Electronic Notes in Theoretical Computer Science 6 (1997)
URL: http://www.elsevier.nl/locate/entcs/volume6.html ?? pages

A Model for Recursive-Parallel Programs

O. Kouchnarenko, Ph. Schnoebelen

Leibniz-IMAG, 46 av. F. Viallet, F-38000 Grenoble, France

Abstract

We define a formal model for a class of recursive-parallel systems with specific invocation and synchronization primitives. This original model is infinite-state but can still be analyzed successfully using the "well-structured transition systems" approach.

Introduction

RP schemes are a formal model of concurrency that we introduced in [9]. It is a carefully chosen way of allowing *R*ecursivity in *P*arallel systems (hence the name). In this article, we show how RP schemes can be seen as well-structured transition systems. (Well-structured transition systems, or WSTS's, are a general family of non-necessarily finite transition systems where general decidability results exist [7,1].)

Apart from the original model and the specific decidability results we present, the interest of this work is that it shows how the WSTS approach can be adapted to new situations. We present two WSTS views that extend (section 6) or modify (section 5) the standard WSTS definition.

The article is organized as follows: Section 1 presents the programming language features we want to formalize. Sections 2 and 3 present RP schemes, the formal model, and its behavioral semantics. In Section 4 we compare the expressive powers of RP schemes and related models. Then, Section 5 explains how RP schemes can be viewed as well-structured transition systems, yielding the decidability of some reachability problems. Section 6 gives another well-structured view of RP schemes, allowing further decidability results (e.g. termination). We conclude by relating RP schemes and other formal models.

1 A recursive-parallel programming language

RP is a (family of) imperative parallel programming languages developed for the parallel machine of the IPI Institute in Iaroslavl. The language assumes a shared global memory.

Fig. 1 contains an example of an RP program where we used abstract action names $a, b, c, \ldots$ from some uninterpreted alphabet A instead of the usual basic actions from imperative languages (assignments, ...).

```
program main                  procedure subr1
     a1;                           if b2 then {
l1:  pcall subr1;                    a4;
     a2;                             end;
     if b1 then {                  } else {
       goto l1;                      pcall subr1;
     } else {                        a5;
       wait;                         wait;
       a3;                           end;
       end;                        }
     }
```

Fig. 1. An RP program

RP interests us because of its specific choice of primitives for parallelism and synchronization:

pcall: or "parallel call". This construct invokes co-routines, "*callees*", that will run in parallel while execution proceeds concurrently in the caller.

wait: This construct allows one form of synchronization. An invocation of a procedure can only conclude a `wait` statement when all its callees (and their callees, etc.) are terminated. As long as some of them are still running, the caller waits for their termination.

Termination of a given invocation is obtained by the `end` statement. The use of `wait` is not mandatory: a caller may well choose to terminate and let its callees run. Observe the asymmetry brought by the fact that callees cannot know whether their caller is terminated.

2 Recursive-parallel program schemes

RP programs schemes are better dealt with in a graphic form. A *recursive-parallel program scheme* (over A) is a finite rooted graph G representing the structure of an RP program. We let $RPPS_A$ denote the class of all such graphs. We will only give an informal description of these graphs and let the reader write down the precise formal definitions (alternatively, s/he may refer to [8,9]). Fig. 2 shows, by means of an example, how a scheme is associated, in an obvious way, to the RP program. A scheme has several kind of nodes: rectangular (resp. oval) nodes for basic actions (resp. tests), pentagonal nodes for `pcall`'s, triangular nodes for `wait` statements, etc. In this paper, we write $G = \langle Q, q_0, \ldots \rangle$ to denote that Q is the *set of nodes* of scheme G, q_0 is the *start node*, and we do not need names for the remaining components of G.

