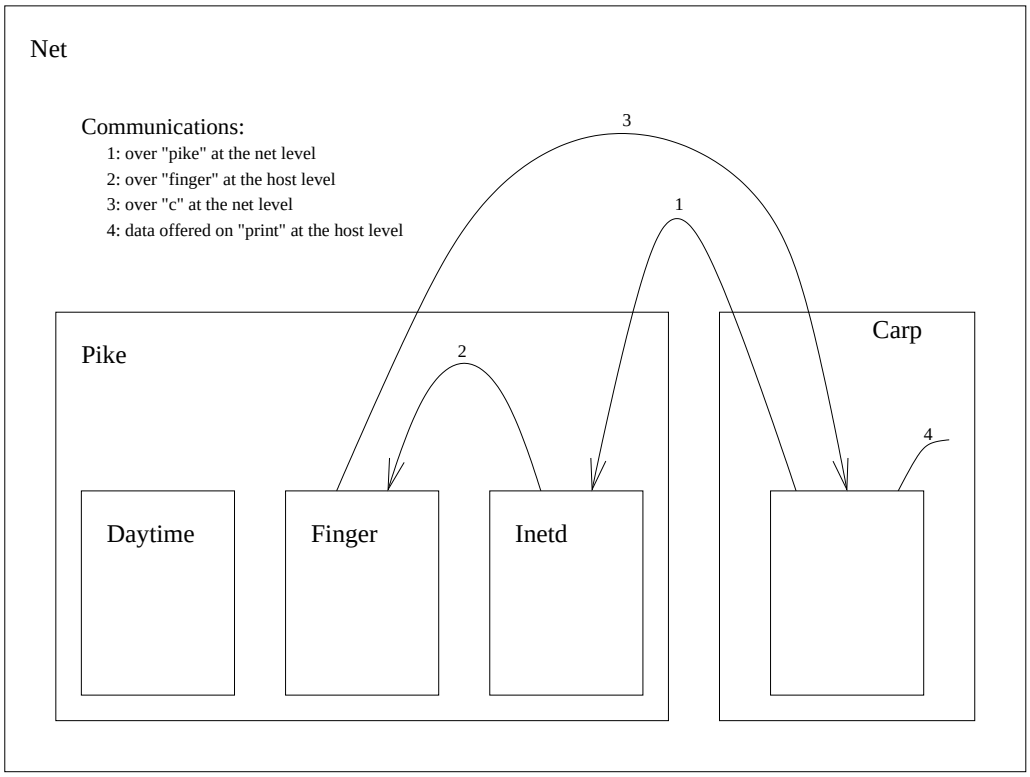


Fig. 1. Operating a remote `finger` service through an `inet` dæmon

Channel names are drawn from a countably infinite supply, $Chan$. Syntactically, they behave exactly as in the π-calculus. Levels are rather more constrained: we assume prior choice of some finite and totally ordered set $Level$. Throughout the paper we use $app < host < net$, and take ℓ and m as metavariables for levels.

Processes are given by the following syntax, based on the asynchronous polyadic π-calculus.

$$
\begin{array}{lllll}
\text{Process } P, Q ::= & 0 & \text{inactive process} & \mid \bar{a}\langle\vec{b}\rangle & \text{output tuple} \\
\mid & P \mid Q & \text{parallel composition} & \mid a(\vec{b}).P & \text{input} \\
\mid & \ell[P] & \text{local area at level } \ell & \mid\ !a(\vec{b}).P & \text{replicated input} \\
\mid & \nu a{:}\sigma.P & \text{fresh channel } a \text{ of type } \sigma & &
\end{array}
$$

The only novelty here is $\ell[P]$, which represents a process P running in a local area at level ℓ; we refer to a process of this form as an *agent*. Areas, like processes, are anonymous; this is in contrast to systems for locations, which are usually tagged with identifiers.

Channel names may be bound or free in any process. The binding prefixes are as usual the input prefixes $a(\vec{b})$, $!a(\vec{b})$ and restriction $\nu a{:}\sigma$; the type σ gives information about the level of operation of a and the tuples it carries. We write $fn(P)$ for the set of free names of process P.

We identify process terms up to *structural congruence* '$\equiv$', the smallest congruence relation containing the following equations:

$$
\begin{array}{rrl}
P \mid 0 \equiv P & a(\vec{b}).P \equiv a(\vec{c}).P\{\vec{c}/\vec{b}\} & \vec{c} \cap fn(P) = \emptyset \\
P \mid Q \equiv Q \mid P & !a(\vec{b}).P \equiv !a(\vec{c}).P\{\vec{c}/\vec{b}\} & \vec{c} \cap fn(P) = \emptyset \\
(P \mid Q) \mid R \equiv P \mid (Q \mid R) & \nu a{:}\sigma.P \equiv \nu b{:}\sigma.P\{b/a\} & b \notin fn(P) \\
\nu a{:}\sigma.0 \equiv 0 & \nu a{:}\sigma.\nu b{:}\tau.P \equiv \nu b{:}\tau.\nu a{:}\sigma.P & a \neq b \\
\ell[\nu a{:}\sigma.P] \equiv \nu a{:}\sigma.(\ell[P]) & (\nu a{:}\sigma.P) \mid Q \equiv \nu a{:}\sigma.(P \mid Q) & a \notin fn(Q)
\end{array}
$$

2.2 Scope and areas

The interesting equation in this structural congruence is $\ell[\nu a{:}\sigma.P] \equiv \nu a{:}\sigma.(\ell[P])$, commuting name binding and area boundaries. A consequence of this is that the scope of a channel name, determined by ν-binding, is quite independent from the layout of areas, given by $\ell[-]$. Scope determines where a name is known, and this will change as a process evolves: areas determine how a name can be used, and these have a fixed structure.

For a process description to be meaningful, this fixed structure of nested areas must accord with the predetermined ordering of levels. For example, a *net* may contain a *host*, but not vice versa; similarly a *host* cannot contain another *host*. Writing $<_1$ for the one-step relation in the total order of levels, we require that in a well-formed process every nested area must be $<_1$-below the one above.

Consider some occurrence of a bound channel name a in a well-formed process P, as the subject of some action: $\bar{a}\langle - \rangle$, $a(-)$, or $!a(-)$. The *scope* of a is the enclosing ν-binding $\nu a{:}\sigma.(-)$. The *local area* of this occurrence of a is the enclosing level ℓ area $\ell[-]$. A single name may have several disjoint local areas within its scope. It is also possible for a name to occur outside any local area of the right level; in this case it may only be treated as data, not used for communication.

2.3 Type system

Channel types have the following rather simple grammar.

$$\text{Type } \sigma \ ::= \ \vec{\sigma}@\ell$$

A type declaration of the form $a : \vec{\sigma}@\ell$ states that a is a level ℓ channel carrying tuples of values whose types are given by the vector $\vec{\sigma}$. The base types are those with empty tuples: a channel of type $()@\ell$ is for synchronization within an ℓ-area. Additional base datatypes like *int* or *string* can be incorporated without difficulty.

Figure 2 presents the rules for deriving type assertions of the form $\Gamma \vdash_\ell P$, where Γ is a finite map from channel names to types. This states that

$$\Gamma \vdash_\ell 0 \qquad \frac{\Gamma \vdash_\ell P \quad \Gamma \vdash_\ell Q}{\Gamma \vdash_\ell P \,|\, Q} \qquad \frac{\Gamma, \vec{b} : \vec{\sigma} \vdash_\ell P}{\Gamma \vdash_\ell a(\vec{b}).P} \quad \Gamma(a) = \vec{\sigma}@m \text{ with } \ell \leq m$$

$$\frac{\Gamma \vdash_\ell P}{\Gamma \vdash_m \ell[P]} \ell <_1 m \qquad \frac{\Gamma, a : \sigma \vdash_\ell P}{\Gamma \vdash_\ell \nu a{:}\sigma.P} \qquad \frac{\Gamma, \vec{b} : \vec{\sigma} \vdash_\ell P}{\Gamma \vdash_\ell\ !a(\vec{b}).P} \quad \Gamma(a) = \vec{\sigma}@m \text{ with } \ell \leq m$$

$$\Gamma \vdash_\ell \bar{a}\langle\vec{b}\rangle \quad \text{if } \Gamma(a) = \vec{\sigma}@m,\ \Gamma(\vec{b}) = \vec{\sigma} \text{ and } \ell \leq m$$

Fig. 2. Types for processes in the local area calculus

OUT	$\Gamma \vdash_\ell\ \bar{a}\langle\vec{b}\rangle \xrightarrow{\bar{a}\langle\vec{b}\rangle} 0$	
IN	$\Gamma \vdash_\ell\ a(\vec{b}).P \xrightarrow{a(\vec{b})} P$	$\vec{b} \cap dom(\Gamma) = \emptyset$
IN!	$\Gamma \vdash_\ell\ !a(\vec{b}).P \xrightarrow{a(\vec{b})} P \,|\, !a(\vec{b}).P$	$\vec{b} \cap dom(\Gamma) = \emptyset$
PAR	$\dfrac{\Gamma \vdash_\ell\ P \xrightarrow{\alpha} P'}{\Gamma \vdash_\ell\ P \,|\, Q \xrightarrow{\alpha} P' \,|\, Q}$	
COMM	$\dfrac{\Gamma \vdash_\ell\ P \xrightarrow{\bar{a}\langle\vec{c}\rangle} P' \quad \Gamma \vdash_\ell\ Q \xrightarrow{a(\vec{b})} Q'}{\Gamma \vdash_\ell\ P \,|\, Q \xrightarrow{\tau} P' \,|\, Q'\{\vec{c}/\vec{b}\}}$	
BIND	$\dfrac{\Gamma, a : \sigma \vdash_\ell\ P \xrightarrow{\alpha} P'}{\Gamma \vdash_\ell\ \nu a{:}\sigma.P \xrightarrow{\alpha} \nu a{:}\sigma.P'}$	$a \notin fn(\alpha)$
AREA	$\dfrac{\Gamma \vdash_\ell\ P \xrightarrow{\alpha} P'}{\Gamma \vdash_m\ \ell[P] \xrightarrow{\alpha} \ell[P']}$	if α is $\bar{a}\langle\vec{b}\rangle$ or $a(\vec{b})$ then $\Gamma(a) = \vec{\sigma}@m'$ with $\ell <_1 m \leq m'$

Fig. 3. Operational semantics for the local area calculus

process P is well-typed at level ℓ in context Γ. The static checking provided by the type system makes two assurances: tuples sent over channels will always be the right size, and a well-typed process will not attempt to communicate on a name above its level of operation.

2.4 Operational semantics

We give the calculus a late-binding, small-step transition semantics, following the regular π-calculus. There is only one addition: although the static type system guarantees that a process will not initiate communication on a name above its operating level, we still need a dynamic check to make sure that no active communication escapes from its local area.

The operational semantics is given as an inductively defined relation on well-typed processes, indexed by their level ℓ and context Γ. Figure 3 gives

rules for deriving transitions of the form

$$\Gamma \vdash_\ell P \xrightarrow{\alpha} Q$$

where $\Gamma \vdash_\ell P$ and α is one of the following.

$$\begin{array}{rcll} \text{Transition } \alpha & ::= & \bar{a}\langle\vec{b}\rangle & \text{output} \\ & | & a(\vec{b}) & \text{input} \\ & | & \tau & \text{silent internal action} \end{array}$$

We make a few observations of these rules and the side-conditions attached to them.

- Active use of the structural congruence '$\equiv$' is essential to make full use of the rules: a process term may need to be rearranged or α-converted before it can make progress. For example, there is no symmetric form for the PAR rule (and no need for one).
- In order to apply the COMM rule it may be necessary to use structural congruence to expand the scope of communicated names to cover both sender and recipient.
- Late binding is enforced by the side-condition $\vec{b} \cap dom(\Gamma) = \emptyset$ on the input rules; this ensures that input names are chosen fresh, ready for substitution $Q\{\vec{c}/\vec{b}\}$ in the COMM rule. Again, we can always α-convert our processes to achieve this.
- The side-condition $m \leq m'$ on AREA is the dynamic check that prevents communications escaping from their local area.

