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 $A \cup \{\texttt{call}, \texttt{wait}, \texttt{end}\}$.

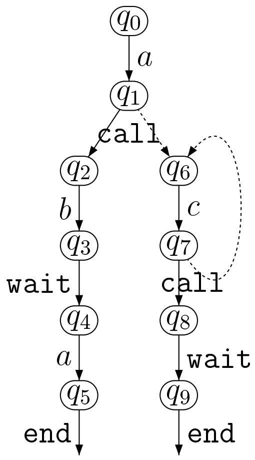

Fig. 1. A scheme

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:

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

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

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

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

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

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

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

Example 2.1 $(q_0, \emptyset) \stackrel{a}{\rightarrow} (q_1, \emptyset) \xrightarrow{\texttt{call}} (q_2, (q_6, \emptyset)) \stackrel{c}{\rightarrow} (q_2, (q_7, \emptyset)) \xrightarrow{\texttt{call}} (q_2, (q_8, (q_6, \emptyset))) \stackrel{b}{\rightarrow} (q_3, (q_8, (q_6, \emptyset))) \cdots$ 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.

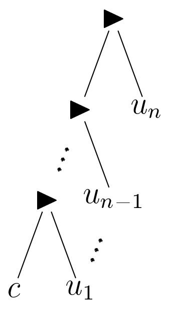

Fig. 2. $c \blacktriangleright^n (u_1, \ldots, u_n)$

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 $q \to q' \blacktriangleright t$. $\square$

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

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

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 $q \in Q$,

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

end: if $\Lambda(q) = \mathtt{end}$ then Δ_G contains $q \xrightarrow{\mathtt{end}} 0, 0$, (De)

call: if $\Lambda(q) = (\mathtt{call}, q', q'')$ then Δ_G contains $q \xrightarrow{\mathtt{call}} q', q''$, (Dc)

wait: if $\Lambda(q) = (\mathtt{wait}, q')$ then Δ_G contains $q \xrightarrow{\mathtt{wait}} q', 0$. (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.

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

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

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

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

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, $\mathcal{Q} = \{p, p' \dots\}$ 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 $\mathcal{B}^+(\{1, \dots, n\} \times \mathcal{Q})$. 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 $\langle t_1, \dots, t_n \rangle$ 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 $\mathcal{F}^{\times n} \overset{\text{def}}{=} (\mathcal{F} \cup \{\perp\})^n$ where the arity of $f_1 \dots f_n$ is the maximum of the arities of the f_i, assuming $\perp$ has arity 0.

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

We say a n-ary relation $R \subseteq T^n_{\mathcal{F}}$ 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 $\rightarrow^*$ and $\xrightarrow{!}{}^*$ between RPA terms are recognizable:

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

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

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

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 $f, g \in (\mathcal{F} \cup \{\perp\})$) this means $\delta(p', fg)$ is false, and (3) we quantify over all $q \in Q$, all $c \in \mathit{Const}$, and all $f \in (\mathcal{F} \cup \{\perp\})$.

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

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

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

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

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

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

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

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

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

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

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

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

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 same smooth and general automata-theoretic reasoning.

Reachability sets. For any recognizable language L, the sets $\mathit{Pre}^*(L) \stackrel{\text{def}}{=} \{u \mid u \xrightarrow{*} v \text{ for some } v \in L\}$ and $\mathit{Post}^*(L) \stackrel{\text{def}}{=} \{u \mid v \xrightarrow{*} u \text{ for some } v \in L\}$ 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 ${\mathit{Pre}^*}_C(L) \stackrel{\text{def}}{=} \{u \mid u \xrightarrow{\sigma} v \text{ for some } v \in L \text{ and } \sigma \in C\}$ and ${\mathit{Post}^*}_C(L) \stackrel{\text{def}}{=} \{u \mid v \xrightarrow{\sigma} u \text{ for some } v \in L \text{ and } \sigma \in C\}$ where $C \subseteq \mathit{Act}^*$ is a constraint on acceptable labels for reachability. Not all regular $C \subseteq \mathit{Act}^*$ 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 $\mathit{Mod}(\varphi)$ 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 $\mathit{Pre}^*(L)$ for recognizable L. But with recognizability of $\xrightarrow{*}$, we get a simple model checking algorithm for the full transition logic [5], i.e. the first-order logic $FO(\rightarrow, \xrightarrow{*})$. 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 $u \equiv_{\mathcal{S}} u' \xrightarrow{\alpha} v' \equiv_{\mathcal{S}} v$ for some u', v'. We adopt the usual extensions $u \stackrel{\sigma}{\Rightarrow} v$ (for $\sigma \in \mathit{Act}^*$) 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', $s \xrightarrow{*} s'$ 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 $MSO(\rightarrow)$, the monadic second-order logic with $\rightarrow$ as the only predicate, is undecidable over RPA terms.