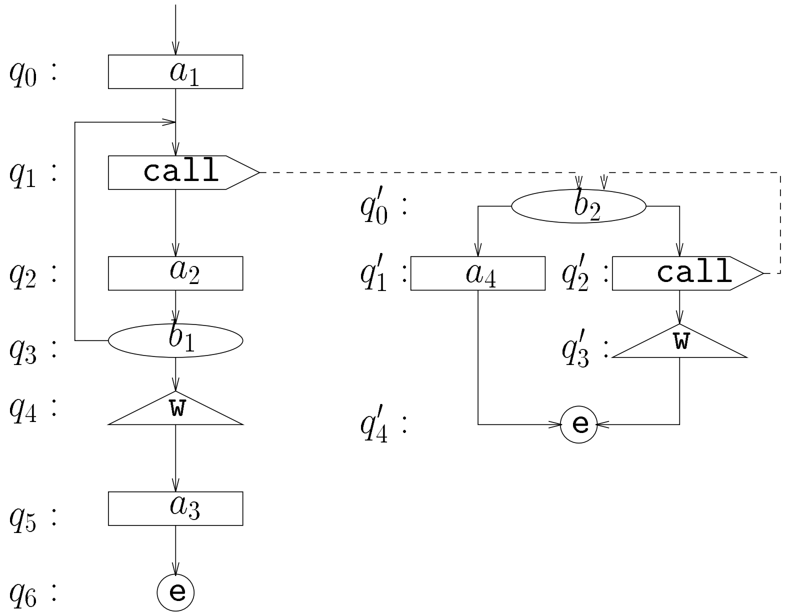

Fig. 2. Scheme associated to the RP program from Fig. 1

3 Behavioral semantics

We consider a given scheme $G = \langle Q, q_0, \ldots \rangle \in RPPS_A$ and define a formal notion of behavior under the form of a labeled transition system $TS(G)$. Labels are taken from A_τ ($\stackrel{\text{def}}{=} A \cup \{\tau\}$, where τ is a special name denoting silent, internal computation) and ranged over by $\alpha, \ldots$

G may be the scheme G_P associated to some RP program P but, more generally, it may be any valid scheme without any textual, linear presentation. Because we do not interpret the basic actions in G, our notion of behavior only partly captures the real behavior of a program P to which the scheme G could be associated. The difference is that

- $TS(G)$ does not consider the values of data components of P in its definition of a state, and
- $TS(G)$ considers "`if` ... `then` c_1 `else` c_2" instructions as non-deterministic "c_1 `or` c_2".

These abstractions do not preclude a meaningful analyzing of P by examination of $TS(G_P)$: any safety property (in an enlarged sense where termination is preserved) satisfied by $TS(G_P)$ is also satisfied by the interpreted semantics of P [9,8].

Definition 3.1 The set of *hierarchical states* of a scheme G is the least set $M(G)$ s.t. if $q_1, \ldots, q_n$ are nodes of G, and $\xi_1, \ldots, \xi_n \in M(G)$ are hierarchical states, then the multiset $\xi = \{(q_1, \xi_1), \ldots, (q_n, \xi_n)\}$ is in $M(G)$. (In particular, $\emptyset \in M(G)$.)

The intuition is that $\xi = \{(q_1, \xi_1), \ldots, (q_n, \xi_n)\}$ denotes a state where n concurrent activities are present. One such activity, say (q_i, ξ_i), is the invocation of an RPC co-routine (currently in state/node q_i) with a family ξ_i of children invocations. So that a hierarchical state ξ looks like a marking of some Petri net (with tokens in the $q, q', \ldots$ nodes) with an additional tree-like structure between tokens, keeping track of the parent-child relation created by the `pcall`'s, and used by the `wait`'s statements.

In the rest of the article, we often omit a few parenthesis when this does not introduce any confusion, and write e.g. q, ξ for $\{(q, \xi)\}$. Also, we use the customary multiset notations "$\xi + \xi'$", "$\xi \subseteq \xi'$", ... to denote addition, inclusion, ...

Definition 3.2 The transition system $TS(G) = \langle M(G), A_\tau, \rightarrow, \xi_0 \rangle$ has initial state $\xi_0 \stackrel{\text{def}}{=} q_0, \emptyset$, and labeled transition relation $\rightarrow \subseteq M(G) \times A_\tau \times M(G)$ defined as the least family of triples (ξ, a, ξ') (written "$\xi \xrightarrow{a} \xi'$") obeying the following rules:

action: *If q is an α-labeled action node (or test node) in G, and q' is a successor node of q, then $q, \xi \xrightarrow{\alpha} q', \xi$ for all ξ.*

end: *If q is an* `end` *node in G, then $q, \xi \xrightarrow{\tau} \xi$ for all ξ.*

call: *If q is a* `pcall` *node in G, with successor node q' and with invoked node q'', then $q, \xi \xrightarrow{\tau} q', (\xi\ +\ q'', \emptyset)$ for all ξ.*

wait: *If q is a* `wait` *node in G, and q' is a successor node of q, then $q, \emptyset \xrightarrow{\tau} q', \emptyset$.*

paral1: *If $\xi \xrightarrow{\alpha} \xi'$ then $\xi + \xi'' \xrightarrow{\alpha} \xi' + \xi''$ for all ξ''.*

paral2: *If $\xi \xrightarrow{\alpha} \xi'$ and q is a node of G, then $q, \xi \xrightarrow{\alpha} q, \xi'$.*

Rules for parallelism state that any activity $\xi \xrightarrow{\alpha} \xi'$ can still take place when brothers are present (i.e. in some $\xi + \xi''$) or when a parent is present (i.e. in some q, ξ). The wait-rule states how we can only perform a `wait` statement in state q if the invoked sons are all terminated (and then not present anymore). The other rules state how children invocations are created and kept around.