In previous work [8] we demonstrated that reduction preserves types, and as a consequence the semantics does successfully capture the intuition behind areas and levels: areas retain their structure over transitions, and actions on a channel are never observed above the correct operating level.

2.5 Example: Internet daemon

Recall the internet service example given earlier: a host *Carp* wishes to contact a *Finger* dæmon running on host *Pike*, through a general *Inet* dæmon. Figure 4 fills out the details of this, including type definitions.

The type *service* for *finger* and *daytime* expands to $(string@net)@host$. This means that these channels can be used only for *host*-level communication, but the values carried will themselves be *net*-level names. The *host*-level communication is between *Inet* and *Finger* or *Daytime*; the *net*-level communication is the response sent out to the original enquirer, in this case over channel c to machine *Carp*. Channel *pike* has a *net*-level type that acts as a gateway to this, reading the name of a service and a channel where that service should send its reply.

$$Carp = host[\ \nu c{:}response.(\overline{pike}\langle finger, c\rangle \mid c(x).\overline{print}\langle x\rangle)\ ]$$
$$Pike = host[\ Inet \mid Finger \mid Daytime\ ]$$

$$Inet = !pike(s,r).\bar{s}\langle r\rangle$$
$$Finger = !finger(y).\bar{y}\langle PikeUsers\rangle$$
$$Daytime = !daytime(z).\bar{z}\langle PikeDate\rangle$$

$$\Gamma = \{finger, daytime : service \qquad service = response@host$$
$$pike : (service, response)@net \qquad response = string@net$$
$$print : string@host\}$$

$$\Gamma\ \vdash_{net}\ (Carp \mid Pike)$$

Fig. 4. Example of processes using local areas: an Internet server dæmon

We can now apply our operational semantics to see this in action.

$$\begin{aligned}
\Gamma\ \vdash_{net}\ (Carp \mid Pike) &\equiv (\ host[\ \nu c{:}response.(\overline{pike}\langle finger, c\rangle \mid c(x).\overline{print}\langle x\rangle)\ ] \\
&\quad \mid host[\ Inet \mid Finger \mid Daytime\ ]\ ) \\
\text{extend scope of } c \quad &\equiv \nu c{:}response.(\ host[\ \overline{pike}\langle finger, c\rangle \mid c(x).\overline{print}\langle x\rangle\ ] \\
&\quad \mid host[\ Inet \mid Finger \mid Daytime\ ]\ ) \\
\text{expand } Inet \quad &\equiv \nu c{:}response.(\ host[\ \overline{pike}\langle finger, c\rangle \mid c(x).\overline{print}\langle x\rangle\ ] \\
&\quad \mid host[\ !pike(s,r).\bar{s}\langle r\rangle \\
&\qquad \mid Finger \mid Daytime\ ]\ ) \\
\text{communication} \quad &\xrightarrow{\tau} \nu c{:}response.(\ host[\ c(x).\overline{print}\langle x\rangle\ ] \\
\text{on } pike@net \quad &\quad \mid host[\ \overline{finger}\langle c\rangle \mid Inet \\
&\qquad \mid Finger \mid Daytime\ ]\ ) \\
\text{expand } Finger \quad &\equiv \nu c{:}response.(\ host[\ c(x).\overline{print}\langle x\rangle\ ] \\
&\quad \mid host[\ \overline{finger}\langle c\rangle \mid Inet \\
&\qquad \mid !finger(y).\bar{y}\langle PikeUsers\rangle \mid Daytime\ ]\ ) \\
\text{communication} \quad &\xrightarrow{\tau} \nu c{:}response.(\ host[\ c(x).\overline{print}\langle x\rangle\ ] \\
\text{on } finger@host \quad &\quad \mid host[\ Inet \mid \bar{c}\langle PikeUsers\rangle \\
&\qquad \mid Finger \mid Daytime\ ]\ ) \\
\text{communication} \quad &\xrightarrow{\tau} \nu c{:}response.(\ host[\ \overline{print}\langle PikeUsers\rangle\ ] \\
\text{on } c@net \quad &\quad \mid host[Inet \mid Finger \mid Daytime\ ]\ )
\end{aligned}$$

After a sequence of internal communications at the *net* and *host* level, the first host *Carp* is ready to print the information *PikeUsers*, and host *Pike* is restored to its original configuration.

3 The target π-calculus.

The target calculus is an asynchronous π-calculus [1,2,11], with guarded recursion and name testing.

$$P \ ::= \ a(\boldsymbol{b}).P \ \mid \ \bar{a}\langle \boldsymbol{b} \rangle \ \mid \ 0 \ \mid \ (P|Q) \ \mid \ \mu X.P \ \mid \ \nu x.P \ \mid \ \mathit{if}\ x = y\ \mathit{then}\ P\ \mathit{else}\ Q$$

The absence of output prefixing and choice reflects the local area calculus. Other aspects are tuned to make the encoding as simple as possible by reducing internal transitions and avoiding inert processes. For example, we could easily use replication $!P$ rather than recursion, but this needs an extra trigger channel.

Unusually, our channels carry non-empty lists of values rather than tuples; we write $\boldsymbol{b}$ for this, with ; for list concatenation. This is because the translation multiplexes the action of several polyadic $la\pi$-channels onto a single ether, and hence a single π-channel may carry packets of different sizes. We use head/tail pattern matching on these lists to unwrap packets; in fact, testing on the head element of a list is always enough to determine its length. An alternative would be to further encode lists using standard π-calculus techniques [22], or possibly type packets with some polymorphic datatype [17].

We need to test names for both equality and inequality, and so combine these into a conditional with operational rules derived from those for matching [15].

$$\frac{P \xrightarrow{\alpha} P'}{\mathit{if}\ x = x\ \mathit{then}\ P\ \mathit{else}\ Q \xrightarrow{\alpha} P'} \qquad \frac{Q \xrightarrow{\alpha} Q'}{\mathit{if}\ x = y\ \mathit{then}\ P\ \mathit{else}\ Q \xrightarrow{\alpha} Q'} \quad x \neq y$$

With these rules we can consistently add the following convenient structural congruence:

$$(\mathit{if}\ x = x\ \mathit{then}\ P\ \mathit{else}\ Q) \ \equiv \ P\,.$$

The operational semantics of the calculus is otherwise quite standard, and we omit the details.

4 Encoding local areas

In this section we present a compositional encoding of $la\pi$-terms into the π-calculus, following the scheme outlined in the introduction. All communication is mapped into packets passing over designated ether channels; thus tuple output $\bar{a}\langle \vec{b} \rangle$ becomes list output $\bar{e}\langle a; \boldsymbol{b} \rangle$ where ether e varies according to the level of a. To keep track of which ether to use, we maintain an environment Δ mapping levels to ether names. The encoding is parameterized over this, and takes the form

$$[\![\Gamma \vdash_\ell P]\!]_\Delta$$

where P is a well-typed term of level ℓ in context Γ, and Δ assigns ethers to levels ℓ and above.

Structure:

$$\begin{aligned}
[\![\Gamma \vdash_\ell 0]\!]_\Delta &= 0 \\
[\![\Gamma \vdash_\ell P \,|\, Q]\!]_\Delta &= [\![\Gamma \vdash_\ell P]\!]_\Delta \;\mid\; [\![\Gamma \vdash_\ell Q]\!]_\Delta \\
[\![\Gamma \vdash_\ell m[P]]\!]_\Delta &= \nu e.[\![\Gamma \vdash_m P]\!]_{\Delta, m \mapsto e} \qquad e \notin fn(P) \cup cod(\Delta) \\
[\![\Gamma \vdash_\ell \nu a{:}\sigma.P]\!]_\Delta &= \nu a.[\![\Gamma, a{:}\sigma \vdash_\ell P]\!]_\Delta
\end{aligned}$$

Actions, all with $e = \Delta(m)$ where $\Gamma(a) = \vec{\sigma}@m$:

$$\begin{aligned}
[\![\Gamma \vdash_\ell \bar{a}\langle\vec{b}\rangle]\!]_\Delta &= \bar{e}\langle a; \boldsymbol{b}\rangle \\
[\![\Gamma \vdash_\ell a(\vec{b}).P]\!]_\Delta &= \mu X.e(x; \boldsymbol{b}).\mathit{if}\ x = a\ \mathit{then}\ [\![\Gamma, \boldsymbol{b}{:}\vec{\sigma} \vdash_\ell P]\!]_\Delta\ \mathit{else}\ (\bar{e}\langle x; \boldsymbol{b}\rangle \,|\, X) \\
[\![\Gamma \vdash_\ell\ !a(\vec{b}).P]\!]_\Delta &= \mu X.e(x; \boldsymbol{b}).(X \mid \mathit{if}\ x = a\ \mathit{then}\ [\![\Gamma, \boldsymbol{b}{:}\vec{\sigma} \vdash_\ell P]\!]_\Delta\ \mathit{else}\ \bar{e}\langle x; \boldsymbol{b}\rangle)
\end{aligned}$$

Fig. 5. Rules for encoding $la\pi$ into the plain π-calculus

Figure 5 presents the full encoding, with one clause for each constructor; here we go through each one individually. The null process, parallel composition, and name restriction are unchanged.

$$\begin{aligned}
[\![\Gamma \vdash_\ell 0]\!]_\Delta &= 0 \\
[\![\Gamma \vdash_\ell P \,|\, Q]\!]_\Delta &= [\![\Gamma \vdash_\ell P]\!]_\Delta \mid [\![\Gamma \vdash_\ell Q]\!]_\Delta \\
[\![\Gamma \vdash_\ell \nu a{:}\sigma.P]\!]_\Delta &= \nu a.[\![\Gamma, a{:}\sigma \vdash_\ell P]\!]_\Delta
\end{aligned}$$

To place a process in a local area, as an agent, we create a new ether name and assign it a level in the environment Δ. A side condition ensures that we do not accidentally capture any existing names when introducing the new ether.

$$[\![\Gamma \vdash_\ell m[P]]\!]_\Delta = \nu e.[\![\Gamma \vdash_m P]\!]_{\Delta, m \mapsto e} \qquad e \notin fn(P) \cup cod(\Delta)$$

Translating an output action uses the environment and the assignment of levels to ethers to find the correct ether for the output channel. It then sends both the output channel and the data for transmission over this ether name as a list.

$$[\![\Gamma \vdash_\ell \bar{a}\langle\vec{b}\rangle]\!]_\Delta = \bar{e}\langle a; \boldsymbol{b}\rangle \qquad \text{with } e = \Delta(m) \text{ where } \Gamma(a) = \vec{\sigma}@m$$

An encoded input also uses the environment and the assignment of levels to ether names to find which ether it should listen on. When it receives a packet over this ether, it tests the head of the list to see if it matches the input channel name. If it does, then the packet is meant for this input and execution continues as appropriate. If the names do not match, then this packet is meant for some other channel in the same area. The packet is resent

and the process restarts.

$$[\![\Gamma \vdash_\ell a(\vec{b}).P]\!]_\Delta = \mu X.e(x;\boldsymbol{b}).\mathit{if}\ x = a\ \mathit{then}\ [\![\Gamma, \boldsymbol{b}{:}\vec{\sigma} \vdash_\ell P]\!]_\Delta\ \mathit{else}\ (\bar{e}\langle x;\boldsymbol{b}\rangle \mid X)$$
$$\text{with } e = \Delta(m) \text{ where } \Gamma(a) = \vec{\sigma}@m.$$

Replicated input is the same, except that the process restarts whether or not the input key is correctly matched.

$$[\![\Gamma \vdash_\ell !a(\vec{b}).P]\!]_\Delta = \mu X.e(x;\boldsymbol{b}).(X \mid \mathit{if}\ x = a\ \mathit{then}\ [\![\Gamma, \boldsymbol{b}{:}\vec{\sigma} \vdash_\ell P]\!]_\Delta\ \mathit{else}\ \bar{e}\langle x;\boldsymbol{b}\rangle)$$
$$\text{with } e = \Delta(m) \text{ where } \Gamma(a) = \vec{\sigma}@m.$$

The encoding is well-defined up to structural congruence.