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

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 $\searrow\!\!\!\searrow$ 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 $\searrow\!\!\!\searrow^*$ for the reflexive-transitive closure of $\searrow\!\!\!\searrow$, we deduce that $\searrow\!\!\!\searrow^*$ coincide with $\searrow^* \curvearrowright^*$ and then with $\searrow \curvearrowright^=$, where $\curvearrowright^=$ denotes $\curvearrowright \cup Id$.

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

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

Proof. By induction on v and case analysis. See Appendix A.7. $\square$

Hence, by Newman's Lemma, $\searrow\!\!\!\searrow$ 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 $u \rightleftarrows v \searrow\!\!\!\searrow w$, then $u \swarrow\!\!\!\swarrow v' \overset{*}{\rightleftarrows} w$ for some v'.*

Proof. By induction on u and case analysis. See Appendix A.8. $\square$

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

8.1.3 Convertibility

Finally, we combine simplifications and permutations in $\leftrightsquigarrow$, a relation defined as $\left(\mathrel{\searrow\!\!\searrow} \cup \rightleftarrows \cup \mathrel{\swarrow\!\!\swarrow}\right)^*$. 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.* $u \mathrel{\searrow\!\!\searrow}^* u' \overset{*}{\rightleftarrows} v' \mathrel{{}^*\!\!\swarrow\!\!\swarrow} v$,
(c) $u{\downarrow} \overset{*}{\rightleftarrows} v{\downarrow}$.

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

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

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

Proof. The $(\Leftarrow)$ 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 $(\Rightarrow)$ direction is proved in Appendix A.9. $\square$

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

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

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

Lemma 8.6 *If* L *is regular then* $[L]_{\swarrow\!\!\swarrow}$ *is regular. Furthermore, from a tree automaton* $\mathcal{A}$ *recognizing* L, *one can build in polynomial-time a tree automaton* $\mathcal{A}'$ *for* $[L]_{\swarrow\!\!\swarrow}$ *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 $t \overset{*}{\mapsto} r_p^q$ iff t is some $0 \blacktriangleright^n (t_1, \ldots, t_n)$ and $p \blacktriangleright^n (t_1, \ldots, t_n) \overset{*}{\mapsto} q$

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 $u \xrightarrow{\alpha} u'$ implies that $v \xrightarrow{\alpha} v'$ 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 $t \xrightarrow{\sigma} u'$ 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 $u \xrightarrow{*} w$ 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 $L' = Pre^*(L) = \{t \mid t \xrightarrow{*} t' \in L\}$ 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 $u \rightarrow$.
($\Leftarrow$) 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 $u_2 \rightarrow$ by ind. hyp. and then $u \rightarrow$ by R3-R4.
2. if $State(u_2) = \emptyset$ then $State(u_1) \neq \emptyset$, $u_1 \rightarrow$ by ind. hyp., and then $u \rightarrow$ 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 $u \blacktriangleright q \xrightarrow{n-1} t \rightarrow v \blacktriangleright q$. 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 $t_2 = q$. $t \rightarrow v \blacktriangleright q$ 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 $t \rightarrow^{*} t'$, 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 $t \rightarrow t'$). Hence $v \xrightarrow{!}{}^{*} v'$ and $t \rightarrow^{*} t'$.

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

A.4 Proof of Lemma 3.7

($\Leftarrow$): one gets $v \blacktriangleright t \stackrel{!}{\to}^* v \blacktriangleright t'$ by R3-R4, and follows with $v \blacktriangleright t' \to^* v' \blacktriangleright t'$ by R5 if $v \stackrel{!}{\to}^* v'$, or by R5-R6 if $v \to^* v'$ and $t' \not\to$.
($\Rightarrow$): we have either (a) $v \blacktriangleright t \stackrel{!}{\to}^* v' \blacktriangleright t'$ or (b) $v \blacktriangleright t \to^* v_1 \blacktriangleright t_1 \stackrel{?}{\to} v_2 \blacktriangleright t_2 \to^* v' \blacktriangleright t'$. 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 $u \Searrow w'$.

- If $v \searrow w$ 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 $u \searrow \rightleftarrows w$.

  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 $u \searrow v' = c \blacktriangleright^n (w_1, \ldots, w_{i-2}, w_i, \ldots, w_{i+m-1}, w_{i-1}, w_m, \ldots, w_n)$ works since $v' \overset{*}{\rightleftarrows} w$ with m uses of rule (P1). The second case is similar.

- If $v \searrow w$ 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. $w'_i \searrow w_i$. 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 $w''_i \rightleftarrows w'_i \searrow w_i$ yields $w''_i \searrow v'' \overset{*}{\rightleftarrows} w_i$ for some v'', and we deduce $u \searrow v' \overset{*}{\rightleftarrows} w$ 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 $u \searrow u'$ (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) 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.