Our example scheme from Fig. 2 admits, among others, an execution starting with

$$q_0, \emptyset \xrightarrow{a_1} q_1, \emptyset \xrightarrow{\tau} q_2, (q'_0, \emptyset) \xrightarrow{b_2} q_2, (q'_2, \emptyset) \xrightarrow{a_3} q_3, (q'_2, \emptyset) \xrightarrow{b_1} q_4, (q'_2, \emptyset) \xrightarrow{\tau} q_4, (q'_3, (q'_0, \emptyset))$$

where the `wait` node q_4 cannot be passed yet, as long as the children invocations are not terminated.

Some consequences of Definition 3.2 can be stated immediately. First, $TS(G)$ is a finitely branching transition system. Then there exists only one terminated state: $\xi \not\rightarrow$ iff $\xi = \emptyset$.

More importantly, it is easy to see whether a state in $TS(G)$ is normed. Recall that a state ξ is *normed*, written $\xi \downarrow$, if there exists a terminating behaviour starting from ξ [6]. (NB: This is a "may terminate", quite different from the "must terminate" usually assumed in termination problems, and considered in Section 6.)

We can decide whether $\xi\downarrow$ in two steps. First we reduce normedness of arbitrary states in $M(G)$ to normedness of nodes in G through the following equivalences:

$$(\xi+\xi')\downarrow \text{ iff } \xi\downarrow \text{ and } \xi'\downarrow,$$
$$\emptyset\downarrow \text{ always},$$
$$(q,\xi)\downarrow \text{ iff } (q,\emptyset)\downarrow \text{ and } \xi\downarrow .$$

Then, writing $q\downarrow$ instead of the clumsier $(q,\emptyset)\downarrow$, we can state the following equalities based on the structure of G:

$q\downarrow = true$ if q is an `end` node,

$q\downarrow = q'\downarrow$ if q' is the successor node of some `wait` or action node q,

$q\downarrow = \bigvee\{q'\downarrow,\ q' \text{ a successor of some test node } q\}$,

$q\downarrow = q'\downarrow$ and $q''\downarrow$ if q', q'' are the successors of a `pcall` node q.

Now, seen as a function from Q into $\{false, true\}$, with $false \leq true$, the "$\downarrow$" predicate is the smallest solution of the previous set of equations, and can be computed by the usual fixpoint algorithms.

4 Expressivity

RP schemes and their hierarchical states semantics are an infinite-state model of concurrent behavior. Their expressive power is in some way larger than P/T nets because they allow a parent invocation to wait for the termination of its children. On the other hand, they do not allow synchronization between concurrent components.

In this article, we investigate the expressive power of RP schemes by studying the class $L(RPPS)$ of languages generated by RP schemes. Here the language $L(G)$ generated by a given scheme $G \in RPPS_A$ is understood as the set of traces of all executions (completed behaviours) where τ actions are invisible except when they indicate divergence, i.e. an infinite sequence of τ's at the end of a trace. Hence $L(G) \subseteq A^* \cup A^\omega \cup A^*.\tau^\omega$.

Then we have the following expressivity results:

Theorem 4.1 (i) *$L(RPPS)$ equals $L(PA)$ which are the languages generated by PA (Process Algebra) programs [3,2].*

(ii) *$L(RPPS)$ strictly includes $L(BPP) \cup L(BPA)$ which are the languages generated by BPP (Basic Parallel Processes) and BPA (Basic Process Algebra) programs [4,5].*

(iii) *There is no inclusion between $L(RPPS)$ and $L(PN)$, the class of languages generated by labeled P/T nets (Petri Nets).*

Proof (Sketch)