Proposition 4.1 *For any $la\pi$-terms P and Q, if $P \equiv Q$ then $[\![P]\!]_\Delta \equiv [\![Q]\!]_\Delta$.*

Proof. Because the encoding is compositional, it is enough to check that all of the structural axioms for $la\pi$ given at the end of §2.1 translate to valid π-calculus equivalences. All of these are immediate; the only significant case is that $\ell[\nu a{:}\sigma.P] \equiv \nu a{:}\sigma.(\ell[P])$ becomes exchange of name binders $\nu e.\nu a.[\![P]\!]_\Delta \equiv \nu a.\nu e.[\![P]\!]_\Delta$ for some ether e. □

It is worthwhile noting that the encoding uses π-calculus channels in a highly stereotyped manner. Names fall into two distinct classes: data names, like a and b, which correspond directly to $la\pi$ channels, and ether names like e. All communication involves sending data over ethers. Data names are never used as channels, while ether names are never transmitted, nor do they appear in match tests. One consequence of this is that all our terms happen to lie in the subset studied by Merro [13] as the *local* π-calculus, where names sent over channels may not be used for further communication.

In the introduction we mentioned that in general π-calculus names have a dual rôle, for identity and for communication; what happens in the translation is that each rôle is mapped to a different name.

5 Correctness of the encoding

A $la\pi$-process and its encoding behave in very similar ways, and this is preserved under reduction. Our main result is that they enjoy a form of bisimilarity on outputs, up to the translation between direct and ether-based communication.

Theorem 5.1 *For any well-typed process $\Gamma \vdash_\ell P$ in the local area π-calculus and transition $\alpha = \bar{a}\langle\vec{b}\rangle$ or $\alpha = \tau$, the following hold.*

(i) *If $\Gamma \vdash_\ell P \stackrel{\alpha}{\Longrightarrow} P'$ then $[\![\Gamma \vdash_\ell P]\!]_\Delta \stackrel{[\![\alpha]\!]_\Delta}{\Longrightarrow} [\![\Gamma \vdash_\ell P']\!]_\Delta$.*

(ii) *If $[\![\Gamma \vdash_\ell P]\!]_\Delta \stackrel{[\![\alpha]\!]_\Delta}{\Longrightarrow} Q$ then there is P' such that $\Gamma \vdash_\ell P \stackrel{\alpha}{\Longrightarrow} P'$ and $[\![\Gamma \vdash_\ell P']\!]_\Delta \equiv Q$.*

Here $[\![\tau]\!]_\Delta = \tau$ *and* $[\![\bar{a}\langle\vec{b}\rangle]\!]_\Delta = \bar{e}\langle a, \boldsymbol{b}\rangle$ *where* $e = \Delta(m)$ *for* $\Gamma(a) = \vec{\sigma}@m$.

This result says nothing about inputs: in fact, a term and its encoding generally have different input behaviour, with the translated terms being more receptive than the original. It is conventional though in asynchronous calculi to regard only output as observable, as there is no way in principle to know when an input has been received.

In the terminology of Nestmann and Pierce [16], this is an *operational correspondence* between the calculi. Although expressed in terms of weak transitions, the correspondence is in fact rather close. Transitions match exactly, except that a single internal π-transition may map to zero $la\pi$-transitions. Unfortunately this does introduce the possibility of divergence: most translated terms can perform an unbounded sequence of τ steps as they collect and return ether packets. Divergence also arises in Nestmann and Pierce's choice encoding, except that there it is inserted by design, to give a more convenient full abstraction result; their initial encoding is divergence free. In our system, divergence arises rather naturally from mechanism of ethers. We expect that replacing this with more pragmatic lists of readers and writers would lead to a divergence-free encoding, but at a cost of considerable complexity.

Theorem 5.1 follows without difficulty from the following more precise results, which characterise exactly the possible actions of encoded processes.

Lemma 5.2 *For any well-typed process* $\Gamma \vdash_\ell P$ *in the local area* π*-calculus the following hold. In each case* $Q = [\![\Gamma \vdash_\ell P]\!]_\Delta$ *and* $e = \Delta(m)$ *where* $\Gamma(a) = \vec{\sigma}@m$.

(i) *If* $\Gamma \vdash_\ell P \xrightarrow{\bar{a}\langle\vec{b}\rangle} P'$ *then* $Q \xrightarrow{\bar{e}\langle a,\boldsymbol{b}\rangle} Q'$ *where* $Q' \equiv [\![\Gamma \vdash_\ell P']\!]_\Delta$.

(ii) *If* $\Gamma \vdash_\ell P \xrightarrow{a(\vec{b})} P'$ *then* $Q \xrightarrow{e(x,\boldsymbol{b})} Q'$ *such that for any vector of names* $\vec{c}$ *we have* $Q'\{a; \boldsymbol{c}/x; \boldsymbol{b}\} \equiv [\![\Gamma' \vdash_\ell P'\{\vec{c}/\vec{b}\}]\!]_\Delta$.

(iii) *If* $\Gamma \vdash_\ell P \xrightarrow{\tau} P'$ *then* $Q \xrightarrow{\tau} Q'$ *where* $Q' = [\![\Gamma \vdash_\ell P']\!]_\Delta$.

(iv) *If* $Q \xrightarrow{\bar{e}\langle a;\boldsymbol{b}\rangle} Q'$ *then* $\Gamma \vdash_\ell P \xrightarrow{\bar{a}\langle\vec{b}\rangle} P'$ *with* $[\![\Gamma \vdash_\ell P']\!]_\Delta \equiv Q'$.

(v) *If* $Q \xrightarrow{e(x;\boldsymbol{b})} Q'$ *then either* $Q' \equiv \bar{e}\langle x; \boldsymbol{b}\rangle \,|\, Q$ *or* $\Gamma \vdash_\ell P \xrightarrow{a(\vec{b})} P'$ *where for any vector of names* $\vec{c}$ *we have* $Q'\{a; \boldsymbol{c}/x; \boldsymbol{b}\} \equiv [\![\Gamma' \vdash_\ell P'\{\vec{c}/\vec{b}\}]\!]_\Delta$.

(vi) *If* $Q \xrightarrow{\tau} Q'$ *then either* $Q \equiv Q'$ *or* $\Gamma \vdash_\ell P \xrightarrow{\tau} P'$ *with* $[\![\Gamma \vdash_\ell P']\!]_\Delta \equiv Q'$.

This makes clear the close connection between $la\pi$-transitions and ether packets. For output, the correspondence is exact: a process can perform an output if and only if its translation can. For input, recall that when an encoded process reads a packet, it tests it and if unsuitable it retransmits the packet again and continues as before. This means that an input in the encoded system may be matched by a similar input in the system it encodes or it may perform an output and revert to the original process.

This choice of two possible responses to any input action is carried over to the case of an encoded process performing a τ. This may either reflect a τ in

$$[\![\, \Gamma \vdash_{net} Carp \mid Pike \,]\!]_{\{net \mapsto n\}} \;=\; Carp' \mid Pike'$$

$$\begin{aligned}
Carp' &= \nu q.\nu c.\,(\; \bar{n}\langle pike, finger, c\rangle \\
&\qquad \mid \mu X.n(x;\boldsymbol{y}).\mathit{if}\ x = c\ \mathit{then}\ \bar{q}\langle print;\boldsymbol{y}\rangle\ \mathit{else}\ (\bar{n}\langle x;\boldsymbol{y}\rangle \mid X)\;) \\
Pike' &= \nu p.(Inet' \mid Finger' \mid Daytime') \\
Inet' &= \mu X.n(x;\boldsymbol{y}).(X \mid \mathit{if}\ x = pike\ \mathit{then}\ \bar{p}\langle\boldsymbol{y}\rangle\ \mathit{else}\ \bar{n}\langle x;\boldsymbol{y}\rangle) \\
Finger' &= \mu X.p(s,r).(X \mid \mathit{if}\ s = finger\ \mathit{then}\ \bar{n}\langle r, PikeUsers\rangle\ \mathit{else}\ \bar{p}\langle s,r\rangle) \\
Daytime' &= \mu X.p(s,r).(X \mid \mathit{if}\ s = daytime\ \mathit{then}\ \bar{n}\langle r, PikeDate\rangle\ \mathit{else}\ \bar{p}\langle s,r\rangle)
\end{aligned}$$

Ether names: n, p, q
Data names: $pike, finger, daytime, print, c, r, s$

Fig. 6. Example of processes using local areas: an Internet server dæmon

the system it encodes or it may be a rejected communication, in which case the process it reduces to is congruent to the original.

The proofs for each clause in the lemma follow a similar pattern. For clauses (i)–(iii), we break down a process into the part that performs the action and a surrounding context. Next we use the encoding rules to encode these parts. Then we show how the encoding of the part of the $la\pi$-process that performs the action can perform a matching π-action. Finally, we show that the encoding of the context allows this similar action to escape. There is a dependency, in that we must prove parts (i) and (ii) before (iii).

Clauses (iv)–(vi) are proved similarly, but in the reverse direction. First we break down Q, into the parts that perform the action and a context, then using this decomposition we characterize P, finally we show that this P can perform the required action and reduce to a process matching Q'.

The direct relationship between the behaviour of a process and its encoding make the proof much easier. In particular, there are no intermediate forms on the π-calculus side to be analysed. If there were such additional "housekeeping" steps, then we would need to enlarge the lemma to cover a one-to-many relation R_Δ between $la\pi$-processes and π-terms.

6 Encoding of the internet daemon

To illustrate how this works we encode the `inetd` example from Section 2.5. Figure 6 shows the result, which can be compared with Figure 4. The translation uses three ethers, for which we take names n, p and q to cover the network, server host *Pike* and client host *Carp* respectively. All $la\pi$-channel names like *finger* map to themselves.

Figure 7 represents graphically the behaviour of the translated system. Grey bars indicate the local ethers; compare this to the direct links of Figure 1.

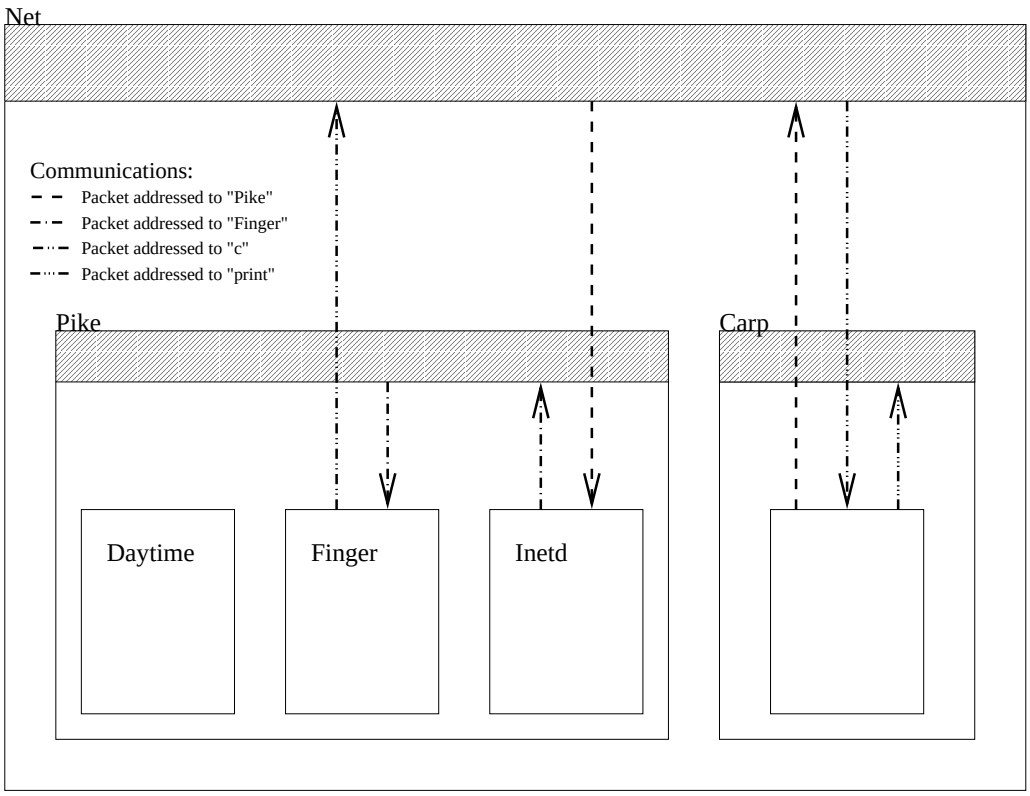


Fig. 7. Ether-based encoding of `inet` dæmon relaying `finger` service

As we expect from Lemma 5.2, reductions of the translated process closely match those of the original given earlier.

