Nordic Journal of Computing

AN AUTOMATA-THEORETIC APPROACH TO THE REACHABILITY ANALYSIS OF RPPS SYSTEMS

A. LABROUE and Ph. SCHNOEBELEN
Lab. Spécification & Vérification
ENS de Cachan & CNRS UMR 8643
61, av. Pdt. Wilson, 94235 Cachan Cedex France
{labroue|phs}@lsv.ens-cachan.fr

Abstract. We show how the reachability analysis of RPPS systems can be tackled with the tree-automata techniques proposed by Lugiez and Schnoebelen for PA. This approach requires that we express the states of RPPS systems in RPA, a tailor-made process rewrite system where reachability is a relation recognizable by finite tree-automata. Two outcomes of this study are (1) an NP algorithm for reachability in RPPS systems, and (2) a simple decision procedure for a large class of reachability problems in RPA systems.

CR Classification: F.3.1 Specifying and Verifying and Reasoning about Programs.

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

1. Introduction

This article is concerned with the verification of RPPS systems (for *Recursive Parallel Program Schemes*), an abstract model introduced by Kouchnarenko and Schnoebelen Kouchnarenko and Schnoebelen [1997, 1997] for the control flow of programming languages with fork/join parallelism and a limited "`wait`" facility for synchronization. The RPPS model is related to Petri nets, the PA process algebra, and Mayr's PRS Mayr [2000]. The reachability analysis of such models has important applications in verification since they are well-suited to the static analysis of programming languages with parallel constructs Esparza and Knoop [1999, 2000].

While RPPS systems can be seen as some kind of Petri nets with recursively nested markings (the viewpoint adopted in Kouchnarenko and Schnoebelen [1997, 1997]), we argue that it is worthwhile to see them as an infinite-state *process algebra* (or process rewrite system). The verification of infinite-state process algebra has been very active in recent years (see Bukart *et al.* [2001] 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

Received . Revised , , .

process algebras (with quite different expressive power) obtained by simple syntactic restrictions on the allowed rewrite rules (see Moller [1996] and Mayr [2000]).

Tree automata. Recently, Lugiez and Schnoebelen showed how reachability problems for PA – a process algebra allowing recursive definitions mixing sequential and parallel composition, without synchronization Baeten and Weijland [1990]. could be solved simply and elegantly via *tree-automata* techniques: the reachability sets for PA processes are recognizable (or "regular") tree languages Lugiez and Schnoebelen [2002]. Beyond the use of tree-automata, this 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 compatible with the tree-automata approach.

The tree-automata approach to PA is further developed in Lugiez and Schnoebelen [2000] where it is shown that the reachability relation between PA processes is an effectively recognizable relation. An important corollary is the decidability of the first-order reachability logic over PA.

Our contribution. We show that the tree-automata approach is able to cope with RPPS systems. There are three main results in the article. 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. Finally, we show how reachability between RPPS markings can be reduced to reachability questions between RPA terms, yielding a direct automata-theoretic algorithm. As a corollary, we obtain a proof of NP-completeness for reachability between RPPS markings. Its "`wait`" facility makes RPPS (and RPA) more expressive than PA (see Appendix A.1) so that the behavior of parallel programs can be modeled more faithfully with RPPS systems, e.g. RPPS can encode some aspects of the controlled AND-parallelism essential in concurrent or parallel Logic Programming Tanaka *et al.* [1986]. Mayr's PRS is more expressive than RPPS, but the reachability analysis of PRS systems is a difficult task.

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 Aceto *et al.* [2001]) rules without losing the recognizability theorem we aim at. PA is not expressive enough for this task. This explains some of the more awkward aspects of RPA.

Related works. Previous decidability results on RPPS Kouchnarenko and Schnoebelen [1997, 1997] relied on more ad-hoc tableaux methods or the well-structure of RPPS Finkel and Schnoebelen [2001]. These results were less general than what we offer in section 7.

The use of regular string languages for handling infinite sets of config-

urations in symbolic model checking of infinite state systems has recently been called "*Regular Model Checking*" Bouajjani *et al.* [2000]. Note that this framework only requires that *immediate* successors (or predecessors) of a regular set of states form a regular set. Sometimes the transitive closure of loops can be handled, but this is still weaker than being able to compute the set of *iterated* successors (or predecessors), or the reachability relation itself.

There exist several other systems for which the reachability relation is recognizable: it is semilinear for BPP Esparza [1997], definable in the additive theory of reals for timed automata Comon and Jurski [1999], a recognizable relation between words for some string rewrite systems Caucal [2000] including pushdown processes (see Kupferman and Vardi [2000] 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.

As explained in Lugiez and Schnoebelen [2002, section 5.3], our approach has some connections with ground tree rewrite systems (TRS), a computational model where reachability is recognizable Dauchet and Tison [1990]. However, there exist differences between the two models: while process rewrite systems such as PA or RPA can be seen as ground TRS with simple left-hand sides, these systems limit rule applicability by additional context conditions – the semantics of sequential composition requires that a term t in $u.t$ cannot be rewritten if u is not terminated – that are not allowed in ground TRSs. Rational tree relations Raoult [1997] are another formalism in which our result could be stated, but they involve more complex constructions than the automata constructions we use. Finally, an anonymous referee suggested a way to encode RPA in Dauchet and Tison's Ground Tree Transducers Dauchet and Tison [1990] : this would give us a (variant of) theorem 2 for free, but the translation for RPA to RPPS (sections 8 and 9) would have to be reinvestigated with this new encoding.

Plan of the article. 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 programs with fork/join parallelism and a restricted "`wait`" facility for synchronization: we refer the reader to Kouchnarenko and Schnoebelen [1997, 1997] for motivations and examples. Here we only present the formal model and compare it with PA in Appendix A.1. For simplicity, we did not add the usual labels on the transitions.

2.1 The structure of RPPS systems

An *RP scheme* (or just "a scheme") is a labeled graph $G = \langle Q, E, \Lambda, q_{\text{init}} \rangle$ where $Q = \{q, q', \ldots\}$ is a finite set of *nodes*, $E \subseteq Q \times Q$ is a set of *edges*, $q_{\text{init}} \in Q$ is the *start node*, and $\Lambda : Q \to \{\texttt{std}, \texttt{wait}, \texttt{end}\} \cup Q$ labels the nodes.

Fig. 2.1 displays G_{exmp}, an example scheme. In such graphical representations we add an incoming arrow to the start node, and omit the label `std` (leaving *standard nodes* blank). When $\Lambda(q) = q'$ we say q is a *call node* and, instead of just q', its graphical representation carries `call` q' with a dashed arrow pointing to q'. Finally, we require that no edge leaves from an *end node*.

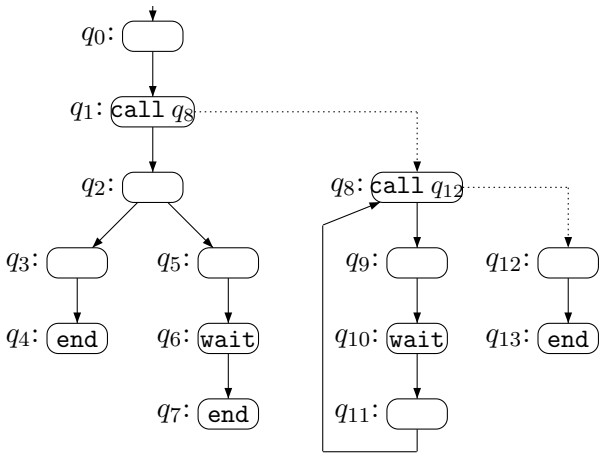


Fig. 2.1: G_{exmp}, an example RP scheme.

A scheme like G_{exmp} is the control graph of a system with fork/join parallelism: control move from one node to another following the edges of the scheme until possible termination at an `end` node. Traversing `call` nodes gives rise to parallel (sub)processes that evolve independently. The only possibility for synchronization is at `wait` nodes, where a process cannot move further until all its subprocesses have terminated. Note the asymmetry here: the behavior of subprocesses cannot be influenced by their father (the process which forked them) while the termination of the subprocesses can enable some steps of the father process.

2.2 Behavioral semantics

Formally, the behavioral semantics of RPPS systems is defined by associating a transition system $\mathcal{M}_G$ with a scheme G. A state of $\mathcal{M}_G$ is a multiset of nodes (denoting the current control states of the concurrent processes) organized with a father-son relationship (relating a process with the father process that spawned it via a `call` instruction).

Formally, the set of *hierarchical states* (also, "markings", or "states") of a system $G = \langle Q, E, \Lambda, q_{\text{init}} \rangle$ 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 $M(G)$. 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 $|\{(q_i, s_i) \mid i = 1, \ldots\}| \stackrel{\text{def}}{=} \sum_{i=1,\ldots}(1 + |s_i|)$.

An alternative, equivalent, way of seeing hierarchical states is as forests of trees labeled with nodes of G and where the ordering of subtrees is not relevant. Following Kouchnarenko and Schnoebelen [1997, 1997], we adopt the "nested multiset" viewpoint because we make heavy use of set operations like $s \cup s'$.

The intuitive meaning is that a hierarchical state of the form $s = \{(q_1, s_1), \ldots, (q_n, s_n)\}$ has n completely independent concurrent activities (or processes). One such process, say (q_i, s_i), is currently in state/node q_i and its family of children are collectively in state s_i.