- (ii) and (iii) are consequences of (i).
- $L(PA) \subseteq L(RPPS)$ is easy to see because one can code the sequential and parallel compositions operations from PA into RP schemes [8]. (A branching-time view of this inclusion is possible, in term of the divergence preserving τ-bisimulation of [11].)
- The reverse inclusion, $L(RPPS) \subseteq L(PA)$, is longer to prove [8]. The idea underlying the proof is better illustrated on an example: consider node q_0 in Fig.3. Here $\xi_0 = q_0, \emptyset$ will spawn a child invocation $\xi_1 = q_1, \emptyset$. ξ_1 will run "in

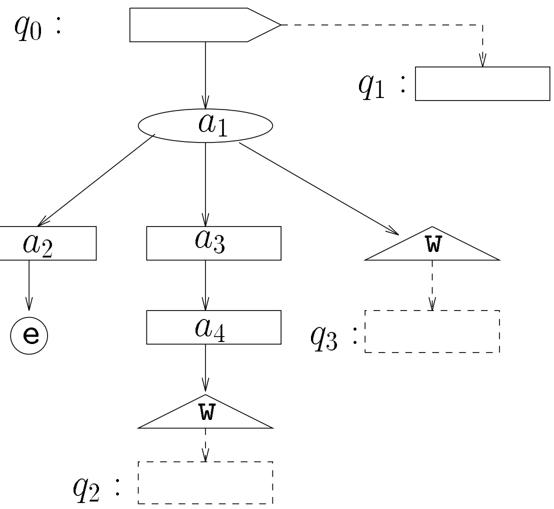

Fig. 3. Example of a `call` node with several continuations

parallel" with ξ_0's continuation, until the continuation enters a `wait` node. At this point, the relationship between ξ_0's continuation and ξ_1 will be a strict sequential composition. Thus a proper encoding of this (fragmentary) RP scheme in PA is

$$\begin{aligned} X_{q_0} &= \Big(a_1.a_2.Nil \mid X_{q_1}\Big) \\ &\quad + \Big(a_1.a_3.a_4.Nil \mid X_{q_1}\Big) . X_{q_2} \\ &\quad + \Big(a_1.Nil \mid X_{q_1}\Big) . X_{q_3} \\ X_{q_1} &= \dots \end{aligned}$$

More generally, to allow possible loops between a node q and one of its derivative `wait` nodes r, we need to introduce new variables X_q^r accounting for all different ways of reaching r (and no other `wait` node). A definition for X_q^r is easy (using the other $X_{q'}^r$). Similarly, we add a simple definition for a new variable X_q^{nw} accounting for all behaviours from q where no `wait` node is ever reached. Then, writing q' for the invoked node from q, and $succ(r)$ for the node following r, we can write the PA definition for a `call` node q as

$$X_q = X_q^{nw} + \sum_r (X_q^r \mid X_{q'}).X_{succ(r)}$$

See [8] for details and a correctness proof. This encoding preserves the generated languages but not the branching-time behavior: the PA process X_{q_0} has to choose immediately which branch will be followed, while state ξ_0 delays this choice. (We conjecture RP schemes are strictly more expressive than PA when branching-time behavior is considered.)

□

5 Well structured RP schemes

Though they are quite powerful, RP schemes can still be analyzed successfully. It turns out that it is possible to give them a structure of well-structured transition system in the sense of [1] (see also [7]).

Well-structured transition systems get their name from their use of a well-ordering between states. Recall that a well-ordering is a partial ordering $(S, \leq)$ s.t. any infinite sequence $s_0, s_1, ..$ contains an infinite increasing subsequence $s_{i_0} \leq s_{i_1} \leq \ldots$ (with $i_0 < i_1 < \ldots$). Two important consequences of well-ordering are:

- any upward-closed set $I \subseteq S$ can be represented by a finite basis $I_0 = \{s_1, \ldots, s_n\}$ of minimal elements s.t. $I = \uparrow(I_0)$ where for any given $M \subseteq S$, $\uparrow(M)$, the *upward-closure of* M, is
$$\uparrow(M) \stackrel{\text{def}}{=} \{s \mid s \geq s_i \text{ for some } s_i \in M\}$$
- any increasing infinite sequence $I_0 \subseteq I_1 \subseteq I_2 \subseteq \cdots$ of upward-closed sets eventually stabilizes at some $I_k = I_{k+1} = I_{k+2} = \ldots$.