$$
\begin{aligned}
Carp' \mid Pike' \equiv\ & (\ \nu q.\nu c.(\bar{n}\langle pike, finger, c\rangle \mid \mu X.n(x;\boldsymbol{y})\ldots \\
& \mid \nu p.(Inet' \mid Finger' \mid Daytime')\ ) \\
\text{extend scope of } p,\ q \text{ and } c \quad \equiv\ & \nu p, q, c.\ (\ \bar{n}\langle pike, finger, c\rangle \mid \mu X.n(x;\boldsymbol{y})\ldots \\
& \mid Inet' \mid Finger' \mid Daytime'\ ) \\
\text{unroll } Inet' \quad \equiv\ & \nu p, q, c.\ (\ \bar{n}\langle pike, finger, c\rangle \mid \mu X.n(x;\boldsymbol{y})\ldots \\
& \mid n(x;\boldsymbol{y}).(Inet' \mid \mathit{if}\ x = pike\ \mathit{then}\ \bar{p}\langle\boldsymbol{y}\rangle\ \mathit{else}\ \bar{n}\langle x;\boldsymbol{y}\rangle) \\
& \mid Finger' \mid Daytime'\ ) \\
\text{communication of } pike \text{ over } n \quad \xrightarrow{\tau}\ & \nu p, q, c.\ (\ \mu X.n(x;\boldsymbol{y})\ldots \\
& \mid \mathit{if}\ pike = pike\ \mathit{then}\ \bar{p}\langle finger, c\rangle\ \mathit{else}\ldots \\
& \mid Inet' \mid Finger' \mid Daytime'\ ) \\
\text{apply test and unroll } Finger' \quad \equiv\ & \nu p, q, c.\ (\ \mu X.n(x;\boldsymbol{y})\ldots \\
& \mid \bar{p}\langle finger, c\rangle \\
& \mid p(s,r).(Finger' \mid \mathit{if}\ s = finger \\
& \qquad\qquad \mathit{then}\ \bar{n}\langle r, PikeUsers\rangle\ \mathit{else}\ \ldots) \\
& \mid Inet' \mid Daytime'\ )
\end{aligned}
$$

$$
\begin{array}{rl}
\text{communication} & \xrightarrow{\tau} \nu p, q, c.\,(\ \mu X.n(x;\boldsymbol{y})\ldots \\
\text{of } \textit{finger} \text{ over } p & \qquad\qquad \mid \textit{if finger} = \textit{finger then } \bar{n}\langle c, \mathit{PikeUsers}\rangle \textit{ else} \ldots \\
& \qquad\qquad \mid \mathit{Inet}' \mid \mathit{Finger}' \mid \mathit{Daytime}'\ ) \\
\text{apply test} & \equiv \nu p, q, c.\,(\ \mu X.n(x;\boldsymbol{y}).\textit{if } x = c \textit{ then } \bar{q}\langle \mathit{print};\boldsymbol{y}\rangle \textit{ else} \ldots \\
& \qquad\qquad \mid \bar{n}\langle c, \mathit{PikeUsers}\rangle \\
& \qquad\qquad \mid \mathit{Inet}' \mid \mathit{Finger}' \mid \mathit{Daytime}'\ ) \\
\text{communication} & \xrightarrow{\tau} \nu p, q, c.\,(\ \textit{if } c = c \textit{ then } \bar{q}\langle \mathit{print}, \mathit{PikeUsers}\rangle \textit{ else} \ldots \\
\text{of } c \text{ over } n & \qquad\qquad \mid \mathit{Inet}' \mid \mathit{Finger}' \mid \mathit{Daytime}'\ ) \\
\text{apply test} & \equiv \nu p, q, c.\,(\ \bar{q}\langle \mathit{print}, \mathit{PikeUsers}\rangle \\
& \qquad\qquad \mid \mathit{Inet}' \mid \mathit{Finger}' \mid \mathit{Daytime}'\ )
\end{array}
$$

Comparing the reduction in given in Section 2.5, notice how communication restricted to a local area ("communication on *finger*@*host*") is replaced by communication on a local ether ("communication of *finger* over p").

Unlike the original $la\pi$-term, other reduction sequences are possible, though they will only add extra τ-transitions. For example, the $\mathit{Daytime}'$ server may mistakenly pick up the *finger* request, but will always immediately rebroadcast it.

7 Conclusion and further work

We have encoded a notion of distributed areas and local communication into the π-calculus, by giving a translation of the local area π-calculus. At the core of this encoding is the technique of replacing communication on a channel name with communication over an ether associated with the appropriate local area.

The operational correspondence of Section 5 says that there is a close relation between the actions of processes and their translations. The next step is to build on this to investigate the degree to which the encoding preserves and reflects equivalences between processes. We would expect adequacy, but not full abstraction, as encoding local areas by ethers exposes them to probing by general π-calculus terms. For example, it is possible to eavesdrop on all top-level communications, even ones involving private names ("packet-snooping"),

Well-known names that mean different things in different places are reminiscent of dynamic binding in programming languages; that slippery concept whereby the meaning of a local variable at a program point depends on how we got there. While there seems to be no direct connection, it would be interesting to know how local areas affect the classic encoding of functions as π-calculus processes [19].

Limiting communication to local areas can be seen as a form of "security". The $la\pi$-calculus does not itself prove processes to be secure, but instead can show how particular protocols operate under imposed security constraints.

(In this sense it is about liveness, rather than safety.) We hope to use this to model aspects of Network Address Translation (NAT), a standard method for shared internet access, which is known to interact poorly with certain kinds of application.

The fixed arrangement of local areas in $la\pi$ does not lend itself to a dynamic runtime structure. There is however some flexibility: where areas appear under replication, they will be freshly created during execution; and empty areas are indistinguishable from the null process. For more general mobility, we are working on an extension of $la\pi$ with primitives for relocating areas, and an associated type system. The encoding given in the present paper does not extend to handle mobility, because it assumes that each process has direct access to the ethers for every containing level. We can suggest a solution though, using an encoding with a network of controllers. Within an area, each process communicates only through its immediate local area controller. Packets are routed by controllers to their destination, and mobile areas can be represented by reprogramming the controllers.

References

[1] Roberto Amadio, Ilaria Castellani, and Davide Sangiorgi. On bisimulations for the asynchronous π-calculus. *Theoretical Computer Science*, 195:291–324, 1998.

[2] Gérard Boudol. Asynchrony and the π-calculus. Rapport de recherche 1702, INRIA, Sophia Antipolis, 1992.

[3] Gérard Boudol, Ilaria Castellani, and Davide Sangiorgi. Observing localities. *Theoretical Computer Science*, 114:31–61, 1993.

[4] Gérard Boudol, Ilaria Castellani, and Davide Sangiorgi. A theory of processes with localities. *Formal Aspects of Computing*, 6:165–200, 1994.

[5] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Secrecy and group creation. In *CONCUR '2000: Concurrency Theory. Proceedings of the 11th International Conference*, Lecture Notes in Computer Science 1877, pages 365–379. Springer-Verlag, 2000.

[6] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structure: Proceedings of FoSSaCS '98*, Lecture Notes in Computer Science 1378, pages 140–155. Springer-Verlag, 1998.

[7] Giuseppe Castagna and Jan Vitek. Seal: A framework for secure mobile computations. In *Internet Programming Languages*, Lecture Notes in Computer Science 1686, pages 47–77. Springer-Verlag, 1999.

[8] Tom Chothia and Ian Stark. A distributed π-calculus with local areas of communication. In *Proceedings of HLCL '00: High-Level Concurrent Languages*, Electronic Notes in Theoretical Computer Science 41.2. Elsevier, 2001.

[9] Cédric Fournet, Jean-Jacques Lévy, and Alain Schmitt. An asynchronous distributed implementation for mobile ambients. In *Theoretical Computer Science: Proceedings of TCS 2000*, Lecture Notes in Computer Science 1872, pages 348–364. Springer-Verlag, August 2000.

[10] Matthew Hennessy and James Riely. Resource access control in systems of mobile agents. In *Proceedings of HLCL '98: High-Level Concurrent Languages*, Electronic Notes in Theoretical Computer Science 16.3, pages 3–17. Elsevier, 1998.

[11] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *ECOOP '91: Proceedings of the European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science 512, pages 133–147. Springer-Verlag, July 1991.

[12] IANA, the Internet Assigned Numbers Authority. Protocol numbers and assignment services: Port numbers. `http://www.iana.org/numbers.html#P`.

[13] Massimo Merro. *Locality in the π-calculus and applications to distributed objects*. PhD thesis, Ecole des Mines, France, October 2000.

[14] Robin Milner. The polyadic π-calculus — a tutorial. Technical Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, University of Edinburgh, 1991.

[15] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100:1–77, 1992.

[16] Uwe Nestmann and Benjamin C. Pierce. Decoding choice encodings. BRICS report RS-99-42, Department of Computer Science, University of Aarhus, December 1999. To appear in *Information and Computation*; a version appeared as a paper at CONCUR '96.

[17] Benjamin Pierce and Davide Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. *Journal of the ACM*, 47(5):531–584, September 2000.

[18] James Riely and Matthew Hennessy. Distributed processes and location failures. In *Automata, Languages and Programming: Proceedings of the 24th International Colloquium ICALP '97*, Lecture Notes in Computer Science 1256, pages 471–481. Springer-Verlag, 1997.

[19] Davide Sangiorgi. Lazy functions and mobile processes. Rapport de recherche 2515, INRIA, Sophia Antipolis, 1995.

[20] Davide Sangiorgi. Locality and non-interleaving semantics in calculi for mobile processes. *Theoretical Computer Science*, 155:39–83, 1996.

[21] Peter Sewell. Global/local subtyping and capability inference for a distributed π-calculus. In *Automata, Languages and Programming: Proceedings of the 25th International Colloquium ICALP 98*, Lecture Notes in Computer Science 1442. Springer-Verlag, 1998.

[22] David Turner. *The Polymorphic Pi-Calculus: Theory and Implementation.* PhD thesis, Laboratory for Foundations of Computer Science, Edinburgh University, 1996. Also published as LFCS Technical Report 345.

[23] José-Luis Vivas and Mads Dam. From higher-order π-calculus to π-calculus in the presence of static operators. In *CONCUR '98: Concurrency Theory. Proceedings of the 9th International Conference*, Lecture Notes in Computer Science 1466. Springer-Verlag, 1998.

Turing Machines, Transition Systems, and Interaction

Dina Q. Goldin [1]

Computer Science and Engineering Department, University of Connecticut, Storrs, CT 06269, USA

Scott A. Smolka [2]

Department of Computer Science, SUNY at Stony Brook Stony Brook, NY 11794-4400, USA

Peter Wegner [3]

Department of Computer Science, Brown University Providence, RI 02912, USA

Abstract

We present *Persistent Turing Machines* (PTMs), a new way of interpreting Turing-machine computation, one that is both interactive and persistent. We show that the class of PTMs is isomorphic to a very general class of effective transition systems. One may therefore conclude that the extensions to the Turing-machine model embodied in PTMs are sufficient to make Turing machines expressively equivalent to transition systems. We also define the *persistent stream language* (PSL) of a PTM and a corresponding notion of PSL-equivalence, and consider the infinite hierarchy of successively finer equivalences for PTMs over finite interaction-stream prefixes. We show that the limit of this hierarchy is strictly coarser than PSL-equivalence, a "gap" whose presence can be attributed to the fact that the transition systems corresponding to PTM computations naturally exhibit unbounded nondeterminism.