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

std: if q is a standard node, and (q, q') is an edge in G, then $(q, s) \rightarrow (q', s)$ for all s, (Ga)

end: if q is an end node, then $(q, s) \rightarrow s$ for all s, (Ge)

call: if q is a call node, labeled with `call` q'', and (q, q') is an edge in G, then $(q, s) \rightarrow (q', s + (q'', \emptyset))$ for all s, (Gc)

wait: if q is a wait node, and (q, q') is an edge in G, then $(q, \emptyset) \rightarrow (q', \emptyset)$, (Gw)

paral1: if $s \rightarrow s'$ then $s + s'' \rightarrow s' + s''$ for all s'', (Gp1)

paral2: if $s \rightarrow s'$ then $(q, s) \rightarrow (q, s')$ for all $q \in Q$. (Gp2)

Rules **paral1** and **paral2** for parallelism express that any activity $s \rightarrow s'$ 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 leave a `wait` state only 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 the transition system $\langle M(G), \rightarrow, s_0 \rangle$ where the initial state is $s_0 \stackrel{\text{def}}{=} (q_0, \emptyset)$.

EXAMPLE 1. $(q_0, \emptyset) \rightarrow (q_1, \emptyset) \rightarrow (q_2, (q_8, \emptyset)) \rightarrow (q_2, (q_9, (q_{12}, \emptyset))) \rightarrow (q_2, (q_{10}, (q_{12}, \emptyset))) \rightarrow (q_2, (q_{10}, (q_{13}, \emptyset))) \rightarrow (q_2, (q_{10}, \emptyset)) \cdots$ is an execution sequence of the system $\mathcal{M}_{G_{\text{exmp}}}$ associated with the scheme of Fig. 2.1.

3. The process algebra RPA

In this section we define RPA, a process algebra designed to encode RPPS schemes.

3.1 RPA terms

Let $Q = \{q, \ldots\}$ be a finite set of *process constants* that does not contain the special symbol *Nil* and that is partitioned as $Q_{\rm nd} \cup Q_{\rm d}$ of *non-delayable* constants (those in $Q_{\rm nd}$) and *delayable* ones (those in $Q_{\rm d}$).

From Q we define $\mathit{Const} \stackrel{\rm def}{=} Q \cup \{\mathit{Nil}\}$, ranged over by $c, c', \ldots$, and T_Q, the set of *RPA terms*, or just "terms", ranged over by $t, u, v, \ldots$, given by the following grammar:

$$t, u ::= t \blacktriangleright u \mid c \mid c' \cdots$$

where $c, c', \ldots$ are all elements of *Const*. As will appear, $t \blacktriangleright u$ is a form of "t in parallel with u" where u has higher priority than some parts of t and may delay them.

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

RPA terms are binary trees where the left- and right-hand sides do not play the same role, 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. 3.1).

This motivates the convenient abbreviation "$c \blacktriangleright^n (u_1, \ldots, u_n)$", defined inductively by

$$\begin{aligned} c \blacktriangleright^n (u_1, \ldots, u_n) &\stackrel{\rm def}{=} \big(c \blacktriangleright^{n-1} (u_1, \ldots, u_{n-1})\big) \blacktriangleright u_n \quad \text{for } n > 0, \\ c \blacktriangleright^0 () &\stackrel{\rm def}{=} c. \end{aligned}$$

Note that the "$\blacktriangleright^n$" abbreviation is only defined (and meaningful) with a $c \in \mathit{Const}$ in the left-hand side.

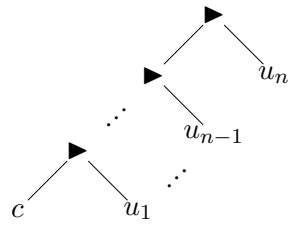


Fig. 3.1: $c \blacktriangleright^n (u_1, \ldots, u_n)$

3.2 Transitions between RPA terms

A (guarded) RPA *declaration* is a finite set $\Delta \subseteq Q \times Const \times T_Q$ of *process rewrite rules*, written $\{q_i \rightarrow_\Delta c_i, t_i \mid i = 1, \ldots, n\}$. 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.

A declaration Δ defines two transition relations $\xrightarrow{\text{nd}} \subseteq T_Q \times T_Q$ and $\xrightarrow{\text{d}} \subseteq T_Q \times T_Q$. A step of the form $t \xrightarrow{\text{d}} u$ can be delayed by parallel components with higher priority whereas a step of the form $t \xrightarrow{\text{nd}} u$ cannot be delayed.

Formally, the possible steps are given by the following SOS rules:

$$\text{R1}\ \frac{}{q \xrightarrow{\text{nd}} c \blacktriangleright t}\ \text{if } (q \rightarrow_\Delta c, t) \in \Delta \text{ and } q \in Q_{\text{nd}} \qquad \text{R3}\ \frac{t \xrightarrow{\text{nd}} t'}{u \blacktriangleright t \xrightarrow{\text{nd}} u \blacktriangleright t'}$$

$$\text{R2}\ \frac{}{q \xrightarrow{\text{d}} c \blacktriangleright t}\ \text{if } (q \rightarrow_\Delta c, t) \in \Delta \text{ and } q \in Q_{\text{d}} \qquad \text{R4}\ \frac{t \xrightarrow{\text{d}} t'}{u \blacktriangleright t \xrightarrow{\text{nd}} u \blacktriangleright t'}$$

$$\text{R5}\ \frac{t \xrightarrow{\text{nd}} t'}{t \blacktriangleright u \xrightarrow{\text{nd}} t' \blacktriangleright u} \qquad \text{R6}\ \frac{t \xrightarrow{\text{d}} t'}{t \blacktriangleright u \xrightarrow{\text{d}} t' \blacktriangleright u}\ \text{if } State(u) = \emptyset$$

Observe that a rule in Δ gives rise to a delayable or non-delayable step according to whether the left-hand side is from Q_{d} or Q_{nd}. Then the nd label on steps propagates upwards (rules R3 and R5), while the d label propagates upwards from a left-subterm *after a test is made on the right-subterm* (rule R6), or is transformed into a nd from a right-subterm (rule R4).

We write $u \rightarrow v$ when $u \xrightarrow{\text{nd}} v$ or $u \xrightarrow{\text{d}} v$. For $n \in \mathbb{N}$, we let $\xrightarrow{n}$ and $\xrightarrow{\text{nd},n}$ denote respectively the iterated relations $(\rightarrow)^n$ and $(\xrightarrow{\text{nd}})^n$. Similarly $\xrightarrow{*}$ and $\xrightarrow{\text{nd},*}$ denote the reflexive-transitive closures of $\rightarrow$ and $\xrightarrow{\text{nd}}$ respectively. As usual, $u \rightarrow$ and $u \not\rightarrow$ mean respectively that $u \rightarrow v$ for some v (resp. for no v).

3.3 Decomposition properties for RPA steps

The following key *decomposition* lemmas show how a compound step requires the existence of smaller steps. These results are the basis of our main automata-theoretic construction in section 6.

LEMMA 1. *$u \rightarrow$ iff $State(u) \neq \emptyset$.*

PROOF. ($\Rightarrow$): by induction on the derivation of $u \rightarrow$.
($\Leftarrow$): by structural induction on u. If $u = Nil$ then $State(u) = \emptyset$. If $u = q \in Q$ then Δ has at least one rule $q \rightarrow q', v$ and rule R1-R2 entails $u \rightarrow$. If u is some $u_1 \blacktriangleright u_2$, then either $State(u_2) \neq \emptyset$ or $State(u_1) \neq \emptyset$. If $State(u_2) \neq \emptyset$

then $u_2 \to$ by ind. hyp. and then $u \to$ by R3-R4. Otherwise $u_1 \to$ (by ind. hyp.) and then $u \to$ by R5-R6. Observe that, since $State(u_2) = \emptyset$, the condition for rule R6 is met. □

LEMMA 2. *If $u \to v$ then $|v| > |u|$.*

PROOF. By induction on the derivation of $u \to v$. □

LEMMA 3. *$c \xrightarrow{*} c'$ iff $c = c'$.*

PROOF. By Lemma 2, $c \xrightarrow{n} c'$ entails $n = 0$. □

LEMMA 4. *If $u \blacktriangleright v \to w$ then w has the form $u' \blacktriangleright v'$ and either $u \to u'$ and $v = v'$, or $v \to v'$ and $u = u'$.*

PROOF. A step from $u \blacktriangleright v$ can only be deduced with a rule in the R3-R6 range. In all cases the premises of the rule provide what the lemma expects. □

LEMMA 5. *$v \blacktriangleright t \xrightarrow{\text{nd},*} v' \blacktriangleright t'$ iff $v \xrightarrow{\text{nd},*} v'$ and $t \xrightarrow{*} t'$.*

PROOF. ($\Leftarrow$): From $v \xrightarrow{\text{nd},*} v'$ and $t \xrightarrow{*} t'$, one shows $v \blacktriangleright t \xrightarrow{\text{nd},*} v \blacktriangleright t'$ using R3-R4 and $v \blacktriangleright t' \xrightarrow{\text{nd},*} v' \blacktriangleright t'$ using R5.
($\Rightarrow$): We assume $v \blacktriangleright t \xrightarrow{\text{nd},n} v' \blacktriangleright t'$ and prove the result by induction on n. The case $n = 0$ is obvious. If now $v \blacktriangleright t \xrightarrow{\text{nd},1} u \xrightarrow{\text{nd},n-1} v' \blacktriangleright t'$, then u must be some $v'' \blacktriangleright t''$ (Lemma 4) and the ind. hyp. entails $v'' \xrightarrow{\text{nd},*} v'$ and $t'' \xrightarrow{*} t'$. The step $v \blacktriangleright t \xrightarrow{\text{nd}} v'' \blacktriangleright t''$ was obtained by R3, R4 or R5, so that $v \xrightarrow{\text{nd}} v''$ and $t = t''$ (if R5), or $v = v''$ and $t \to t''$ (if R3 or R4). Hence $v \xrightarrow{\text{nd},*} v'$ and $t \xrightarrow{*} t'$. □

LEMMA 6. *$u \xrightarrow{\text{nd},*} v$ iff for all $t \in T_Q$, $u \blacktriangleright t \xrightarrow{*} v \blacktriangleright t$.*