$TS(G)$, the transition system associated to some RP scheme G, can be given a well-structure through the following notion of embedding between hierarchical states:

Definition 5.1 (Embedding) A hierarchical state $\xi = \{(q_1, \xi_1), \ldots, (q_n, \xi_n)\}$ is *embedded* into $\xi' = \{(q'_1, \xi'_1), \ldots, (q'_m, \xi'_m)\}$, written $\xi \preceq \xi'$, if

- $\xi \preceq \xi'_j$ for some $j = 1, \ldots, m$, or
- there are some $j_1, \ldots, j_n$ (all distinct) in $\{1, \ldots, m\}$ s.t. for $i = 1, \ldots, n$ we have $(q_i, \xi_i) \preceq (q'_{j_i}, \xi'_{j_i})$, defined as $\begin{cases} q_i = q'_{j_i} \text{ and } \xi_i \preceq \xi'_{j_i}, \\ \text{or } \{(q_i, \xi_i)\} \preceq \xi'_{j_i}. \end{cases}$

This inductive definition yields a well-founded partial ordering, with $\emptyset$ as minimum element. The intuition is that $\xi \preceq \xi'$ when ξ can be obtained by removing nodes in ξ' (and grafting the remaining branches appropriately). By Kruskal's Tree Theorem [10], this order is also a well-ordering.

In our model, the embedding ordering is fundamental because transitions in $TS(G)$ are compatible (in some sense) with it:

Proposition 5.2 (Downward-compatibility for $\preceq$) *For all $\xi_1 \preceq \xi'_1$ and step $\xi'_1 \xrightarrow{\alpha} \xi'_2$, then either $\xi_1 \preceq \xi'_2$, or there is a $\xi_1 \xrightarrow{\alpha} \xi_2$ with $\xi_2 \preceq \xi'_2$.*

We use the notation $\xi[\zeta]$ to denote that ξ contains an occurence of ζ at a given position. $\xi[.]$ is a context in which variants of ζ could be plugged, yielding variants of ξ.

Proof (Sketch) If $\xi_1' \xrightarrow{\alpha} \xi_2'$, then ξ_1' has the form $\xi_1'[q]$ and ξ_2' is $\xi_1'[\zeta]$ with $q \xrightarrow{\alpha} \zeta$ a valid transition. Now if $\xi_1 \preceq \xi_1' = \xi_1'[q]$ then there are two cases. Either $\xi_1 \preceq \xi_1'[\emptyset]$ (i.e. embedding ξ_1 does not need the q component in ξ_1') and then $\xi_1 \preceq \xi_2'$, or ξ_1 has the form $\xi_1[q]$ with $\xi_1[.] \preceq \xi_1'[.]$. Then there is a transition $\xi_1 \xrightarrow{\alpha} \xi_2 = \xi_1[\zeta]$, entailing $\xi_2 \preceq \xi_2'$. □

Write $Post(\xi) \stackrel{\text{def}}{=} \{\xi' \mid \xi \xrightarrow{\alpha} \xi' \text{ for some } \alpha\}$ for the set of all immediate successors of some ξ. Extend to $Post^m(\xi)$ (resp. $Post^*(\xi)$) for the m-steps (resp. many-steps) successors. A consequence of downward-compatibility is

$$\uparrow\!(S) \subseteq \uparrow\!(S') \text{ implies } \begin{cases} \uparrow\!(S \cup Post(S)) \subseteq \uparrow\!(S' \cup Post(S')) \\ \uparrow\!(Post^*(S)) \subseteq \uparrow\!(Post^*(s')) \end{cases} \tag{1}$$

The downward-compatibility property (and some simple effectiveness properties) allow us to state some decidability results, e.g. of some *control-state reachability* problems, as a special case of

Theorem 5.3 (i) *Given a state ξ, we can compute (a finite basis of) the upward-closure of $Post^*(\xi)$.*

(ii) *Then, given any upward-closed set $I \subseteq M(G)$, it can be decided whether all states reachable from ξ are in I.*

Proof. (i) Define a sequence of sets of states with

$$S_0 \stackrel{\text{def}}{=} \{\xi\}, \quad S_{i+1} \stackrel{\text{def}}{=} S_i \cup Post(S_i)$$

Clearly all S_i's are computable because $\to$ is finitely branching and $Post(\zeta)$ is computable for any ζ. $S_i \subseteq S_{i+1}$ entails $\uparrow\!(S_0) \subseteq\uparrow\!(S_1) \subseteq \cdots$ But the $\uparrow\!(S_i)$'s are upward-closed, so that there is some rank k s.t. $\uparrow\!(S_k) =\uparrow\!(S_{k+1}) =\uparrow\!(S_{k+2}) = \dots$ (though perhaps $S_k \neq S_{k+1} \neq \dots$). In fact, as soon as $\uparrow\!(S_k) =\uparrow\!(S_{k+1})$ for some k, (1) entails $\uparrow\!(S_k) = \uparrow\!(S_{k'})$ for all $k' \geq k$. We can effectively compute such a k: this amounts to detecting when two finite sets have a same upward-closure, which is easy when $\preceq$ is decidable.