We also consider *amnesic* PTMs and a corresponding notion of equivalence based on *amnesic stream languages* (ASLs). It can be argued that amnesic stream languages are representative of the classical view of Turing-machine computation. We show that the class of ASLs is strictly contained in the class of PSLs. Furthermore, the hierarchy of PTM equivalence relations collapses for the subclass of amnesic PTMs. These results indicate that, in a stream-based setting, the extension of the Turing-machine model with persistence is a nontrivial one, and provide a formal foundation for reasoning about programming concepts such as objects with static attributes.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

1 Introduction

Several researchers have recently observed that the Turing-machine model of computation, the focus of which is on a theory of computable functions, falls short when it comes to modeling modern computing systems whose hallmarks are interaction and reactivity. For example, van Leeuwen in [LW00b] states:

> ... the classical Turing paradigm may no longer be fully appropriate to capture all features of present-day computing.

Also, Wegner[Weg98] has conjectured that interactive models of computation are more expressive than "algorithmic" ones such as Turing machines. It would therefore be interesting to see what extensions are necessary to Turing machines to capture the salient aspects of interactive computing. Moreover, it would be desirable if the alterations made to the classical model could in some sense be kept minimal.

Motivated by these goals, we investigate a new way of interpreting Turing-machine computation, one that is both interactive and persistent. In particular, we present *persistent Turing machines* (PTMs). A PTM is a nondeterministic 3-tape Turing machine that upon receiving an input token from its environment, computes for a while and then outputs the result to the environment, and this process is repeated forever. A PTM is persistent in the sense that a notion of "state" (work-tape contents) is maintained from one computation to the next.

The main results we have obtained about PTMs are the following.

- We formalize the notions of interaction and persistence in PTMs in terms of the *persistent stream language* (PSL) of a nondeterministic 3-tape Turing machine (N3TM) (Section 2). Given an N3TM M and work-tape contents w, $PSL(M, w)$ is coinductively defined to be the set of infinite sequences (interaction streams) of pairs of the form (w_i, w_o) such that each pair represents a computation performed by the N3TM in response to receiving input token w_i from the environment, producing output token w_o. Moreover, the contents of the work tape (initially w) are left intact from the previous computation upon commencing a new computation. Persistent stream languages induce a natural, stream-based notion of equivalence for PTMs.
- We then define a very general kind of effective transition system called *interactive transition systems* (ITSs), and equip ITSs with three notions of behavioral equivalence: *ITS isomorphism*, *interactive bisimulation* and *in-*

[1] Research supported in part by NSF grant IRI-9733678. Email: dqg@cse.uconn.edu
[2] Research supported in part by NSF grant CCR-9988155 and ARO grants DAAD190110003 and DAAD190110019. Email: sas@cs.sunysb.edu
[3] Email: pw@cs.brown.edu

teractive stream equivalence (Section 3). We show that ITS isomorphism refines interactive bisimulation, and interactive bisimulation refines interactive stream equivalence.

Our main result concerning ITSs is that the class of ITSs is isomorphic to the class of PTMs, thereby allowing one to view PTMs as ITSs "in disguise" (Section 4). This addresses a question heretofore left unanswered concerning the relative expressive power of Turing machines and transition systems. Till now, the emphasis has been on showing that various kinds of process algebras, with transition-system semantics, are capable of simulating Turing machines in lock-step [Bou85,dS85,BBK87,BIM88,Dar90,Vaa93]. The other direction—namely: What extensions are required of Turing machines so that they can simulate transitions systems?—is answered by our result.

- We also define an infinite hierarchy of successively finer equivalences for PTMs over finite interaction-stream prefixes and show that the limit of this hierarchy does *not* coincide with PSL-equivalence (Section 5). The presence of this "gap" can be attributed to the fact that the transition systems corresponding to PTM computations naturally exhibit unbounded nondeterminism. In contrast, it is well known that classical Turing-machine computations are nondeterministically bounded.
- Finally, we define the *amnesic stream language* (ASL) of an N3TM and a corresponding notion of amnesic PTM (Section 6). In this case, the N3TM begins each new computation with a blank work tape. Our main result about ASLs is that the class of ASLs is strictly contained in the class of PSLs. Amnesic stream languages are representative of the classical view of Turing-machine computation. One may consequently conclude that, in a stream-based setting, the extension of the Turing-machine model with persistence is a nontrivial one, and provides a formal foundation for reasoning about programming concepts such as objects with static attributes. We additionally show that ASL-equivalence coincides with the equivalence induced by considering interaction-stream prefixes of length one, the bottom of our equivalence hierarchy; and that this hierarchy collapses in the case of amnesic PTMs.

2 Persistent Turing Machines

In this section, we show how classical Turing machines can be reinterpreted as interactive computing devices. We shall consider, in fact, *non-deterministic 3-tape Turing machines* (N3TMs), each equipped with an input, work, and output tape. It is well known that N3TMs are equivalent to single-tape TMs. That is, given an N3TM M accepting some language L, there exists a single-tape TM accepting L [HU79].

The key concept we need to render Turing machines interactive is *interaction streams*: infinite sequences of token pairs of the form (w_i, w_o). Each such pair represents a computation performed by the N3TM, producing out-

put tape contents w_o, in response to w_i being placed on its input tape by the environment. Moreover, the N3TM is allowed to "remember" its previous "state" (work-tape contents) upon commencing a new computation. We shall therefore refer to such N3TMs as *persistent Turing machines* (PTMs) and to the sets of interaction streams they generate as *persistent stream languages*.

To begin our formal treatment of PTMs, we define a *macrostep* of an N3TM as a shorthand notation for a (possibly divergent) computation of a Turing machine. Our choice of terminology is inspired by the treatment of Statecharts semantics in [PS91].

Definition 2.1 *Let M be an N3TM having alphabet Σ, and let w_i, w, w' and w_o be words over Σ. We say that $\langle w_i, w\rangle \models\!\!\Rightarrow_M \langle w', w_o\rangle$* (yields in one macrostep) *if M, when started in its initial control state with w_i, w, ϵ on its input, work, and output tapes, respectively, has a halting computation that produces w_i, w', w_o as the respective contents of its input, work, and output tapes.*

Should M's computation diverge, we write $\langle w_i, w\rangle \models\!\!\Rightarrow_M \langle s_{div}, \tau\rangle$, where $s_{div}, \tau \notin \Sigma^*$ and s_{div} is a special "divergence state" such that $\langle w_i, s_{div}\rangle \models\!\!\Rightarrow_M \langle s_{div}, \tau\rangle$ for all inputs w_i.

We sometimes omit the subscript M from the macrostep notation when it is clear from the context.

We view macrosteps as transitions from one PTM "state" (encoded in the contents of the work tape) to another, an idea that is formalized in Section 4. Divergent computations of the underlying N3TM bring the PTM to the "divergence state," a special absorbing state not in Σ^* that outputs τ (in analogy with the internal τ action of CCS [Mil89]) in conjunction with the current and all subsequent inputs. Our treatment of divergence is consistent with the failures-divergence refinement model of CSP [BR85].

The contents of the input tape is not changed by a macrostep, reflecting the read-only nature of input tapes in our framework. Moreover, a macrostep begins with a blank output tape (ϵ is the empty word) reflecting a write-only semantics for output tapes. Note, however, that a macrostep may begin with a non-blank work tape, in contrast to the classical setting where the work tape is assumed to be blank at the start of computation. This convention plays an essential role in the definition of a PTM's "persistent stream language" given below (Definition 2.2).

To formally define interaction streams and persistent stream languages, fix the alphabet of an N3TM to be Σ, and let A be an enumerable set of *action tokens*. $\mathcal{S}_A$, the class of *streams over* A, is defined as follows: $\mathcal{S}_A = A \times \mathcal{S}_A$.[4] Then the class of interaction streams is given by $\mathcal{S}_{\Sigma^* \times \Sigma^* \uplus \{\tau\}}$, the members of which are pairs of the form $\langle (w_i, w_o), \sigma'\rangle$ with $(w_i, w_o) \in \Sigma^* \times (\Sigma^* \uplus \{\tau\})$ and $\sigma' \in \mathcal{S}_{\Sigma^* \times (\Sigma^* \uplus \{\tau\})}$.

[4] We have defined streams *coinductively*; see, e.g, [BM96]. This style of definition will allow us to apply coinduction as a proof technique later in the paper.

Definition 2.2 *Given an N3TM M and some $w \in \Sigma^* \uplus \{s_{div}\}$, $PSL(M,w)$ (the* persistent stream language of M with memory w*) is defined as follows:*

$$PSL(M,w) = \{\langle (w_i, w_o), \sigma' \rangle \in \mathcal{S}_{\Sigma^* \times \Sigma^* \uplus \{\tau\}} \mid \exists w' \in \Sigma^* \uplus \{s_{div}\} : \langle w_i, w \rangle \Longrightarrow_M \langle w', w_o \rangle \text{ and } \sigma' \in PSL(M,w')\}$$

$PSL(M)$, the persistent stream language *of M, is defined as $PSL(M,\epsilon)$. N3TMs M_1 and M_2 are* PSL-equivalent*, notation $M_1 =_{PSL} M_2$, if $PSL(M_1) = PSL(M_2)$. We also have that $\mathcal{PSL} = \{PSL(M) \mid M$ is an N3TM$\}$.*

Example 2.3 *Consider the N3TM M_{Latch} that outputs the first bit of the input token it received in conjunction with its* previous *interaction with the environment (except for the first interaction where it outputs a 1). $PSL(M_{Latch})$ therefore contains interaction streams of the form*

$$\{(w_1, 1), (w_2, w_1[1]), (w_3, w_2[1]), \ldots\},$$

where, in general, $w[i]$ denotes the i th bit of the string w.

For example, if the input tokens M_{Latch} receives from the environment are single bits, and the first four of these form the bit sequence 1001*, then the corresponding interaction stream $\sigma_{io} \in PSL(M_{Latch})$ would be of the form:*

$$\sigma_{io} = \{(1,1), (0,1), (0,0), (1,0), \ldots\}$$

We also consider M'_{Latch}, an unreliable version of M_{Latch}. Apart from behaving like M_{Latch}, it can nondeterministically exhibit a divergent (nonterminating) computation in conjunction with any input but the first.

For an interaction stream $\sigma \in PSL(M_{Latch})$, $PSL(M'_{Latch})$ will contain σ as well as, for all $k > 1$, a k-divergent version of σ where the computation diverges at the k th macrostep. In this case, the first $k-1$ pairs in σ remain the same, and the output tokens for all subsequent pairs are replaced by τ. For example, a 3-divergent version of σ_{io} is $\{(1,1), (0,1), (0,\tau), (1,\tau), \ldots\}$.

One might argue that the interaction between M_{Latch} and its environment is not essential; rather its behavior could be modeled by a machine that receives its entire (infinite) stream σ of input tokens prior to computation and then proceeds to output (the first bit of each element of) σ prepended with a 1. The problem with this approach is that, in general, the elements of σ are generated dynamically and therefore cannot be known in advance.

3 Interactive Transition Systems

In this section, we introduce a kind of "effective" transition system (see, for example, [Vaa93]) that we shall refer to as "interactive transition systems." An important result about interactive transition systems is that they are isomorphic to PTMs (Theorem 4.7).

Let Σ be a finite *alphabet* not containing τ, the "internal action."

Definition 3.1 *An* interactive transition system *(ITS) is a triple* $\langle S, m, r \rangle$ *where:*

- $S \subseteq \Sigma^* \uplus \{s_{div}\}$ *is the set of* states*, where* $s_{div} \notin \Sigma^*$ *is a special "divergence" state.*
- $m \subseteq S \times \Sigma^* \times S \times (\Sigma^* \uplus \{\tau\})$ *is the* transition relation*. We require that* m*, restricted to* $S \times \Sigma^* \times S \times \Sigma^*$*, is* recursive*, i.e., its interpretation as the function* $m : S \times \Sigma^* \to 2^{S \times \Sigma^*}$ *is recursively enumerable. Moreover,* m *is such that:*
  - *if* $\langle s, w_i, s_{div}, w_o \rangle \in m$*, then* $w_o = \tau$*, for all* s, w_i*; and*
  - *if* $\langle s_{div}, w_i, s, w_o \rangle \in m$*, then* $w_o = \tau$ *and* $s = s_{div}$*, for all* w_i*.*
- $r \in S$ *is the* initial state *(root).*

We use Σ to encode the states of an ITS. This is for convenience only; any effective encoding will do. Intuitively, a transition $\langle s, w_i, s', w_o \rangle$ of an ITS T means that T, while in state s and having received input string w_i from its environment, transits to state s' and outputs w_o. Moreover, such transitions are effective. Divergent computation is modeled by a τ-transition to the absorbing state s_{div}. We assume that all states in S, with the possible exception of s_{div}, are *reachable* from the root.