PROOF. Direct from lemma 5. □

LEMMA 7. *$v \blacktriangleright t \xrightarrow{*} v' \blacktriangleright t'$ iff $t \xrightarrow{*} t'$ and* $\begin{cases} t' \not\to \text{ and } v \xrightarrow{*} v', \\ \text{or } v \xrightarrow{\text{nd},*} v'. \end{cases}$

PROOF. ($\Leftarrow$): one gets $v \blacktriangleright t \xrightarrow{\text{nd},*} v \blacktriangleright t'$ by R3-R4, and follows with $v \blacktriangleright t' \to^* v' \blacktriangleright t'$ by R5 if $v \xrightarrow{\text{nd},*} v'$, or by R5-R6 if $v \xrightarrow{*} v'$ and $t' \not\to$.
($\Rightarrow$): we have either (a) $v \blacktriangleright t \xrightarrow{\text{nd},*} v' \blacktriangleright t'$ or (b) $v \blacktriangleright t \xrightarrow{*} v_1 \blacktriangleright t_1 \xrightarrow{\text{d}} v_2 \blacktriangleright t_2 \xrightarrow{*} v' \blacktriangleright t'$. In case (a), Lemma 5 concludes. In case (b), rule R6 requires $State(t_1) = \emptyset$ so that $t_1 \not\to$ (Lemma 1). It follows that $t' = t_1$ and $t' \not\to$. □

LEMMA 8. *$q \xrightarrow{*} v \blacktriangleright t$ iff there exists a rule $(q \rightarrow_\Delta c, u)$ in Δ s.t. $u \xrightarrow{*} t$, and* $\begin{cases} t \not\rightarrow \text{ and } c \xrightarrow{*} v, \\ \text{or } c \xrightarrow{\text{nd},*} v. \end{cases}$

PROOF. ($\Rightarrow$): if $q \xrightarrow{n} v \blacktriangleright t$ then $n > 0$ and the first step must be some $q \rightarrow c \blacktriangleright u$ obtained by R1-R2 via some $q \rightarrow_\Delta c, u$ in Δ. Then $c \blacktriangleright u \xrightarrow{n-1} v \blacktriangleright t$ and Lemma 7 concludes.
($\Leftarrow$): obvious by combining R1-R2 and Lemma 7. □

LEMMA 9. *$q \xrightarrow{\text{nd},*} v \blacktriangleright t$ iff $q \in Q_{\text{nd}}$ and there exists a rule $q \rightarrow_\Delta c, u$ in Δ s.t. $u \xrightarrow{*} t$ and $c \xrightarrow{\text{nd},*} v$.*

PROOF. (Idea) This extends Lemma 5 in the same way that Lemma 8 extends Lemma 7. □

4. Embedding RPPS schemes into RPA

The behavior of an RPPS scheme can be faithfully encoded in RPA.

Assume $G = \langle Q, E, \Lambda, q_{\text{init}} \rangle$ is a scheme. With G we associate an RPA system where nodes are used as process constants: Q_d contains all wait nodes from G and Q_{nd} contains the other nodes.

The set of rules Δ_G is obtained from Λ. Formally, for any $q \in Q$,

action: if q is a standard or a wait node and (q, q') is an edge in E, then Δ_G contains $q \rightarrow q', Nil$, (Da)

end: if q is an end node, then Δ_G contains $q \rightarrow Nil, Nil$, (De)

call: if q is a call node, labeled with `call` q'', and (q, q') is an edge in E, then Δ_G contains $q \rightarrow q', q''$. (Dc)

We associate a hierarchical state $\mathcal{S}(t)$ with any term $t \in T_Q$ and, reciprocally, an RPA 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

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

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

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

where Eq. (T) for $\mathcal{T}(s)$ requires that one picks some ordering of the elements of the multiset s. Independently of what ordering is chosen, $\mathcal{S}$ and $\mathcal{T}$ behave like an abstraction-concretization pair:

LEMMA 10. *For all $s \in M(G)$, $\mathcal{S}(\mathcal{T}(s)) = s$.*

PROOF. By structural induction on s. Assume $s = \{(q_1, s_1), \ldots, (q_n, s_n)\}$.

$$\begin{array}{rll} \mathcal{S}(\mathcal{T}(s)) & = \mathcal{S}(Nil \blacktriangleright^n (q_1 \blacktriangleright \mathcal{T}(s_1), \ldots, q_n \blacktriangleright \mathcal{T}(s_n))) & \text{by (T)} \\ & = \mathcal{S}(q_1 \blacktriangleright \mathcal{T}(s_1)) + \cdots + \mathcal{S}(q_n \blacktriangleright \mathcal{T}(s_n)) & \text{by (S1)} \\ & = (q_1, \mathcal{S}(\mathcal{T}(s_1))) + \cdots + (q_n, \mathcal{S}(\mathcal{T}(s_n))) & \text{by (S2)} \\ & = (q_1, s_1) + \cdots + (q_n, s_n) & \text{by ind. hyp.} \\ & = s. & \end{array}$$

□

RPA terms denoting the same hierarchical marking can be seen as equivalent, and we define $t \equiv_{\mathcal{S}} u \stackrel{\text{def}}{\Leftrightarrow} \mathcal{S}(t) = \mathcal{S}(u)$. We write $[u]$ for the equivalence class of u w.r.t. $\equiv_{\mathcal{S}}$, and $T_{\equiv_{\mathcal{S}}}$ for the set of equivalence classes of T_Q w.r.t. $\equiv_{\mathcal{S}}$. Note that $\equiv_{\mathcal{S}}$ is not a congruence: $(Nil \blacktriangleright u) \equiv_{\mathcal{S}} u$ whereas $(Nil \blacktriangleright u) \blacktriangleright v \not\equiv_{\mathcal{S}} u \blacktriangleright v$.

It is now possible to state how steps between RPA terms are related to steps between RPPS hierarchical states:

THEOREM 1. (CORRECTNESS OF THE EMBEDDING) *The mapping $\mathcal{S}$ from RPA terms to hierarchical states is a bisimulation between $\mathcal{M}_G$ and the transition system $\langle T_Q, \rightarrow \rangle$.*

This result relies on the following two Propositions, both proved in the Appendix.

PROPOSITION 1. *If $u \rightarrow v$ is an RPA step, then $\mathcal{S}(u) \rightarrow \mathcal{S}(v)$ is a step in $\mathcal{M}_G$.*

PROPOSITION 2. *If $\mathcal{S}(u) = s$ and $s \rightarrow s'$ is a step in $\mathcal{M}_G$, then $u \rightarrow v$ for some v s.t. $\mathcal{S}(v) = s'$.*

5. Tree languages and tree automata

Here we recall the classical tree-automata notions we need. More details can be found in Comon *et al.* [1997–99] (Vardi [1997] and Chandra *et al.* [1981] for alternating automata).

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 $\mathcal{F}_0 = \{a, b\}$, $\mathcal{F}_1 = \{h\}$ and $\mathcal{F}_2 = \{g\}$, $T_{\mathcal{F}}$ contains trees like a, $g(a, b)$ and $g(h(g(h(b), a)), b)$. A *tree language* is any subset L of $T_{\mathcal{F}}$. Note that we only consider languages of *finite* trees.

5.2 Tree automata

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

A rule $(q, f, q_1, \ldots, q_n) \in \delta$ is written $q \mapsto f(q_1, \ldots, q_n)$, in accordance with the "top-down" viewpoint on tree automata (or, equivalently, the view that tree automata are regular tree grammars).

Rules in δ are used to rewrite *mixed trees* from $T_{\mathcal{F} \cup \mathcal{Q}}$, or trees where control states can be used as extra constant symbols. Sequences of rewrite steps of the form $t_0 \mapsto t_1 \mapsto t_2 \mapsto \ldots t_n$ are called *derivations*, and we write $t \overset{*}{\mapsto} t'$ when there exists a derivation between two mixed trees.

EXAMPLE 2. Let us consider the tree automaton $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \delta \rangle$ with $\mathcal{F}$ as in the example above, $\mathcal{Q} \overset{\text{def}}{=} \{p_0, p_1\}$, and δ containing the following set of eight rules:

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

The following derivation

$$p_0 \mapsto g(p_1, p_1) \mapsto g(h(p_1), p_1) \mapsto g(h(p_1), a) \mapsto g(h(b), a)$$

shows that $p_0 \overset{*}{\mapsto} g(h(b), a)$.

When $p \overset{*}{\mapsto} t$ for a control state $p \in \mathcal{Q}$ and a tree $t \in T_{\mathcal{F}}$, we say that t is recognized (or accepted) by state p. For $p \in \mathcal{Q}$, $L(p)$ denotes $\{t \mid p \overset{*}{\mapsto} t\}$. When $\mathcal{A}$ is further equipped with a set $F \subseteq \mathcal{Q}$ of *accepting states*, $L(\mathcal{A})$, the tree language recognized by $\mathcal{A}$, is defined by $L(\mathcal{A}) \overset{\text{def}}{=} \bigcup_{p \in F} L(p)$.

EXAMPLE 3. We continue Example 2. Since $p_0 \overset{*}{\mapsto} g(h(b), a)$, we have $g(h(b), a) \in L(p_0)$. In fact, one easily sees that $L(p_0)$ and $L(p_1)$ are the sets of all $t \in T_{\mathcal{F}}$ having an even (respectively, odd) number of leaves.

A tree language $L \subseteq T_{\mathcal{F}}$ is *recognizable* (also, *regular*) iff there exists a tree automaton $\mathcal{A}$ such that L is $L(\mathcal{A})$. It is well known that recognizable tree languages are closed under boolean operations, morphisms, *etc.*

5.3 Alternating tree automata

Alternating automata generalize standard automata Chandra *et al.* [1981]. Formally, an *alternating* tree automaton is a tuple $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, F, \delta \rangle$ where now δ – called the *table* of the automaton – 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, $\mathcal{B}^+(X) = \{\theta, \ldots\}$

is the set of positive boolean formulae over X. For a set $X = \{x, y, \ldots\}$, $\mathcal{B}^+(X)$ is given by the following grammar:

$$\theta ::= \theta' \vee \theta'' \mid \theta' \wedge \theta'' \mid \mathit{true} \mid \mathit{false} \mid x \mid y \mid \ \ldots$$

For example, δ may contain expressions like $\delta(p, g) = (1, p_1) \vee ((1, p_2) \wedge (2, p_3) \wedge (2, p_4))$. The meaning of such a "rule" is that a term t of the form $g(t_1, t_2)$ is recognized by state p iff (t_1 is recognized by p_1, or t_1 by p_2 and t_2 by p_4).