Now there just remains to see that $\uparrow\!(S_k) =\uparrow\!(Post^*(\xi))$, but this is obvious since $S_i = Post^0(\xi) \cup \cdots \cup Post^i(\xi)$. Finally S_k is a finite basis of $\uparrow\!(Post^*(\xi))$.

(ii). Because I is upward-closed, $Post^*(\xi) \subseteq I$ iff $\uparrow\!(Post^*(\xi)) \subseteq I$. This last condition is easy to check once we have S_k (assuming I is given through a finite basis). □

(Of course, more elaborate implementations of this decision result are possible.)

6 Another well-structured view

Proposition 5.2 introduced downward-compatibility while in the literature, upward-compatibility is more commonly used [7,1]. It turns out that RP schemes also enjoy upward-compatibility. For this we need a more sophisticated ordering taking normedness into account:

Definition 6.1 ($\perp$-embedding) A hierarchical state $\xi = \{(q_1,\xi_1),\dots,(q_n,\xi_n)\}$ is $\perp$-*embedded* into $\xi' = \{(q'_1,\xi'_1),\dots,(q'_m,\xi'_m)\}$, written $\xi \preceq_\perp \xi'$, if

(i) $\xi\downarrow \Leftrightarrow \xi'\downarrow$, and

(ii) • $\xi \preceq_\perp \xi'_j$ for some $j = 1,\dots,m$, or
- there are some $j_1,\dots,j_n$ (all distinct) in $\{1,\dots,m\}$ s.t. for $i = 1,\dots,n$

$$(q_i,\xi_i)\preceq_\perp(q'_{j_i},\xi'_{j_i}),\text{ defined as }\begin{cases} q_i = q'_{j_i} \text{ and } \xi_i \preceq_\perp \xi'_{j_i}, \\ \text{or } \{(q_i,\xi_i)\}\preceq_\perp \xi'_{j_i}. \end{cases}$$

Again, this defines a well-ordering. It is decidable (because being normed is decidable) and is compatible with transitions in the following sense

Proposition 6.2 (Upward-compatibility for $\preceq_\perp$) *For all $\xi_1 \preceq_\perp \xi'_1$ and step $\xi_1 \xrightarrow{\alpha} \xi_2$, there is a $n \geq 2$ and a sequence $\xi'_1 \to \xi'_2 \to \cdots \xi'_{n-1} \xrightarrow{\alpha} \xi'_n$ s.t. $\xi_2 \preceq_\perp \xi'_n$ and $\xi_1 \preceq_\perp \xi'_k$ for $k = 1,\dots,n-1$.*

Proof (Sketch) ξ_1 is some $\xi_1[q]$ s.t. the step $\xi_1 \xrightarrow{\alpha} \xi_2$ is the occurrence of some $q \xrightarrow{\alpha} \zeta$ in the $\xi_1[.]$ context (and then $\xi_2 = \xi_1[\zeta]$). $\xi_1 \preceq_\perp \xi'_1$ means that ξ'_1 can be written as some $\xi'_1[q]$ in a way respecting $\perp$-embedding. In the simplest cases, $q \xrightarrow{\alpha} \zeta$ is possible in the $\xi'_1[.]$ context and we get a step $\xi'_1 \xrightarrow{\alpha} \xi'_2 \stackrel{\text{def}}{=} \xi'_1[\zeta]$. Then it is possible (but tedious) to prove $\xi_2 \preceq_\perp \xi'_2$.

When $q \xrightarrow{\alpha} \zeta$ is not possible in the $\xi'_1[.]$ context, this means that $q \xrightarrow{\alpha} \zeta$ is a `wait` transition, only possible when q has no children, a condition fulfilled by the $\xi_1[.]$, and not the $\xi'_1[.]$, context. Then $\xi'_1[q]$ can be written as $\xi''[q,\zeta_q]$ where ζ_q ($\neq \emptyset$) are the children of q in $\xi'_1[q]$. There $\xi''[q,\emptyset] \xrightarrow{\alpha} \xi''[\zeta,\emptyset]$ is possible. The key argument now is that ζ_q may terminate because $\perp$-embedding requires $\emptyset \preceq_\perp \zeta_q$. So that there is a sequence

$$(\zeta_q =)\ \zeta_1 \to \zeta_2 \cdots \to \zeta_m = \emptyset$$

(obviously with $\emptyset \preceq_\perp \zeta_i$ for $i = 1,\dots,m$) allowing

$$(\xi'_1 =)\ \xi''[q,\zeta_1] \to \xi''[q,\zeta_2] \cdots\ \xi''[q,\emptyset] \xrightarrow{\alpha} \xi''[\zeta,\emptyset]$$