We now define three notions of equivalence for ITSs, each of which is successively coarser than the previous one.

Definition 3.2 *Two ITSs* $T_1 = \langle S_1, m_1, r_1 \rangle$ *and* $T_2 = \langle S_2, m_2, r_2 \rangle$ *are* isomorphic*, notation* $T_1 =_{iso} T_2$*, if there exists a bijection* ψ*:* $S_1 \to S_2$ *such that:*

(i) $\psi(r_1) = r_2$

(ii) $\forall w_i, w_o \in \Sigma^*, s, s' \in S : \langle s, w_i, s', w_o \rangle \in m_1$ *iff* $\langle \psi(s), w_i, \psi(s'), w_o \rangle \in m_2$

Definition 3.3 *Let* $T_1 = \langle S_1, m_1, r_1 \rangle$ *and* $T_2 = \langle S_2, m_2, r_2 \rangle$ *be ITSs. A relation* $\mathcal{R} \subseteq S_1 \times S_2$ *is a* (strong) interactive bisimulation *between* T_1 *and* T_2 *if it satisfies:*

(i) $r_1 \mathcal{R} r_2$

(ii) *if* $s \mathcal{R} t$ *and* $\langle s, w_i, s', w_o \rangle \in m_1$*, then there exists* $t' \in S_2$ *with* $\langle t, w_i, t', w_o \rangle \in m_2$ *and* $s' \mathcal{R} t'$*;*

(iii) *if* $s \mathcal{R} t$ *and* $\langle t, w_i, t', w_o \rangle \in m_2$*, then there exists* $s' \in S_1$ *with* $\langle s, w_i, s', w_o \rangle \in m_1$ *and* $s' \mathcal{R} t'$*.*

T_1 *and* T_2 *are* interactively bisimilar*, notation* $T_1 \sim T_2$*, if there exists an interactive bisimulation between them.*

Note that our definition of interactive bisimilarity is such that if $s \mathcal{R} t$, then s is divergent (has a τ-transition to s_{div}) if and only if t is divergent.

Definition 3.4 *Given an ITS* $T = \langle S, m, r \rangle$ *and a state* $s \in S$*,* $ISL(T(s))$ *(the* interactive stream language of T in state s*) is defined as follows:*

$$ISL(T(s)) = \{\langle (w_i, w_o), \sigma' \rangle \in \mathcal{S}_{\Sigma^* \times \Sigma^* \uplus \{\tau\}} \mid \exists s' \in S : \langle s, w_i, s', w_o \rangle \in m \ \wedge$$

$\sigma' \in ISL(T(s'))\}$

$ISL(T)$, the interactive stream language *of T, is defined as $ISL(T(r))$. Two ITSs T_1 and T_2 are* interactive stream equivalent*, notation $T_1 \approx T_2$, if $ISL(T_1) = ISL(T_2)$.*

It is straightforward to show that $=_{iso}$, $\sim$, and $\approx$ are equivalence relations.

Proposition 3.5 *$=_{iso} \subset \sim$ and $\sim \subset \approx$.*

Proof. The proof that ITS isomorphism (strictly) refines interactive bisimilarity is straightforward. The proof that interactive bisimilarity refines interactive stream equivalence is by coinduction on the structure of the interaction streams in $ISL(T_1)$.

To show that interactive bisimilarity strictly refines interactive stream equivalence, consider the following pair of ITSs over alphabet $\Sigma = \{0, 1\}$:

$T_1 = \langle \{r_1, s_1, t_1\}, , m_1, r_1 \rangle$ and $T_2 = \langle \{r_2, s_2\}, m_2, r_2 \rangle$, where
$m_1 = \{\langle r_1, 0, s_1, 1\rangle, \langle r_1, 0, t_1, 1\rangle, \langle s_1, 0, r_1, 1\rangle, \langle t_1, 1, r_1, 0\rangle\}$ and
$m_2 = \{\langle r_2, 0, s_2, 1\rangle, \langle s_2, 0, r_2, 1\rangle, \langle s_2, 1, r_2, 0\rangle\}$.

It is easy to see that $T_1 \approx T_2$ but $T_1 \not\sim T_2$. □

4 Isomorphism of ITS and PTM

In this section, we show that the class of PTMs and the class of ITSs are isomorphic. For this purpose, we assume a fixed alphabet Σ, denote the class of PTMs with alphabet Σ by $\mathcal{M}$, and denote the class of ITSs with alphabet Σ by $\mathcal{T}$.

We begin by defining the "reachable memories" of a PTM. Recalling (Definition 3.1) that the states of an ITS are assumed to be reachable from the ITS's root, reachable memories provide us with an analogous concept for PTMs.

Definition 4.1 *Let $M \in \mathcal{M}$ be a PTM with alphabet Σ. Then $reach(M)$, the* reachable memories *of M, is defined as:*

$$\begin{aligned} reach(M) = \{w \in \Sigma^* \uplus \{s_{div}\} \mid \\ \exists k \geq 0, \exists w_i^1, \ldots, w_i^k, w_o^1, \ldots, w_o^k, s^1, \ldots, s^k \in \Sigma^* \uplus \{s_{div}\} : \\ \langle w_i^1, \epsilon\rangle \models\!\!\Longrightarrow_M \langle s^1, w_o^1\rangle, \langle w_i^2, s^1\rangle \models\!\!\Longrightarrow_M \langle s^2, w_o^2\rangle, \\ \ldots, \langle w_i^k, s^{k-1}\rangle \models\!\!\Longrightarrow_M \langle s^k, w_o^k\rangle \text{ and } w = s^k\} \end{aligned}$$

As noted above, we will show that $\mathcal{M}$ and $\mathcal{T}$ are isomorphic, preserving natural equivalence relations. For $\mathcal{T}$, the relation in question is ITS isomorphism (Definition 3.2), and for $\mathcal{M}$ it will be *macrostep equivalence*, which we now define.

Definition 4.2 *Two PTMs M_1, M_2 are* macrostep equivalent*, notation $M_1 =_{ms} M_2$, if there exists a bijection ϕ: $reach(M_1) \to reach(M_2)$ such that:*

(i) $\phi(\epsilon) = \epsilon$

(ii) $\forall w_i, w_o \in \Sigma^*, s, s' \in reach(M_1)$:
$\langle w_i, s, \epsilon\rangle \models\!\!\Longrightarrow_{M_1} \langle w_i, s', w_o\rangle$ *iff* $\langle w_i, \phi(s)\rangle \models\!\!\Longrightarrow_{M_2} \langle \phi(s'), w_o\rangle$

The mapping $\xi : \mathcal{M} \to \mathcal{T}$ is given by $\xi(M) = \langle reach(M), m, \epsilon\rangle$, where $\langle s, w_i, s', w_o\rangle \in m$ iff $\langle w_i, s\rangle \models\!\!\Longrightarrow_M \langle s', w_o\rangle$. Note that $\xi(M)$ is indeed an ITS, as $reach(M)$ is enumerable, m is effective, and the set of states of $\xi(M)$ is reachable from its root. By definition, ξ is a transition-preserving isomorphism from the reachable memories of M to the states of T.

Example 4.3 *The ITSs of Figure 1 depict the image, under ξ, of the PTMs M_{Latch} and M'_{Latch} of Example 2.3. Transitions such as $(1*, 0)$ represent the infinite family of transitions where, upon receiving a bit string starting with 1 as input, the ITS outputs a 0.*

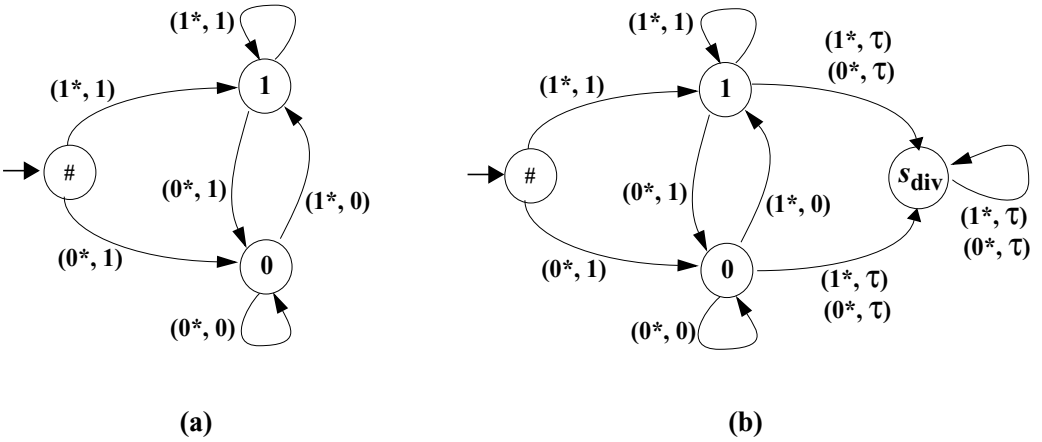


Fig. 1. (a) $\xi(M_{Latch})$ and (b) $\xi(M'_{Latch})$

It is easy to see that persistent stream languages are preserved by ξ.

Proposition 4.4 *For all $M, M' \in \mathcal{M}$, $PSL(M) = ISL(\xi(M))$ and $M =_{PSL} M'$ iff $\xi(M) \approx \xi(M')$.*

The proof uses coinduction to establish a stronger result, namely, if $\sigma \in PSL(M, w)$, for any reachable memory $w \in \Sigma^* \uplus \{s_{div}\}$ of M, then $\sigma \in ISL(T(w))$.

Proof. We prove only one direction, namely that $PSL(M) \subseteq ISL(\xi(M))$; the other direction is analogous. Let $w \in \Sigma^* \uplus \{s_{div}\}$ be a reachable memory of M and σ a stream in $PSL(M, w)$. According to Definition 2.2, there exists $w' \in \Sigma^* \uplus \{s_{div}\}$ such that $\sigma = \langle (w_i, w_o), \sigma'\rangle$ where $\langle w_i, w\rangle \models\!\!\Longrightarrow_M \langle w', w_o\rangle$ and $\sigma' \in PSL(M, w')$.

Let $T = \xi(M)$; by definition, w is a state of T. Since w' is also reachable memory of M, it is also a state of T. We prove coinductively that $\sigma \in ISL(T(w))$. By definition of ξ, $\langle w, w_i, w', w_o\rangle$ is a transition of T. By coinduction, we have that $\sigma' \in ISL(T(w'))$. Therefore, by Definition 3.4, $\sigma \in ISL(T(w))$. Since w was arbitrary, let $w = \epsilon$. It follows that for all $\sigma \in PSL(M)$, it is the case that $\sigma \in ISL(T)$. □

The following proposition shows that ξ maps equivalent PTMs to equivalent ITSs.

Proposition 4.5 *For all $M_1, M_2 \in \mathcal{M}$, $M_1 =_{ms} M_2$ iff $\xi(M_1) =_{iso} \xi(M_2)$.*

Proof. Set ψ in the definition of $=_{iso}$ (Definition 3.2) to the ϕ in the definition of $=_{ms}$ (Definition 4.2), for the $\Rightarrow$-direction of the proof, and vice versa for the $\Leftarrow$-direction of the proof. □

The following proposition shows that ξ is surjective.