Formally, the classical definition of when a term t is recognized by a state p, written $t \models p$, is by induction on the structure of t, via an inductive definition of when a tuple $t_1, \ldots, t_n$ is recognized by a formula θ over $\{1, \ldots, n\} \times \mathcal{Q}$:

$$\begin{array}{lll}
f(t_1, \ldots, t_n) & \models p & \text{iff } t_1, \ldots, t_n \models \delta(p, f), \\
t_1, \ldots, t_n & \models (i, p) & \text{iff } t_i \models p, \\
t_1, \ldots, t_n & \models \theta \wedge \theta' & \text{iff } t_1, \ldots, t_n \models \theta \text{ and } t_1, \ldots, t_n \models \theta', \\
t_1, \ldots, t_n & \models \theta \vee \theta' & \text{iff } t_1, \ldots, t_n \models \theta \text{ or } t_1, \ldots, t_n \models \theta', \\
t_1, \ldots, t_n & \models \mathit{true} & \text{always}, \\
t_1, \ldots, t_n & \models \mathit{false} & \text{never}.
\end{array}$$

Example 4. We define an alternating tree automaton (with $\mathcal{F}$ as above) that recognizes the set L of all terms where every g is above at least one a. We use two control states: p_0 that recognizes L, and p_a that recognizes terms with at least one a. Writing the table for p_a is easy:

$$\begin{array}{ll}
\delta(p_a, a) = \mathit{true}, & \delta(p_a, h) = (1, p_a), \\
\delta(p_a, b) = \mathit{false}, & \delta(p_a, g) = (1, p_a) \vee (2, p_a).
\end{array}$$

For p_0, the table is

$$\begin{array}{ll}
\delta(p_0, a) = \mathit{true}, & \delta(p_0, h) = (1, p_0), \\
\delta(p_0, b) = \mathit{true}, & \delta(p_0, g) = (1, p_0) \wedge (2, p_0) \wedge \big((1, p_a) \vee (2, p_a)\big).
\end{array}$$

The meaning of $\delta(p_0, g)$ is simply "a term $g(t_1, t_2)$ is in L iff both t_1 and t_2 are in L, and one of them contains an 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 tree languages recognized by alternating tree automata is exactly the class of tree languages recognized by non-alternating tree automata.

The benefits of using alternating automata is that complex constraints are sometimes easier to describe via alternating automata (as the previous example illustrates). In some cases, the description can be exponentially more succinct with alternating automata than with standard non-deterministic automata Chandra *et al.* [1981]. The price to pay is that usual operations on automata can be computationally more complex. In this article we use alternating automata because they conveniently express the constraints on trees we rely on.

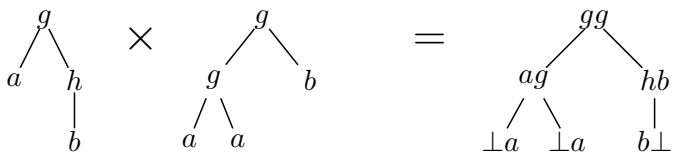
5.4 Recognizable relations on trees

We follow Comon *et al.* [1997–99] and Lugiez and Schnoebelen [2000]. A tuple $\langle t_1, \ldots, t_n \rangle$ of n trees from $T_{\mathcal{F}}$ can be encoded as a single tree, denoted $t_1 \times \cdots \times t_n$, on a product alphabet $\mathcal{F}^{\times n} \stackrel{\text{def}}{=} (\mathcal{F} \cup \{\bot\})^n$. In $\mathcal{F}^{\times n}$, the arity of the symbol $(f_1, \ldots, f_n)$, or shortly $f_1 \ldots f_n$, is the maximum of the arities of the f_i, assuming $\bot$ has arity 0. Formally

$$\prod_{i=1}^{m} f_i(t_{i,1}, \ldots, t_{i,n_i}) \stackrel{\text{def}}{=} f_1 \ldots f_m \Big(\prod_{i=1}^{m} t_{i,1},\ \ldots,\ \prod_{i=1}^{m} t_{i,k} \Big)$$

assuming that the arity of $f_1 \ldots f_m$ is $k = \max(n_1, \ldots, n_m)$, and letting $t_{i,j}$ stand for $\bot$ when $n_i < j \leq k$.

All this is better understood via an example: the product $g(a, h(b)) \times g(g(a, a), b)$ is $gg(ag(\bot a, \bot a), hb(b\bot))$ as illustrated in the following picture:



When tuples of terms can be encoded as single terms, there is a natural definition for recognizable relations:

Definition 1. *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 recognizable tree language (seen as a subset of $T_{\mathcal{F}^{\times n}}$).*

Beyond the usual boolean operations, it is well-known (and easy to see) that recognizable relations are closer under projections and *cylindrifications* – these are canonical inverses of projections. A cylindrification of some $R \subseteq T_{\mathcal{F}}^n$ is the largest $R' \subseteq T_{\mathcal{F}}^m$ whose n-dimensional projection is R – . Furthermore, the corresponding automata constructions are simple (and linear-time) Comon *et al.* [1997–99].

6. Recognizability of the reachability relation for RPA

We designed the RPA algebra with the aim of characterizing behavioral properties in automata-theoretic terms. Indeed T_Q, the set of RPA terms, is $T_{\mathcal{F}}$ for $\mathcal{F}$ given by $\mathcal{F}_2 = \{\blacktriangleright\}$ and $\mathcal{F}_0 = \mathit{Const}$. An example of what we aim at is given by the following proposition:

Proposition 3. *The set $L_{\text{term}} \stackrel{def}{=} \{u \in T_Q \mid u \not\rightarrow\}$ of* terminated terms *is recognizable.*

PROOF. $u \not\to$ iff $State(u) = \emptyset$ (Lemma 1). Thus the following automaton with an unique accepting state $p_\downarrow$ and the following table

$$\delta(p_\downarrow, Nil) = true, \qquad \delta(p_\downarrow, q) = false, \qquad \delta(p_\downarrow, \blacktriangleright) = (1, p_\downarrow) \wedge (2, p_\downarrow) \quad (6.1)$$

recognizes L_{term}. □

The main result of this article is that the reachability relations $\xrightarrow{*}$ and $\xrightarrow{\text{nd},*}$ between RPA terms are recognizable (in the sense of Definition 1).

To prove this we define the alternating automaton $\mathcal{A}_{\xrightarrow{*}} = \langle \mathcal{F}, \mathcal{Q}, \delta \rangle$ 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 $|\mathcal{Q}|$, the number of control states, is in $O(|\Delta|)$).

$\mathcal{A}_{\xrightarrow{*}}$ recognizes pairs $\langle u, v \rangle$ of terms (encoded as products $u \times v$). Here we define the table δ with the following conventions:

- when, for some $p' \in \mathcal{Q}$ and some $f, g \in \mathcal{F} \cup \{\bot\}$, $\delta(p', fg)$ is not explicitly defined, this means that $\delta(p', fg)$ is *false*, and
- the definition implicitly ranges over all $q \in Q$, all $c \in Const$, and all $f \in \mathcal{F} \cup \{\bot\}$.

We start with the table for $p_\downarrow$, a simple extension of (6.1):

$$\delta(p_\downarrow, fNil) = true, \qquad \delta(p_\downarrow, fq) = false, \qquad \delta(p_\downarrow, f\,\blacktriangleright) = (1, p_\downarrow) \wedge (2, p_\downarrow). \quad (6.2)$$

We go on with the table for p and $\bar{p}$:

$$\delta(p, NilNil) = \delta(\bar{p}, NilNil) = true \quad (6.3)$$

$$\delta(p, qq') = \delta(\bar{p}, qq') = \begin{cases} true \text{ if } q = q', \\ false \text{ otherwise} \end{cases} \quad (6.4)$$

$$\delta(p, \blacktriangleright\blacktriangleright) = (2, p) \wedge \big[(1, \bar{p}) \vee ((2, p_\downarrow) \wedge (1, p))\big] \quad (6.5)$$

$$\delta(\bar{p}, \blacktriangleright\blacktriangleright) = (1, \bar{p}) \wedge (2, p) \quad (6.6)$$

$$\delta(p, q\,\blacktriangleright) = \bigvee_{q \xrightarrow{}_\Delta c, u} (2, p_u) \wedge \big[(1, \bar{p}_c) \vee ((2, p_\downarrow) \wedge (1, p_c))\big] \quad (6.7)$$

$$\delta(\bar{p}, q\,\blacktriangleright) = \begin{cases} \displaystyle\bigvee_{q \xrightarrow{}_\Delta c, u} (2, p_u) \wedge (1, \bar{p}_c) \text{ if } q \in Q_{\text{nd}}, \\ false \text{ otherwise} \end{cases} \quad (6.8)$$

and conclude with the rules for the p_t and $\bar{p}_t$:

$$\delta(p_t, fNil) = \delta(\bar{p}_t, fNil) = \begin{cases} \mathit{true} \text{ if } t = \mathit{Nil}, \\ \mathit{false} \text{ otherwise} \end{cases} \tag{6.9}$$

$$\delta(p_t, fq) = \delta(\bar{p}_t, fq) = \begin{cases} \mathit{true} \text{ if } t = q, \\ \mathit{false} \text{ otherwise} \end{cases} \tag{6.10}$$

$$\delta(p_{t_1 \blacktriangleright t_2}, f \blacktriangleright) = (2, p_{t_2}) \wedge \left[(1, \bar{p}_{t_1}) \vee ((2, p_\downarrow) \wedge (1, p_{t_1}))\right] \tag{6.11}$$

$$\delta(\bar{p}_{t_1 \blacktriangleright t_2}, f \blacktriangleright) = (1, \bar{p}_{t_1}) \wedge (2, p_{t_2}) \tag{6.12}$$

$$\delta(p_q, f \blacktriangleright) = \delta(p, q \blacktriangleright) \tag{6.13}$$

$$\delta(\bar{p}_q, f \blacktriangleright) = \delta(\bar{p}, q \blacktriangleright) \tag{6.14}$$