Let $n \stackrel{\text{def}}{=} m+1$ and $\xi'_i \stackrel{\text{def}}{=} \xi''[q,\zeta_i]$. There only remains to check that $\xi_1 \preceq_\perp \xi''[q,\zeta_i]$ for $i = 1,\dots,m$ (a consequence of $\emptyset \preceq_\perp \zeta_i$) and that $\xi_2 \preceq_\perp \xi'_n$. □

The upward-compatibility property (and some simple effectiveness properties) allows us to state some other decidability results, e.g. the decidability of the halting problem, as a special case of

Theorem 6.3 *Given a state ξ, and an upward-closed*[1] *$I \subseteq M(G)$, it is decidable whether all computations eventually reach a state not in I.*

[1] w.r.t. $\perp$-embedding.

Corollary 6.4 *It is decidable whether all computations starting from some state ξ eventually terminate.*

Proof. Indeed, $M(G) \setminus \{\emptyset\}$, the set of non-terminated states, is an upward-closed set (w.r.t. $\perp$-embedding) for which a finite basis is easily constructed. Then we just apply Theorem 6.3. □

Theorem 6.3 extends a similar theorem from [1] to our more general upward-compatibility property. We nevertheless choose to give a complete proof because (1) we find our presentation clearer (also less concerned with algorithmic improvements), and (2) it explains why we need the specific requirement, in Proposition 6.2, that $\xi_1 \preceq_\perp \xi'_k$ for $k = 1, \ldots, n-1$.

We need an auxiliary construction and define $RT(\xi)$, the *reachability tree* starting from ξ, as a rooted tree with nodes labeled by states. More precisely

- Nodes are *live* or *dead*.
- The root is a live node labeled by ξ.
- Any live node labeled by some ζ has sons labeled by the immediate successors (if any) of ζ, i.e. one son for each ζ' s.t. $\zeta \to \zeta'$ in $TS(G)$.
- A son node n labeled by ζ' is live unless we can find a node $n' \neq n$ in the path from the root to node n, labeled with some ξ' s.t. $\xi' \preceq_\perp \zeta'$. Then we say n' *subsumes* n and n is a dead node.

Because $TS(G)$ is a finitely branching transition system, $RT(\xi)$ is a finite tree. This is a classical argument: assume $RT(\xi)$ is infinite, then it has an infinite path (by Koening's Lemma) and, because $\preceq_\perp$ is a well-ordering, we can find along this path an earlier node n' subsuming a later node n. Thus n must be a dead node, a contradiction. Because $RT(\xi)$ is finite, it can be computed effectively.

Now the proof of Theorem 6.3 relies on the following lemma, which translates the initial question into an equivalent one that can be easily decided (by finiteness of $RT(\xi)$) when I is given via a finite basis.

Lemma 6.5 *All computations (in the transition system) starting from ξ eventually reach a state not in I iff all complete paths (in $RT(\xi)$) eventually reach a (node carrying a) state not in I.*

Proof. The "$\Leftarrow$" direction is easy because each computation has a prefix under the form of a complete path in $RT(\xi)$. The "$\Rightarrow$" direction is more involved. Assume, by way of contradiction, that a complete path in $RT(\xi)$ has only nodes labeled with states in I. This path is some $n_0, \ldots, n_k$, labeled by $\zeta_0, \ldots, \zeta_k$. Then we can build a computation $(\xi =)\ \xi_0 \to \xi_1 \to$ where all states are greater (w.r.t. $\preceq_\perp$) than one of the ζ_i's, and thus belong to I.

We build the ξ_i's inductively, starting from $\xi_0 = \xi = \zeta_0$. Assume we have already built $\xi_0, \ldots, \xi_n$. ξ_n is greater than some ζ_i. There are two cases:

- If $i < k$ then there is a step $\zeta_i \to \zeta_{i+1}$. By upward-compatibility, there exists a sequence $\xi_n \to \cdots \to \xi_m$ $(m > n)$ with $\xi_n, \ldots, \xi_{m-1} \succeq_\perp \zeta_i$ and $\xi_m \succeq_\perp \zeta_{i+1}$.