Proposition 4.6 *For all $T \in \mathcal{T}$, there exists $M \in \mathcal{M}$ such that $T = \xi(M)$.*

Proof. Let $T = \langle S, m, \epsilon \rangle$. To prove the result, we exhibit a bijective mapping $\omega : S \to \Sigma^* \uplus \{s_{div}\}$ and a PTM $M \in \mathcal{M}$ such that that $\omega(r) = \epsilon$, $\omega(s_{div}) = s_{div}$ and

$$\langle s, w_i, s', w_o \rangle \in m \text{ iff } \langle w_i, \omega(s) \rangle \models\!\!\Rightarrow_M \langle \omega(s'), w_o \rangle$$

where $w_o \in \Sigma^* \uplus \{\tau\}$. Let $T_0 = \langle S_0, \Sigma, m_0, \epsilon \rangle$, where

$$S_0 = \{\omega(s) \mid s \in S\}; m_0 = \{\langle \omega(s), w_i, \omega(s'), w_o \rangle \mid \langle s, w_i, s', w_o \rangle \in m\}$$

Clearly, $\xi(M) = T$. Also, $T_0 =_{iso} T$, where ω is the desired mapping. □

The main result of this section, which essentially allows one to view persistent Turing machines and interactive transition systems as one and the same, now follows.

Theorem 4.7 *The structures $\langle \mathcal{M}, =_{ms} \rangle$ and $\langle \mathcal{T}, =_{iso} \rangle$ are isomorphic.*

Proof. It follows from Propositions 4.5 and 4.6 that ξ is a structure-preserving bijection. □

5 Equivalence Hierarchy

All stream-based notions of equivalence presented so far for PTMs are relative to infinite streams. In this section, we define equivalences over finite stream prefixes, to obtain an infinite hierarchy of equivalence relations for PTMs. We show that there is a gap between the limit of the hierarchy and PSL equivalence. When proving the existence of this gap, we also demonstrate that PTM computations exhibit unbounded nondeterminism.

We first define the family of stream prefix operators, pref_k.

Definition 5.1 *Let $\mathcal{S}_A$ be the set of streams over some set A of tokens and let $\sigma \in \mathcal{S}_A$. Then $\sigma = \langle a, \sigma' \rangle$ for some $a \in A, \sigma' \in \mathcal{S}_A$. For all $k \geq 1$, $\text{pref}_k(\sigma)$ is defined inductively as follows:*

$$\text{pref}_k(\sigma) = \begin{cases} \langle a, \epsilon \rangle & \text{if } k = 1 \\ \langle a, \text{pref}_{k-1}(\sigma') \rangle & \text{otherwise} \end{cases}$$

We next define the *k-prefix language* of a PTM M, the set of prefixes of length $\leq k$ of the interaction streams in $PSL(M)$. PTMs with the same k-prefix language are called *k-equivalent*.

Definition 5.2 *For any $k \geq 1$ and any PTM M, the* k-prefix language *of M is given by $L_k(M) = \cup_{i \leq k} \{\text{pref}_i(\sigma) \,|\, \sigma \in PSL(M)\}$. Moreover, the pair of PTMs M_1, M_2 are* k-equivalent*, notation $M_1 =_k M_2$, if $L_k(M_1) = L_k(M_2)$.*

Proposition 5.3 *For any $k \geq 1$, $(k+1)$-equivalence strictly refines k-equivalence, i.e., $=_{k+1} \subset =_k$.*

Proof. That $(k+1)$-equivalence refines k-equivalence follows from Definition 5.2. To prove that the refinement is strict, consider the sequence of PTMs ${M_{Ct}}^1, {M_{Ct}}^2, \ldots$, where, for any k, ${M_{Ct}}^k$ is the PTM with binary outputs that ignores its inputs, outputting k 1's and thereafter outputting 0's only.

Essentially, these PTMs are counters, counting off k inputs. It can be shown that for all $k \geq 1$, $L_k({M_{Ct}}^k) = L_k({M_{Ct}}^{k+1})$, but $L_{k+1}({M_{Ct}}^k) \neq L_{k+1}({M_{Ct}}^{k+1})$. This is accomplished by observing that the stream behavior of ${M_{Ct}}^k$ and ${M_{Ct}}^{k+1}$ is identical up to and including stream prefixes of length k, but $< (1,1),(1,1),\ldots,(1,1),(0,0) > \in L_{k+1}({M_{Ct}}^k) - L_{k+1}({M_{Ct}}^{k+1})$. □

Proposition 5.3 establishes an *infinite hierarchy* of stream-based equivalence relations for PTMs, the limit point of which is *∞-equivalence*.

Definition 5.4 *PTMs M_1 and M_2 are called* ∞-equivalent*, notation $M_1 =_\infty M_2$, if $L_\infty(M_1) = L_\infty(M_2)$, where $L_\infty(M) = \cup_{k \geq 1} L_k(M)$.*

Clearly, $=_\infty$ refines $=_k$, for all k. But how do $=_\infty$ and $=_1$ (the end points of the hierarchy) relate to the stream-based equivalences we defined earlier in Section 2? We consider this question in Propositions 5.5 and 6.4.

Proposition 5.5 *PSL-equivalence strictly refines ∞-equivalence, i.e., $=_{PSL} \subset =_\infty$.*

Proof. That PSL-equivalence refines ∞-equivalence follows from the definitions. To prove that the refinement is strict, we define PTMs M_{1^*} and M_{1^*0}, which ignore their inputs, and output a zero or a one with each macrostep. PTM M_{1^*} has a persistent bit b and a persistent string n representing some natural number in unary notation, both of which are initialized at the beginning of the first macrostep. In particular b is nondeterministically set to 0 or 1, and n is initialized to some number of 1's using the following loop:

```
while true do
  write a 1 on the work tape and move head to the right;
  nondeterministically choose to exit the loop or continue
od
```

M_{1^*}'s output at every macrostep is determined as follows:

```
if b = 1
```

```
then output 1;
else if n > 0
       then decrement n by 1 and output 1;
       else output 0
```

PTM M_{1^*0} behaves the same as M_{1^*} except that b is always initialized to 0. Now note that the L_∞-languages of these PTMs is the same, consisting of all finite sequences of pairs of the form:

$$\{(in_1, out_1), \ldots, (in_k, out_k)\},$$

where the $in_j \in \Sigma^*$ are input tokens and out_j is 1 for the first j pairs in the sequence (for some $j \leq k$) and 0 for the rest (if any). However, $PSL(M_{1^*}) \neq PSL(M_{1^*0})$; in particular, the stream $\{(1,1),(1,1),...\} \in PSL(M_{1^*}) - PSL(M_{1^*0})$. □

The ITS corresponding to M_{1^*0}, i.e., $\xi(M_{1^*0})$, is depicted in Figure 2 and demonstrates that PTMs are capable of exhibiting *unbounded nondeterminism* in a natural way. Though the number of 1's at the beginning of each interaction stream is always finite, it is unbounded. The ITS for M_{1^*} is similar.

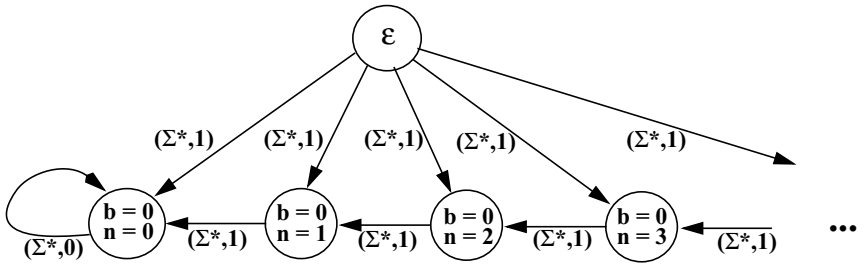


Fig. 2. The ITS corresponding to the PTM M_{1^*0}.

6 Amnesic Stream Computation

In this section, we present the notion of *amnesic* stream computation, where the contents of the persistent work tape is erased (or simply ignored) at each macrostep. We show that *amnesic stream languages* (ASLs) constitute a proper subset of PSLs, and that ASL equivalence coincides with the bottom of the infinite equivalence hierarchy presented in Section 5.

The amnesic stream language for an N3TM is defined similarly to the NT3M's persistent stream language (Definition 2.2). However, each computation of the N3TM begins with a blank work tape; i.e., the N3TM "forgets" the state it was in when the previous computation ended. As before, fix the alphabet of an N3TM to be Σ.

Definition 6.1 *Given an N3TM M, $ASL(M)$ (the* amnesic stream language of *M) is defined as follows:*

$$ASL(M) = \{\langle (w_i, w_o), \sigma' \rangle \in \mathcal{S}_{\Sigma^* \times \Sigma^* \uplus \{\tau\}} \mid \exists w' \in \Sigma^* : \langle w_i, \epsilon \rangle \models\!\!\Longrightarrow_M \langle w', w_o \rangle \wedge \sigma' \in ASL(M)\}.$$

N3TMs M_1 and M_2 are ASL-equivalent*, notation $M_1 =_{ASL} M_2$, if $ASL(M_1) = ASL(M_2)$. We also have that $\mathcal{ASL} = \{ASL(M) \mid M$ is an N3TM$\}$.*

Example 6.2 *The interaction streams in $ASL(M_{Latch})$ (Example 2.3) are of the form $\{(w_1, 1), (w_2, 1), \ldots\}$.*

It is also possible to define amnesic stream languages for ITSs, and a PTM's amnesic stream language would be preserved by the mapping ξ defined in Section 4. The interaction streams contained in the amnesic stream language of an ITS T would be constructed by always returning to T's initial state before moving on to the next input-output token pair in the stream. Although amnesia makes sense for Turing machines—in the classical, non-interactive setting, every Turing-machine computation commences with a blank work tape—its applicability to transition systems is questionable.

The following proposition is used in the proofs of Propositions 6.4 and 6.5.

Proposition 6.3 *Given an N3TM M, let $L(M)$ be defined as follows:*

$$\{(w_i, w_o) \in \Sigma^* \times \Sigma^* \uplus \{\tau\} \mid \exists w' \in \Sigma^* : \langle w_i, \epsilon \rangle \models\!\!\Longrightarrow_M \langle w', w_o \rangle\}$$

Then, $ASL(M) = \mathcal{S}_{L(M)}$, the set of all streams over $L(M)$.

Proposition 6.4 $=_{ASL} \; = \; =_1$

Proposition 6.5 $\mathcal{ASL} \subset \mathcal{PSL}$

Proof. To prove $\mathcal{ASL} \subseteq \mathcal{PSL}$, it suffices to show that, given an N3TM M, we can construct an N3TM M' such that $PSL(M') = ASL(M)$. The construction is as follows:

> M' always starts its computation by erasing the contents of its work tape and moving the work-tape head back to beginning of tape; it then proceeds just like M.

From Definitions 2.2 and 6.1, it follows that $PSL(M') = ASL(M)$.

To prove that the inclusion of $\mathcal{ASL}$ in $\mathcal{PSL}$ is strict, we refer to M_{Latch} and $\sigma_{io} \in PSL(M_{Latch})$ defined in Example 2.3 to show that there does not exist an N3TM M such that $ASL(M) = PSL(M_{Latch})$. Assume such an N3TM M exists; then, $\sigma_{io} \in ASL(M)$. Therefore, by Proposition 6.3, $(0, 0)$, the third element of σ_{io}, is in $L(M)$. This in turn implies that there are interaction streams in $ASL(M)$ starting with $(0, 0)$. But no stream in $PSL(M_{Latch})$ can start with $(0, 0)$, leading to a contradiction. Therefore, no such M exists. □

We say that a PTM M is *amnesic* if $PSL(M) \in \mathcal{ASL}$.

Example 6.6 *M_{Latch} is not amnesic. Neither are the M_{Ct} PTMs defined in the proof of Proposition 5.3. Even though they ignore their input values, these*

PTMs remember the number *of inputs they have consumed, and are therefore not amnesic.*

On the other hand, some recently proposed extensions of Turing-machine computation to the stream setting do not capture persistence. For example, the squaring machine *of [PR98, Figure 1], which repeatedly accepts an integer n from its environment and outputs n^2, is clearly amnesic.*

Most of the results obtained in this paper rely on the persistence of PTMs; that is, they do not hold if we restrict our attention to amnesic PTMs. For example, the whole equivalence hierarchy collapses in this case.