LEMMA 11. *$\mathcal{A}_{\xrightarrow{*}}$ satisfies the following correctness properties:*

$$L(p) = \{u \times v \mid u \xrightarrow{*} v\}, \qquad L(\bar{p}) = \{u \times v \mid u \xrightarrow{\mathrm{nd},*} v\}, \tag{6.15}$$

$$L(p_t) = \{u \times v \mid t \xrightarrow{*} v\}, \qquad L(\bar{p}_t) = \{u \times v \mid t \xrightarrow{\mathrm{nd},*} v\}, \tag{6.16}$$

$$L(p_\downarrow) = \{u \times v \mid v \not\rightarrow\}, \tag{6.17}$$

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

PROOF. (Sketch) For (6.17), we already noted that the table for $p_\downarrow$ is the obvious modification of (6.1) that ensures (6.17).

The properties (6.15) and (6.16) are proved simultaneously, by induction over the derivations $u \xrightarrow{*} v$, ..., for the ($\supseteq$) directions, and by induction over the product term for the ($\subseteq$) directions.

The case analysis is made simpler by the fact that every table entry between (6.3) and (6.8) is justified by a behavioral property we proved in Section 3.3. More precisely, Lemma 3 accounts for (6.4), Lemma 7 for (6.5), Lemma 5 for (6.6), Lemma 8 for (6.7), and Lemma 9 for (6.8). Finally, table entries between (6.9) and (6.14) are the obvious adaptations of entries between (6.3) and (6.8) where the u in $u \times v$ is discarded and replaced by a subterm t that the control state p_t or $\bar{p}_t$ carries along. □

We obtain the important corollary:

THEOREM 2. *The relations $\xrightarrow{*}$ and $\xrightarrow{\mathrm{nd},*}$ are recognizable. Furthermore, a tree automaton recognizing them only needs $O(|\Delta|)$ states.*

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

7. Reachability analysis for RPA

Theorem 2 immediately leads to decidability results for a wide range of problems related to reachability between RPA terms. We give a few examples. The nice thing with these results is that they all involve the same smooth and general automata-theoretic reasoning.

Similar results exist for other classes of systems where the reachability relation is recognizable (e.g., BPP Esparza [1997], timed automata Comon and Jurski [1999], pushdown processes Caucal [2000], and PA processes Lugiez and Schnoebelen [2000]) and our sole purpose in this section is to recall how the same general approach applies to all these decidability results.

7.1 Reachability sets

For any recognizable $L \subseteq T_Q$, the sets of *iterated successors*, $Post^*(L) \stackrel{\text{def}}{=} \{u \mid v \xrightarrow{*} u \text{ for some } v \in L\}$, and the set of *iterated predecessors*, $Pre^*(L) \stackrel{\text{def}}{=} \{u \mid u \xrightarrow{*} v \text{ for some } v \in L\}$, are recognizable.

Furthermore, the corresponding automata can be obtained in polynomial-time by standard intersection and projection constructs: assuming an automaton $\mathcal{A}$ for L is known, one constructs an automaton $\mathcal{A}'$ for $L \times T_Q$ by cylindrification, an automaton $\mathcal{A}''$ for $\{u \times v \mid u \in L \text{ and } u \xrightarrow{*} v\}$ by intersecting $\mathcal{A}'$ with $\mathcal{A}_{\xrightarrow{*}}$, and finally an automaton for $Post^*(L)$ by projecting $\mathcal{A}''$ along the second dimension.

7.2 Model checking the EF modal logic

With Pre^*, one can design a simple model checking algorithm for the EF logic (see Lugiez and Schnoebelen [2002] and Mayr [2001]).

Formally, EF is the modal logic given by the following syntax:

$$\varphi ::= \mathsf{EF}\,\varphi \mid \neg\varphi \mid \varphi \wedge \varphi' \mid P_1 \mid P_2 \mid \cdots$$

where $P_1, P_2, \ldots$ are atomic propositions.

The semantics of EF formulae is given by an interpretation $\|P_i\| \subseteq T_Q$ for the atomic propositions, and the inductive definition:

$$\begin{array}{lll} u \models \mathsf{EF}\,\varphi & \text{iff} & u \xrightarrow{*} v \text{ for some } v \text{ s.t. } v \models \varphi, \\ u \models P_i & \text{iff} & u \in \|P_i\|, \end{array}$$

where the standard clauses for negation and conjunction are omitted.

If we now associate, to any EF formula φ, its set of models $Mod(\varphi) \stackrel{\text{def}}{=} \{u \in T_Q \mid u \models \varphi\}$, the semantics of EF entails that $Mod(\varphi)$ can be computed inductively over the structure of φ:

$$\begin{array}{ll} Mod(\mathsf{EF}\,\varphi) = Pre^*(Mod(\varphi)), & Mod(P_i) = \|P_i\|, \\ Mod(\varphi \wedge \varphi') = Mod(\varphi) \cap Mod(\varphi'), & Mod(\neg\varphi) = T_Q \setminus Mod(\varphi). \end{array}$$

Assume that every $\|P_i\|$ is a recognizable subset of T_Q. Then, for every EF formula φ, $Mod(\varphi)$ is a recognizable subset of T_Q. Furthermore, from automata for the $\|P_i\|$, one easily constructs an automaton for $Mod(\varphi)$. This only uses standard boolean constructions, projections and cylindrifications. (This simple-minded approach has the usual inconvenient that every negation can cost an exponential blowup on the size of the resulting automata).

This approach is more general than standard model checking: rather than deciding whether a given u satisfies a given φ, the method computes the set $Mod(\varphi)$ of all "solutions" to φ (the set is given under the form of a recognizing tree automaton).

7.3 Model checking the reachability logic

Model checking EF only uses effective recognizability of $Pre^*(L)$ for recognizable L. With recognizability of $\xrightarrow{*}$ (and $\rightarrow$), one can go beyond EF and design a simple model checking algorithm for the full *reachability logic*, i.e. the first-order logic $FO(\rightarrow, \xrightarrow{*})$.

More formally, let $\varphi(x_1, \ldots, x_n)$ be a first-order formula where the free variables are among $x_1, \ldots, x_n$ and where all predicates are interpreted by recognizable relations over some $T_{\mathcal{F}}$ (we consider first-order logic without function symbols). Then, the set $Mod(\varphi)$ of the solutions of φ, i.e. the set of products $t_1 \times \cdots \times t_n$ s.t. $t_1, \ldots, t_n \models \varphi(x_1, \ldots, x_n)$, is a recognizable set. Furthermore, from automata recognizing the interpretations of the predicates in φ, an automaton for $Mod(\varphi)$ is easily constructed (see Thatcher and Wright [1968] and Comon *et al.* [1997–99] for more details).

For RPA terms, the reachability logic falls in this framework since the binary relations $\rightarrow$ and $\xrightarrow{*}$ are recognizable. Therefore, it is possible to compute the solutions of, for example,

$$\varphi(x) \equiv \forall y, y'(x \xrightarrow{*} y \wedge x \xrightarrow{*} y') \Rightarrow \exists z(y \xrightarrow{*} z \wedge y' \xrightarrow{*} z), \qquad \text{(Confl)}$$

that is, the set of all terms satisfying a confluence property.

Additional predicates can be used as long as they are interpreted by recognizable relations (natural candidates are equality, the atomic propositions $P_1, P_2, \ldots$ we considered for EF in section 7.2, etc.).

While the reachability logic is quite expressive (e.g., all EF formulae are simple reachability formulae), it does not compare easily with temporal logics like CTL or LTL. There exist temporal formulae, e.g. $\mathrm{EG}\, P_0$, that cannot be stated in the reachability logic, and conversely, there exist reachability formulae, like Eq. (Confl) above, that cannot be stated in any of the usual temporal logics – confluence can distinguish processes that are otherwise bisimilar and thus indistinguishable by temporal logics that only look at the computation tree. Also, the set of processes that satisfy Eq. (Confl) cannot be expressed in terms of Pre^*, $Post^*$ and boolean operations –. We refer to Comon and Jurski [1999] and Lugiez and Schnoebelen [2000] for examples of what can be achieved when reachability logic is decidable.

REMARK 1. For the RPA process algebra, it is difficult to extend decidability beyond the reachability logic: 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.

8. Reachability analysis for RPPS

We introduced RPA as a tool for the study of RPPS systems. In this section we show how reachability problems for RPPS systems can be solved with the tree-automata techniques we developed for RPA.

In the rest of this section we assume given a scheme $G = \langle Q, q_0, \Lambda \rangle$ and its encoding in RPA rules Δ_G, as defined in Section 4.