We use them to lengthen our sequence up to ξ_m.

- If $i = k$, then ζ_i is a leaf n_k of $RT(\xi)$. If it is a live node, then ζ_i has no successor, and then $\zeta_i = \emptyset = \xi_n$, so that we have a complete computation.

  If n_k is a dead node, then an earlier ζ_j subsumes ζ_i. And then $\zeta_j \preceq_\perp \xi_n$, so that we are back to the previous case and can lengthen our sequence.

$\square$

Observe that Theorem 6.3 cannot be proven by simply saying that $\rightarrow^+$, the transitive closure of $\rightarrow$, has the compatibility property used in [1]. This fails because

- $\rightarrow^+$ is not finitely branching in general,
- it is not decidable in general,
- more importantly, stating that all "computations" in the $\rightarrow^+$-sense eventually reach a given set is *not equivalent* to the property we are interested in.

7 Related approaches

As far as we know, the RPPS model we introduced is not closely related to other models of concurrency:

- Results from Section 4 indicate a close link between the restricted primitives we allow and the PA fragment of process algebra. However, the process-algebraic view has its own benefits and drawbacks. An advantage of RPPS is that our notion of *hierarchical states* already abstracts from much of the syntax that is required in PA, and directly gives an eye-opening kind of normal form for states.
- In this sense, hierarchical states are more reminiscent of markings in Petri nets. Still, the RPPS model cannot be seen as a special kind of high-level nets [12]. In the general PrTr nets, or in coloured nets, the *nature* of tokens is changed. In RPPS, the "tokens" are not richer. Rather, they are embedded into precise dependency relationships which are carried on through token moves.
- Other models allowing recursivity in parallelism exist, using e.g. denotational semantics, ... In general, they do not try to control how much expressive power is afforded and balance this with decidability issues.
- Perhaps term rewriting systems are the most natural formalism in which we can frame our RPPS proposal. Of course, TRS's are much more general (hence have fewer decidable properties) and not particularly aimed at describing concurrent computations.

Conclusion

We proposed a model for a quite new class of concurrent programs. This models is not finite-state but can be turned into a well-structured transition system, so that the decidability of several interesting problems is easy to prove. The model has no clear connection with other structured transitions systems like P/T nets, lossy channel systems, timed automata, ..., where the well-structure is quite simple to see. We believe it isolates a carefully chosen set of construct for controlling recursivity in parallel systems.

References

[1] P. A. Abdulla, K. Cerans, B. Jonsson, and T. Yih-Kuen. General decidability theorems for infinite-state systems. In *Proc. 11th IEEE Symp. Logic in Computer Science, New Brunswick, NJ*, July 1996.

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

[3] J. A. Bergstra and J. W. Klop. Process theory based on bisimulation semantics. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, Noordwijkerhout, LNCS 354*, pages 50–122. Springer-Verlag, 1989.

[4] S. Christensen. *Decidability and Decomposition in Process Algebras.* PhD thesis, Univ. Edinburgh, September 1993.

[5] S. Christensen, Y. Hirshfeld, and F. Moller. Decidable subsets of CCS. *The Computer Journal*, 37(4):233–242, 1994.

[6] S. Christensen and H. Hüttel. Decidability issues for infinite-state processes - a survey. *EATCS Bull.*, 51:156–166, October 1993.

[7] A. Finkel. Reduction and covering of infinite reachability trees. *Information and Computation*, 89(2):144–179, December 1990.

[8] O. Kouchnarenko. *Sémantique des programmes récursifs-parallèles et méthodes pour leur analyse.* Thèse de Doctorat, Univ. Joseph Fourier-Grenoble I, France, February 1997.

[9] O. Kouchnarenko and Ph. Schnoebelen. Modèles formels pour les programmes récursifs-parallèles. In *Proc. RENPAR'8, Bordeaux*, pages 85–88, May 1996.

[10] J. B. Kruskal. Well-quasi-ordering, the Tree Theorem, and Vazsonyi's conjecture. *Trans. Amer. Math. Soc.*, 95:210–225, 1960.

[11] R. Milner. A modal characterisation of observable machine-behaviour. In *Proc. CAAP'81, Genoa, LNCS 112*, pages 25–34. Springer-Verlag, March 1981.

[12] E. Smith. A survey on high-level Petri-net theory. *EATCS Bull.*, 59:267–293, June 1996.