Proposition 6.7 *For any pair of amnesic PTMs M_1 and M_2, $M_1 =_{ASL} M_2$ iff $M_1 =_{PSL} M_2$.*

7 Related Work

The notions of persistency and interaction embodied in PTMs and ITSs can be found in one form or another in various models of reactive computation including dataflow and related areas [KM77,PS88,RT90,PSS90,KP93,BE94], process calculi [Mil89,MPW92], synchronous languages [Har87,BG92], finite/pushdown automata over infinite words [EHRS00,BCMS01], interaction games [Abr00], reactive modules [AH99], and I/O automata[Lyn96]. The main difference between these approaches and our own is that our focus is on the relationship between Turing machines and transition systems, and on the effects of unbounded nondeterminism on the equivalence hierarchy for PTMs. The other approaches tend to emphasize issues such as correctness and programming, and the computability of a transition step is often left implicit. Moreover, these models of computation are typically purely functional in nature, and, therefore, the notion of persistency or "state" present in PTMs is absent.

Persistency, however, can be captured in dataflow models by "feedback loops" and in process calculi by explicitly modeling the data store. For example, PTM M_{Latch} of Example 2.3 can be modeled in a dataflow setting by the stream transformer $f(s) = (1, s)$, which can be evaluated lazily/on-the-fly. M_{Latch} is a simple example of a PTM: its history dependence only goes back one interaction in time and PTMs are in general capable of expressing history dependence of an unbounded nature. It would therefore be interesting to determine whether stream transformers can encode the behavior of *all* PTMs.

Persistent Turing machines formalize the notion of *Sequential Interaction Machines* introduced in earlier papers by the first and third authors, including [Weg98,GST00]. A major emphasis of this body of work is to show how such a computational framework can be used as a basis for modeling various forms of interactive computing, such as object-oriented, agent-based, and dynamical systems. PTMs also formalize the notion of *embedded components*, according to the criteria presented in [LW00a].

An alternative approach to extending the Turing-machine model to interactive computation is captured by the *Interactive Turing Machines with Advice* (ITMAs) of [LW00b]. Like PTMs, ITMAs are persistent, interactive, and stream-based. Additionally, they incorporate several features aimed at capturing "practical" computing devices, including multiple input/output ports and advice, a kind of oracle modeling hardware and software upgrades. In contrast, PTMs, which have single input and output tapes and do not appeal to oracles, represent a minimal extension to the classical Turing-machine model (persistence of the work tape) needed to attain transition-system expressiveness.

8 Conclusions

We have presented Persistent Turing Machines (PTMs), a stream-based extension of the Turing-machine model with appropriate notions of interaction and persistency. A number of expressiveness results concerning PTMs have been presented, including the expressive equivalence of PTMs and interactive transition systems; the strict inclusion of the set $\mathcal{ASL}$ of amnesic stream languages in the set $\mathcal{PSL}$ of persistent stream languages (showing that "persistence pays"); the "gap" between the limit of the equivalence hierarchy based on finite interaction-stream prefixes and PSL-equivalence; and the collapse of the equivalence hierarchy in the case of amnesic PTMs.

Our results are summarized in Figure 3.

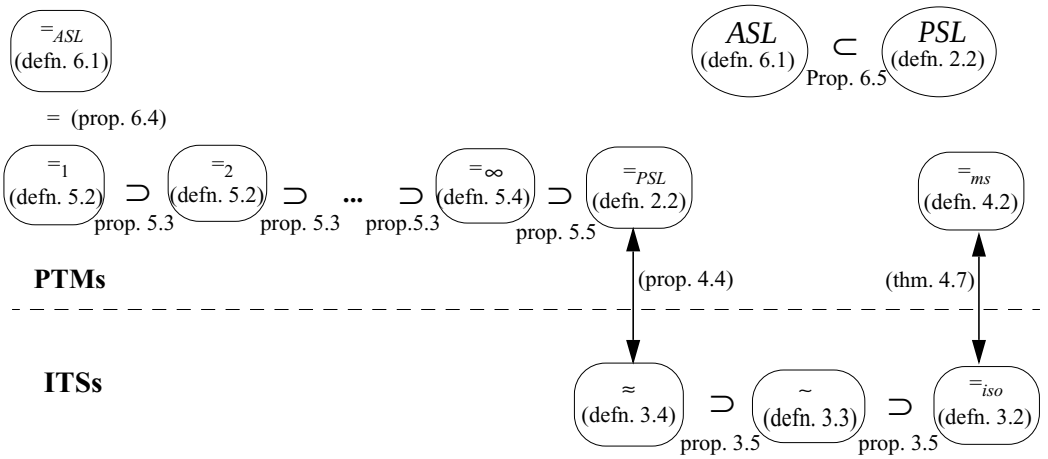


Fig. 3. Summary of results.

It should be noted that, by virtue of our isomorphism result, every equivalence defined for PTMs can be carried over to ITSs, and vice versa. For example, a relation can be defined for PTMs that is analogous to ITS bisimulation; by contrast, bisimulation makes no sense in the traditional Turing-machine context. On the other hand, the transition-system analog of ASL equivalence makes little sense, even though it is natural in the traditional (i.e., non-stream-based) Turing-machine world.

As ongoing work, we are developing a model of PTM computation where PTMs execute concurrently and communicate with each other through their input and output tapes. We conjecture that concurrent PTMs are more expressive than sequential ones in terms of the stream languages they produce. We are also interested in developing a "weak" theory of persistent stream languages and interactive bisimulation in which divergent computation (τ-transitions) is abstracted away.

Acknowledgement

We would like to thank Peter Fejer and the anonymous referees for their valuable comments, and Paul Attie for bringing to our attention the phenomenon of unbounded nondeterminism in PTMs.

References

[Abr00] S. Abramsky. Concurrent interaction games. In J. Davies, A. W. Roscoe, and J. Woodcock, editors, *Millenial Perspectives in Computer Science*, pages 1–12. Palgrave, 2000.

[AH99] R. Alur and T. A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15:7–48, 1999.

[BBK87] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. On the consistency of Koomen's fair abstraction rule. *Theoretical Computer Science*, 51(1/2):129–176, 1987.

[BCMS01] O. Burkart, D. Caucal, F. Moller, and B. Steffen. Verification on infinite structures. In *Handbook of Process Algebra*. Elsevier, 2001.

[BE94] A. Bucciarelli and T. Ehrhard. Sequentiality in an extensional framework. *Information and Computation*, 110(2):265–296, 1994.

[BG92] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.

[BIM88] B. S. Bloom, S. Istrail, and A. R. Meyer. Bisimulation can't be traced. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, 1988.

[BM96] J. Barwise and L. Moss. *Vicious Circles*. CSLI Lecture Notes #60. Cambridge University Press, 1996.

[Bou85] G. Boudol. Notes on algebraic calculi of processes. In K. Apt, editor, *Logics and Models of Concurrent Systems*, pages 261–303. LNCS, Springer-Verlag, 1985.

[BR85] S. D. Brookes and A. W. Roscoe. An improved failures model for communicating sequential processes. In *Proceedings NSF-SERC Seminar on Concurrency*. Springer-Verlag, 1985.

[Dar90] P. Darondeau. Concurrency and computability. In I. Guessarian, editor, *Semantics of Systems of Concurrent Processes, Proceedings LITP Spring School on Theoretical Computer Science*, La Roche Posay, France, 1990. LNCS **469**, Springer-Verlag.

[dS85] R. de Simone. Higher-level synchronizing devices in meije-sccs. *Theoretical Computer Science*, 37:245–267, 1985.

[EHRS00] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Proc. of CAV'2000*, pages 232–247. number 1855 in Lecture Notes in Computer Science, Springer-Verlag, 2000.

[GST00] D. Goldin, S. Srinivasa, and B. Thalheim. Information systems = databases + interaction: Towards principles of information system design. In *Proceedings of ER 2000*, Salt Lake City, Utah, 2000.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA, 1979.

[KM77] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In *Proc. of the IFIP Congress 77*. North-Holland, 1977.

[KP93] G. Kahn and Gordon D. Plotkin. Concrete domains. *Theoretical Computer Science*, 121(1&2):187–277, 1993.

[LW00a] J. van Leeuwen and J. Wiedermann. On algorithms and interaction. In *Proc. of MFCS'2000*, Bratislava, Slovak Republic, August 2000. Springer-Verlag.

[LW00b] J. van Leeuwen and J. Wiedermann. The Turing machine paradigm in contemporary computing. In B. Enquist and W. Schmidt, editors, *Mathematics Unlimited - 2001 and Beyond*. LNCS, Springer-Verlag, 2000.

[Lyn96] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[Mil89] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Parts I and II. *Information and Computation*, 100, 1992.

[Pat90] M.S. Paterson, editor. *Automata, Languages and Programming (ICALP '90)*, volume 443 of *Lecture Notes in Computer Science*, Warwick, England, July 1990. Springer-Verlag.

[PR98] M. Prasse and P. Rittgen. Why Church's thesis still holds: Some notes on Peter Wegner's tracts on interaction and computability. *Computer Journal*, 41(6), 1998.

[PS88] P. Panangaden and E. W. Stark. Computations, residuals, and the power of indeterminancy. In *Proceedings of 15th ICALP*, pages 439–454. Springer-Verlag, Lecture Notes in Computer Science, Vol. 317, 1988.

[PS91] A. Pnueli and M. Shalev. What is in a step: On the semantics of Statecharts. In *Theoretical Aspects of Computer Software*, number 526 in Lecture Notes in Computer Science, pages 244–264, 1991.

[PSS90] P. Panangaden, V. Shanbhogue, and E. W. Stark. Stability and sequentiality in dataflow networks. In Paterson [Pat90], pages 308–321.

[RT90] A. M. Rabinovich and B. A. Trakhtenbrot. Communication among relations. In Paterson [Pat90], pages 294–307.

[Vaa93] F. W. Vaandrager. Expressiveness results for process algebras. Technical Report CS-R9301, Centrum voor Wiskunde en Informatica, Amsterdam, 1993.

[Weg98] P. Wegner. Interactive foundations of computing. *Theoretical Computer Science*, 192, February 1998.

Recent BRICS Notes Series Publications

NS-01-6 **Luca Aceto and Prakash Panangaden, editors.** ***Preliminary Proceedings of the 8th International Workshop on Expressiveness in Concurrency, EXPRESS '01,*** **(Aalborg, Denmark, August 20, 2001), August 2001. vi+139 pp.**

NS-01-5 **Flavio Corradini and Walter Vogler, editors.** ***Preliminary Proceedings of the 2nd International Workshop on Models for Time-Critical Systems, MTCS '01,*** **(Aalborg, Denmark, August 25, 2001), August 2001. vi+ 127pp.**

NS-01-4 **Ed Brinksma and Jan Tretmans, editors.** ***Proceedings of the Workshop on Formal Approaches to Testing of Software, FATES '01,*** **(Aalborg, Denmark, August 25, 2001), August 2001. viii+156 pp.**

NS-01-3 **Martin Hofmann, editor.** ***Proceedings of the 3rd International Workshop on Implicit Computational Complexity, ICC '01,*** **(Aarhus, Denmark, May 20–21, 2001), May 2001. vi+144 pp.**

NS-01-2 **Stephen Brookes and Michael Mislove, editors.** ***Preliminary Proceedings of the 17th Annual Conference on Mathematical Foundations of Programming Semantics, MFPS '01,*** **(Aarhus, Denmark, May 24–27, 2001), May 2001. viii+279 pp.**

NS-01-1 **Nils Klarlund and Anders Møller.** ***MONA Version 1.4 — User Manual*****. January 2001. 83 pp.**

NS-00-8 **Anders Møller and Michael I. Schwartzbach.** ***The XML Revolution*****. December 2000. 149 pp.**

NS-00-7 **Nils Klarlund, Anders Møller, and Michael I. Schwartzbach.** ***Document Structure Description 1.0*****. December 2000. 40 pp.**

NS-00-6 **Peter D. Mosses and Hermano Perrelli de Moura, editors.** ***Proceedings of the Third International Workshop on Action Semantics, AS 2000,*** **(Recife, Brazil, May 15–16, 2000), August 2000. viii+148 pp.**

NS-00-5 **Claus Brabrand.** ***<bigwig> Version 1.3 — Tutorial*****. September 2000. ii+92 pp.**

NS-00-4 **Claus Brabrand.** ***<bigwig> Version 1.3 — Reference Manual*****. September 2000. ii+56 pp.**