8.1 Model checking EF formulae

EF formula can be interpreted over any transition system, and in particular over $\mathcal{M}_G$. Of course, this assumes we are given interpretations $\|P_i\|_{\text{RPPS}} \subseteq M(G)$ for the atomic propositions $P_1, P_2, \ldots$ These interpretations give rise to interpretations $\|P_i\|_{\text{RPA}} \subseteq T_Q$ for the RPA terms, via

$$\|P_i\|_{\text{RPA}} \stackrel{\text{def}}{=} \{t \mid \mathcal{S}(t) \in \|P_i\|_{\text{RPPS}}\} \tag{8.1}$$

Now we can state that the RPA encoding of RPPS schemes preserves EF properties:

LEMMA 12. *For any EF formula φ and any $t \in T_Q$, $t \models \varphi$ iff $\mathcal{S}(t) \models \varphi$.*

PROOF. By induction over the structure of φ. The case where φ is an atomic proposition is given by (8.1). The case where φ is some $\mathsf{EF}\,\psi$ is given by Propositions 1 and 2, and the boolean cases are obvious. □

COROLLARY 1. *For any EF formula φ, $Mod_{RPPS}(\varphi) = \mathcal{S}(Mod_{RPA}(\varphi))$.*

Therefore, model checking EF formulae on RPPS schemes reduces to model checking on RPA systems, which can be solved with the tree-automata methods from Section 7.2 *on the condition that the $\|P_i\|$'s are recognizable RPA sets*. This condition is not a real limitation since most atomic propositions we would like to use in practice give rise to recognizable sets of RPA terms (for example, "$s \models P$ iff q_0 occurs in s", or "$s \models P'$ iff s has nesting depth higher than 2").

Under this assumption it is possible, for any EF formula φ, to compute $Mod_{\text{RPPS}}(\varphi)$ under the form of a recognizable subset of T_Q, from which one can easily see whether a given s satisfies φ.

8.2 Reachability between RPPS markings

The reachability problem for RPPS does not reduce directly to reachability questions on RPA terms. Here the problem we want to solve is "given two markings s and s', does $s \xrightarrow{*} s'$?". Decidability is claimed in Kouchnarenko and Schnoebelen [1997] but the proof (using a tableau method not unlike Mayr's solution for PA Mayr [1997]) was never published. In this section we show how tree automata can help design a simple algorithm running in non-deterministic polynomial time.

For two RPA terms u and v, we write $u \Rightarrow v$ when $u \equiv_{\mathcal{S}} u' \rightarrow v' \equiv_{\mathcal{S}} v$ for some u', v'. For $n \in \mathcal{N}$, we write $u \overset{n}{\Rightarrow} v$ when $u \equiv_{\mathcal{S}} u_0 \rightarrow v_0 \equiv_{\mathcal{S}} u_1 \rightarrow \ldots v_n \equiv_{\mathcal{S}} v$. By convention, $\overset{0}{\Rightarrow} = \equiv_{\mathcal{S}}$. We write $u \overset{*}{\Rightarrow} v$ when $u \overset{n}{\Rightarrow} v$ for some $n \in N$.

Reachability between RPPS markings reduces to $\overset{*}{\Rightarrow}$-reachability between RPA terms, in the following formal sense:

PROPOSITION 4. *For any two RPPS markings s and s', $s \xrightarrow{*} s'$ in $\mathcal{M}_G$ iff $\mathcal{T}(s) \overset{*}{\Rightarrow} \mathcal{T}(t)$.*

PROOF. Combine Theorem 1 and the definition of $\overset{*}{\Rightarrow}$. □

The next section shows that $\overset{*}{\Rightarrow}$-reachability is decidable in non-deterministic polynomial time – this result also holds for RPA systems that are not the encoding of some RPPS scheme.

THEOREM 3. *Reachability between RPPS markings is NP-complete.*

PROOF. Membership in NP is shown by combining Proposition 4 and Theorem 5 below. NP-hardness is well-known already for simpler process algebra like BPP and the reduction from 3SAT given in Esparza [1997, Theorem 3.2] is easily adapted to the RPPS framework. □

9. $\overset{*}{\Rightarrow}$-reachability for RPA

In this section we develop an algorithm for $\overset{*}{\Rightarrow}$-reachability between RPA terms. We consider the general case of an RPA presentation (some $Q = Q_{\mathrm{nd}} \cup Q_{\mathrm{d}}$ with some Δ) without assuming it encodes some RPPS scheme.

9.1 A combinatorial characterization of $\equiv_{\mathcal{S}}$

We start with a characterization of $\equiv_{\mathcal{S}}$ that is more manageable from a tree-automata viewpoint. This is done in several small steps, with the help of some simplification or permutation relations between RPA terms. Our approach is inspired by the treatment of structural congruence in recognizable sets of PA terms Lugiez and Schnoebelen [2002] (but here $\equiv_{\mathcal{S}}$ is not a

congruence). The basic concepts (confluence, commutations, ...) used in this subsection are standard in the study of reduction systems (see e.g. Klop [1992]).

9.1.1 Simplification

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

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

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

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

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 $(\twoheadrightarrow^*)^{-1}$. Note that $u \twoheadrightarrow t$ implies $|u| > |t|$, so that $\twoheadrightarrow$ is Noetherian.

LEMMA 13. (CONFLUENCE OF SIMPLIFICATIONS) *If $u \twoheadleftarrow v \twoheadrightarrow w$, then $u = w$ or $u \twoheadrightarrow v' \twoheadleftarrow w$ for some v'.*

PROOF. By structural induction on v and case analysis. See Appendix A.4. □

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.

9.1.2 Permutation

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

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

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

Obviously, $\rightleftarrows$ is symmetric. We write $\overset{*}{\rightleftarrows}$ to denote its reflexive-transitive closure. When $u \overset{*}{\rightleftarrows} v$, we say u and v are *permutationally equivalent*. Note that $u \overset{*}{\rightleftarrows} v$ implies $|u| = |v|$.

Simplifications and permutations commute:

LEMMA 14. (COMMUTATION) *1. If* $u \rightleftarrows v \searrow w$, *then* $u \searrow v' \overset{*}{\rightleftarrows} w$ *for some* v'.
2. Symmetrically, if $u \swarrow v \rightleftarrows w$, *then* $u \overset{*}{\rightleftarrows} v' \swarrow w$ *for some* v'.

PROOF. We prove 1. by structural induction on u and case analysis. See Appendix Appendix A.5. Then we get 2. by symmetry. □

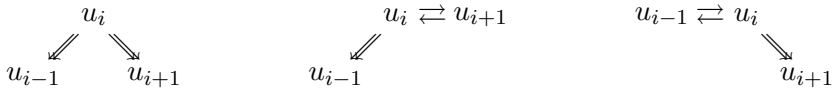
9.1.3 Convertibility

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

LEMMA 15. *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 $(c) \Rightarrow (b) \Rightarrow (a)$.

One proves $(a) \Rightarrow (b)$ by a standard "peaks into valleys" normalization: assume $u \leftrightsquigarrow v$ comes from a conversion sequence $u = u_0 R_1 u_1 R_2 \cdots R_n u_n = v$ with $R_i \in \{\searrow, \swarrow, \rightleftarrows\}$. In such a sequence, we say there is a *peak* at position i if R_{i-1} and R_i are as in one of the following three patterns:



Using Lemmas 13 and 14, it is possible to erase any peak in a conversion sequence. This may lengthen the sequence and may introduce new peaks. However, erasing peaks must eventually terminate: every time a peak is erased, some term u_i is removed from the sequence and we only introduce new terms $v_1, \ldots$ that are strictly smaller than u_i since $u_i \overset{*}{\rightleftarrows} \searrow \overset{*}{\rightleftarrows} v_j$ entails $|u_i| > |v_j|$. When eventually u is converted to v via a sequence with no peak, then $u \searrow^* \overset{*}{\rightleftarrows} {}^*\!\swarrow v$.

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

PROPOSITION 5. $u \equiv_{\mathcal{S}} v$ *if and only if* $u \leftrightsquigarrow v$.

PROOF. The $(\Leftarrow)$ direction is obvious: a simple inspection of the defining rules shows 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 Appendix A.6. □

9.2 Recognizable sets and $\equiv_{\mathcal{S}}$

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 recognizable set L modulo $\mathcal{S}$-equivalence?".

For a tree language $L \subseteq T_Q$ define

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

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

THEOREM 4. *If L is recognizable then $[L]_{\swarrow}$ and $[L]_{\swarrow\!\!\swarrow}$ are recognizable. Furthermore, from a tree automaton $\mathcal{A}$ recognizing L, one can build in polynomial-time tree automata $\mathcal{A}'$ and $\mathcal{A}''$ for $[L]_{\swarrow}$ and $[L]_{\swarrow\!\!\swarrow}$ with $|\mathcal{A}'|$ and $|\mathcal{A}''|$ in $O(|\mathcal{A}|^2)$.*

PROOF. (Sketch) $\mathcal{A}'$ is built on top of $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \delta \rangle$ by adding new control states and new transition rules. The new states are r_{Nil} and all r_p^q for p, q two states of $\mathcal{A}$. The new rules are all rules of the following form, where we quantify over all $p, p', q, q' \in \mathcal{Q}$:

$$r_p^q \mapsto r_{Nil} \blacktriangleright r_p^q, \qquad p \mapsto p \blacktriangleright r_{Nil}, \qquad r_{Nil} \mapsto Nil,$$
$$r_p^q \mapsto r_p^q \blacktriangleright r_{Nil}, \qquad q \mapsto p \blacktriangleright r_p^q, \qquad r_{Nil} \mapsto r_{Nil} \blacktriangleright r_{Nil},$$
$$r_p^{q'} \mapsto r_p^q \blacktriangleright r_q^{q'},$$
$$r_p^{q'} \mapsto r_{Nil} \blacktriangleright q \text{ (if } q' \mapsto p \blacktriangleright q \text{ is in } \delta),$$
$$r_p^{q'} \mapsto r_p^q \blacktriangleright p' \text{ (if } q' \mapsto q \blacktriangleright p' \text{ is in } \delta).$$

This gives an automaton $\mathcal{A}'$ such that, for any $t \in T_Q$ and $p \in \mathcal{Q}$:

$$p \stackrel{*}{\mapsto} t \text{ in } \mathcal{A}' \text{ iff } t \searrow^* u \text{ for some } u \text{ such that } p \stackrel{*}{\mapsto} u \text{ in } \mathcal{A}. \tag{9.1}$$

The proof of (9.1) goes along standard lines, and is given in full detail in Labroue [2002]. It follows that $L_{\mathcal{A}'}(p)$ is $[L_{\mathcal{A}}(p)]_{\swarrow}$.

Once we have $\mathcal{A}'$, it is easy to build $\mathcal{A}''$ on top of this, relying on $\searrow\!\!\searrow^* = \searrow^* \curvearrowright^{=}$. □

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

The following lemma is a direct corollary of Theorem 1 when Δ is the encoding Δ_G of some RPPS scheme. However, it is true of any RPA system.

LEMMA 16. ($\equiv_{\mathcal{S}}$ IS A BISIMULATION) *$u \equiv_{\mathcal{S}} v$ and $u \to u'$ implies that $v \to v'$ for some v' with $u' \equiv_{\mathcal{S}} v'$.*

PROOF. (Idea) Standard but tedious. One proves that $\rightleftarrows$, $\searrow$ and $\curvearrowright$ are bisimulations up-to $\leftrightsquigarrow$. Proposition 5 concludes. □

LEMMA 17. *$u \overset{*}{\Rightarrow} v$ iff $u \overset{*}{\to} w$ for some w s.t. $v{\downarrow} \overset{*}{\rightleftarrows} w{\downarrow}$.*

PROOF. With Proposition 5 and Lemma 15, it is enough to prove that for any $n \in \mathcal{N}$, $t \overset{n}{\Rightarrow} u$ iff $t \overset{n}{\to} u'$ for some $u' \equiv_{\mathcal{S}} u$. Here the ($\Leftarrow$) direction is by definition of $\overset{n}{\Rightarrow}$, and the ($\Rightarrow$) direction is given by Lemma 16 and an induction on n. □

THEOREM 5. *$\overset{*}{\Rightarrow}$-reachability between RPA terms is NP-complete.*

PROOF. We only show membership in NP since NP-hardness is clear when already RPPS reachability is NP-hard. A possible non-deterministic algorithm deciding whether $u \overset{*}{\Rightarrow} v$ is:

(1) compute $v{\downarrow}$ in polynomial-time;

(2) guess a w s.t. $v{\downarrow} \overset{*}{\rightleftarrows} w$ (this essentially amounts to picking a permutation of some subtems of $v{\downarrow}$);

(3) build a tree automaton for $L = [w]_{\swarrow\!\!\swarrow}$ using Theorem 4, and then an automaton for $L' = \mathit{Pre}^*(L) = \{t \mid t \overset{*}{\to} t' \in L\}$ using Theorem 2 (these automata can be built in polynomial-time);

(4) answer yes if $u \in L'$.

Lemma 17 states that this algorithm is correct. □

10. 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 first-order logic of reachability,
- which can be used for the reachability analysis of RPPS systems.

The main difficulty in that work was to discover a process-algebraic presentation of RPPS 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 kinds of steps "$u \xrightarrow{\text{nd}} v$" and "$u \xrightarrow{\text{d}} v$", 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.

Acknowledgements

We thank the anonymous referees whose questions and suggestions helped us improve our earlier version.

Appendix A.

Appendix A.1 Comparing PA, RPPS and PRS

RPPS systems can be used to encode PA processes in a faithful way. In the reverse direction, encoding RPPS schemes into PA processes only preserves linear-time behavior Kouchnarenko and Schnoebelen [1997], and it does not seem possible to do better. Here we illustrate wherein lies the difficulty.

Let us consider the step

$$(q_1, \emptyset) \rightarrow (q_2, (q_8, \emptyset)) \tag{1.1}$$

in the scheme G_{exmp} from Fig. 2.1. Here control moves from q_1 to q_2 while spawning a parallel subprocess in q_8. If the process in q_2 is to follow the branch that leads to q_4, then the two processes will evolve independently and (1.1) can be encoded as some PA rule of the form

$$X_{q_1} \rightarrow X_{q_2 - q_4} \parallel X_{q_8}$$

If q_2 is to follow the branch that leads to q_7, then the wait node at q_6 will be crossed after the subprocess at q_8 is terminated. In this case (1.1) would be encoded as some PA rule of the form

$$X_{q_1} \rightarrow (X_{q_2 - q_6} \parallel X_{q_8}).X_{q_7}$$

If we define the behavior of X_{q_1} by collecting these two rules, we end up with a process that has linear runs mimicking those of $(q_1, \emptyset)$ in G_{exmp} but its branching behavior is different. The PA process X_{q_1} has to decide on its first step whether X_{q_8} and X_{q_2} will be combined in a parallel or in a

sequential composition, while the process $(q_1, \emptyset)$ in G_{exmp} will only make this kind of choice at a later time.

The branching-time behavior of G_{exmp} and other RPPS systems can be encoded in Mayr's PRS framework, but this formalism does not allow the simple automata-theoretic approach we use for reachability analysis of PA and RPPS.

Appendix A.2 Proof of Proposition 1

Assume $u \to v$ is an RPA step. We prove that $\mathcal{S}(u) \to \mathcal{S}(v)$ is a step in $\mathcal{M}_G$ by induction on the derivation of $u \to v$:

- $u \to v$ given by R1: Then u is some $q \in Q_{\text{nd}}$, v is some $c \blacktriangleright t$, and $q \to c, t$ is in Δ_G. We consider the several possibilities for the rule $q \to c, t$:
  - the rule is given by (Da): then q is a standard or a wait node, there is an edge $(q, q') \in E$ and $v = q'$. Thus $(q, \emptyset) \to (q', \emptyset)$ by (Ga), which concludes since $\mathcal{S}(u) = (q, \emptyset)$ and $\mathcal{S}(v) = (q', \emptyset)$.
  - the rule is given by (Dc): then $\Lambda(q) = \texttt{call}\ q''$, there is an edge $(q, q') \in E$ and $v = q' \blacktriangleright q''$. Thus $(q, \emptyset) \to (q', \{(q'', \emptyset)\})$ by (Gc), which concludes since $\mathcal{S}(v) = (q', \mathcal{S}(q'')) = (q', \{(q'', \emptyset)\})$ by (S1-2).
  - the rule is given by (De): then q is an end node and $v = \mathit{Nil} \blacktriangleright \mathit{Nil}$. Thus $(q, \emptyset) \to \emptyset$ by (Ge), which concludes since $\mathcal{S}(v) = \emptyset$.
- $u \to v$ given by R3: then u is some $u' \blacktriangleright u''$, v is some $u' \blacktriangleright v''$, and $u'' \xrightarrow{\text{nd}} v''$ is an RPA step so that, by ind. hyp., $\mathcal{S}(u'') \to \mathcal{S}(v'')$ is a step in $\mathcal{M}_G$. u and v can be written respectively as $c \blacktriangleright^n (u_1, \ldots, u_{n-1}, u'')$ and $c \blacktriangleright^n (u_1, \ldots, u_{n-1}, v'')$ so that $\mathcal{S}(u) \to \mathcal{S}(v)$ is obtained from $\mathcal{S}(u'') \to \mathcal{S}(v'')$ by repeated application of (Gp1), and one application of (Gp2) if $c \neq \mathit{Nil}$.
- $u \to v$ given by R6: then u is some $u' \blacktriangleright u''$, v is some $v' \blacktriangleright u''$, $u' \xrightarrow{\text{d}} v'$ is an RPA step allowed by $\mathit{State}(u'') = \emptyset$, and, by ind. hyp., $\mathcal{S}(u') \to \mathcal{S}(v')$ is a step in $\mathcal{M}_G$. From the condition on u'' we deduce $\mathcal{S}(u'') = \emptyset$, so that $\mathcal{S}(u') = \mathcal{S}(u)$ and $\mathcal{S}(v') = \mathcal{S}(v)$, concluding the proof.
- $u \to v$ given by R2: is dealt mostly like the R1 case.
- $u \to v$ given by R4 or R5: is dealt mostly like the R3 case.

Appendix A.3 Proof of Proposition 2

Assume $\mathcal{S}(u) = s$ and $s \to s'$ is a step in $\mathcal{M}_G$. We prove that $u \to v$ for some v s.t. $\mathcal{S}(v) = s'$ by induction on the structure of u. We start with the case where u is some $\mathit{Nil} \blacktriangleright^n (u_1, \ldots, u_n)$. Then, writing s_i for $\mathcal{S}(u_i)$, s is $s_1 + \cdots + s_n$, and the step $s \to s'$ implies that there exists a step $s_i \to s_i'$ for some $1 \leq i \leq n$. Then s' is $s_1 + \cdots + s_i' + \cdots + s_n$ and the induction

hypothesis entails that $u_i \to v'$ for some v' with $\mathcal{S}(v') = s'_i$. We deduce $Nil \blacktriangleright^i (u_1, \ldots, u_i) \overset{\text{nd}}{\to} Nil \blacktriangleright^i (u_1, \ldots, v')$ using (R3) or (R4) and, writing v for $Nil \blacktriangleright^n (u_1, \ldots, v', \ldots, u_n)$, we further obtain $u \to v$ using (R5) $n-i-1$ times. Observing that $\mathcal{S}(v) = s'$ concludes the proof.

If now u is some $q \blacktriangleright^n (u_1, \ldots, u_n)$ then s is $(q, s_1 + \cdots + s_n)$ (again writing s_i for $\mathcal{S}(u_i)$). If $s \to s'$ comes from some $s_i \to s'_i$ then the proof proceeds very much like the previous case. The other possibility is that $s \to s''$ really comes from q, i.e. is induced by one the rules (Ga), (Ge), (Gc) or (Gw). We consider all cases:

- $s \to s'$ induced by (Ga): Then q is a standard node, and there is an edge $(q, q') \in E$ s.t. s' is $(q', s_1 + \cdots + s_n)$. We have $q \to q', Nil$ in Δ_G by (Da), so that $u \overset{\text{nd}}{\to} v = q' \blacktriangleright^{n+1} (Nil, u_1, \ldots, u_n)$ can be proved with (R1) and (R5). We conclude with $\mathcal{S}(v) = s'$.
- $s \to s'$ induced by (Ge): Then q is an end node and $s' = s_1 + \cdots + s_n$. We have $q \to Nil, Nil$ in Δ_G by (De), so that $u \overset{\text{nd}}{\to} v$ for $v = Nil \blacktriangleright^{n+1} (Nil, u_1, \ldots, u_n)$. We conclude with $\mathcal{S}(v) = s'$.
- $s \to s'$ induced by (Gc): Then $\Lambda(q)$ is some `call` q'', and there is an edge $(q, q') \in E$ s.t. s' is $(q', s_1 + \cdots + s_n + (q'', \emptyset))$. We have a $q \to q', q''$ in Δ_G by (Dc), so that $u \overset{\text{nd}}{\to} v$ for $v = q' \blacktriangleright^{n+1} (q'', u_1, \ldots, u_n)$. We conclude with $\mathcal{S}(v) = s'$.
- $s \to s'$ induced by (Gw): Then q is a wait node, $s = (q, \emptyset)$, and there is an edge $(q, q') \in E$ s.t. s' is $(q', \emptyset)$. Necessarily, $s_i = \emptyset$ for $i = 1, \ldots, n$, entailing $State(u_i) = \emptyset$. There is a $q \to q', Nil$ in Δ_G by (Dw), so that $u \overset{\text{d}}{\to} v$ for $v = q' \blacktriangleright^{n+1} (Nil, u_1, \ldots, u_n)$. We conclude with $\mathcal{S}(v) = s'$.

Appendix A.4 Proof of Lemma 13

We prove the lemma by induction on the structure of 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 $v = Nil \blacktriangleright u$ and, since $u \neq w$, $w = Nil \blacktriangleright u'$ 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 $u \searrow \swarrow w$ using (E2). 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 $u \searrow \swarrow w$ using (E3).

Appendix A.5 Proof of Lemma 14

We assume $u \rightleftarrows v \searrow w$ and prove the Lemma by induction on the structure of w. Write w under the form $c \blacktriangleright^n (w_1, \ldots, w_n)$. If $n = 0$ then $v = \mathit{Nil} \blacktriangleright c$ 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 $v = \mathit{Nil} \blacktriangleright w$ and $u = \mathit{Nil} \blacktriangleright w'$ with $w' \rightleftarrows w$. We are done since $u \searrow w'$.
- If $v \searrow w$ by rule (E2), then v is some $c \blacktriangleright^{n-m+1} (w_1, \ldots, w_{i-1}, \mathit{Nil} \blacktriangleright^m (w_i, \ldots, w_{i+m-1}), w_{i+m}, \ldots, w_n)$ 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 $\mathit{Nil} \blacktriangleright^m (w_i, \ldots, w_{i+m-1})$ subterm of v, then it is easy to see that $u \searrow \rightleftarrows w$.
  Otherwise the $\mathit{Nil} \blacktriangleright^m (w_i, \ldots, w_{i+m-1})$ subterm of v is swapped with w_{i-1} or w_{i_m}. In the first case u is $c \blacktriangleright^{n-m+1} (w_1, \ldots, w_{i-2}, \mathit{Nil} \blacktriangleright^m (w_i, \ldots, w_{i+m-1}), w_{i-1}, w_m, \ldots, w_n)$ 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 $v' = c \blacktriangleright^n (w_1, \ldots, w_{i-1}, v'', w_{i+1}, \ldots, w_n)$.

Appendix A.6 Proof of Proposition 5

There only remains to prove the ($\Rightarrow$) direction of Proposition 5. We start with the following lemma:

LEMMA 18. *$u \leftrightsquigarrow u'$ implies $c \blacktriangleright^n (\ldots, u, \ldots) \leftrightsquigarrow c \blacktriangleright^n (\ldots, u', \ldots)$.*

PROOF. By induction on the length of the conversion sequence $u \leftrightsquigarrow u'$. For the base case, one assumes $u \searrow u'$ (resp. $u \curvearrowright u'$, $u \rightleftarrows u'$) and 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 $c' = \mathit{Nil}$, 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 $v_j \leftrightsquigarrow \mathit{Nil}$ for $j \neq k$. Thus $v \leftrightsquigarrow \mathit{Nil} \blacktriangleright^m (\mathit{Nil}, \ldots, \mathit{Nil}, u, \mathit{Nil}, \ldots \mathit{Nil})$ by Lemma 18. Then $v \leftrightsquigarrow \mathit{Nil} \blacktriangleright u$ by (E2) and $v \leftrightsquigarrow u$ by (E1). The case where $c = \mathit{Nil}$ and $c' \in Q$ is symmetric.

◦ If $c = Nil = c'$, 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 18, then $u \leftrightsquigarrow v$ by (P1).

Otherwise some u_i or v_j has the form $Nil \blacktriangleright^k (w_1, \ldots, w_k)$, 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

Aceto, L., Fokkink, W. J., and Verhoef, C. 2001. Structural operational semantics. In *Handbook of Process Algebra*, Bergstra, J. A., Ponse, A., and Smolka, S. A., Editors. Elsevier Science, chapter 3, 197–292.

Baeten, J. C. M. and Weijland, W. P. 1990. *Process algebra*. Volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge Univ. Press.

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

Bukart, O., Caucal, D., Moller, F., and Steffen, B. 2001. Verification on infinite structures. In *Handbook of Process Algebra*, Bergstra, J. A., Ponse, A., and Smolka, S. A., Editors. Elsevier Science, chapter 9, 545–623.

Caucal, D. 2000. 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*, Volume 1784 of *Lecture Notes in Computer Science*, 48–62.

Chandra, A. K., Kozen, D. C., and Stockmeyer, L. J. 1981. Alternation. *Journal of the ACM 28*, 1, 114–133.

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

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

Dauchet, M. and Tison, S. 1990. The Theory of Ground Rewrite Systems is Decidable. In *Proc. 5th IEEE Symp. Logic in Computer Science (LICS'90), Philadelphia, PA, USA, June 1990*. IEEE Comp. Soc. Press, 242–248.

Esparza, J. 1997. Petri Nets, Commutative Context-Free Grammars, and Basic Parallel Processes. *Fundamenta Informaticae 31*, 1, 13–25.

Esparza, J. and Knoop, J. 1999. 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*, Volume 1578 of *Lecture Notes in Computer Science*. Springer, 14–30.

Esparza, J. and Podelski, A. 2000. 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, 1–11.

Finkel, A. and Schnoebelen, Ph. 2001. Well Structured Transition Systems Everywhere! *Theoretical Computer Science 256*, 1–2, 63–92.

Klop, J. W. 1992. Term Rewriting Systems. In *Handbook of Logic in Computer Science, vol.2. Background: Computational Structures*, Abramsky, S., Gabbay, D. M., and Maibaum, T. S. E., Editors. Oxford Univ. Press, 1–116.

Kouchnarenko, O. and Schnoebelen, Ph. 1997. A Formal Framework for the Analysis of Recursive-Parallel Programs. In *Proc. 4th Int. Conf. Parallel Computing Technologies (PaCT'97), Yaroslavl, Russia, Sep. 1997*, Volume 1277 of *Lecture Notes in Computer Science*. Springer, 45–59.

Kouchnarenko, O. and Schnoebelen, Ph. 1997. A Model for Recursive-Parallel Programs. In *Proc. 1st Int. Workshop on Verification of Infinite State Systems (INFINITY'96), Pisa, Italy, Aug. 1996*, Volume 5 of *Electronic Notes in Theor. Comp. Sci.*. Elsevier Science.

Kupferman, O. and Vardi, M. Y. 2000. 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*, Volume 1855 of *Lecture Notes in Computer Science*. Springer, 36–52.

Labroue, A. 2002. Méthodes algébriques pour la vérification des systèmes infinis. Thèse ENS de Cachan.

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

Lugiez, D. and Schnoebelen, Ph. 2002. The Regular Viewpoint on PA-Processes. *Theoretical Computer Science 274*, 1–2, 89–115.

Mayr, R. 1997. Tableaux methods for PA-processes. In *Proc. Int. Conf. Automated Reasoning with Analytical Tableaux and Related Methods (TABLEAUX'97), Pont-à-Mousson, France, May 1997*, Volume 1227 of *Lecture Notes in Artificial Intelligence*. Springer, 276–290.

Mayr, R. 2000. Process Rewrite Systems. *Information and Computation 156*, 1/2, 264–286.

Mayr, R. 2001. Decidability of Model Checking with the Temporal Logic EF. *Theoretical Computer Science 256*, 1–2, 31–62.

Moller, F. 1996. Infinite Results. In *Proc. 7th Int. Conf. Concurrency Theory (CONCUR'96), Pisa, Italy, Aug. 1996*, Volume 1119 of *Lecture Notes in Computer Science*. Springer, 195–216.

Raoult, J.-C. 1997. Rational tree relations. *Bull. Belg. Math. Soc. 4*, 149–176.

Tanaka, J., Ueda, K., Miyazaki, T., Takeuchi, A., Matsumoto, Y., and Furukawa, K. 1986. Guarded Horn Clauses and Experiences with Parallel Logic Programming. In *Proc. 1986 Fall Joint Computer Conference, Nov. 1986, Dallas, TX, UA*. IEEE Comp. Soc. Press, 948–954.

Thatcher, J. W. and Wright, J. B. 1968. Generalized Finite Automata Theory with an Application to a Decision Problem of Second-Order Logic. *Mathematical Systems Theory 2*, 1, 57–81.

Vardi, M. 1997. Alternating Automata: Checking Truth and Validity for Temporal Logics. In *Proc. 14th Int. Conf. Automated Deduction (CADE'97), Townsville, North Queensland, Australia, July 1997*, Volume 1249 of *Lecture Notes in Computer Science*. Springer, 191–206.