A Formal Framework for the Analysis of Recursive-Parallel Programs

O. Kushnarenko[1] and Ph. Schnoebelen[2]

[1] IRISA, Univ. Rennes I, Campus de Beaulieu, 35042 Rennes Cedex France
email: Olga.Kouchnarenko@irisa.fr
[2] Lab. Spécification & Vérification, ENS de Cachan & CNRS URA 2236, 61, av. Pdt. Wilson, 94235 Cachan Cedex France
email: phs@lsv.ens-cachan.fr

Abstract. RP programs are imperative programs with parallelism and recursion and only a limited way of synchronizing parallel processes. The *formal framework* we propose here combines (1) a formal operational model of abstract programs (or *RP schemes*), (2) a set of decision methods for the analysis of RP schemes, (3) a formal operational model for the interpreted programs, and (4) translation results stating how some behavioural properties of the concrete programs can be correctly checked on the corresponding scheme.[3]

Introduction

Automated (and computer-assisted) verification and analysis of parallel systems are a fast-growing application field for formal methods and techniques in computer science. This is because parallel systems are notably more difficult to understand for human designers and users while formal analysis is not necessarily more difficult from a structural complexity viewpoint.

Today there exists a very successful approach [BCM+92] to some of these questions: systems are commonly modeled by various types of transition systems. In this framework, most problems of system analysis reduce to various kinds of reachability problems on these models. Unfortunately, this classical approach still has many (widely acknowledged) limitations, making it only well-suited to specific kinds of systems. In particular, its main characteristic is the use of *finite-state* transition systems as a foundation.

Recently, many research groups are working to remove, possibly partially, this limitation. They investigate specific fragments (BPP, PA, ...) of general process algebra, specific subclasses of communicating automata, pushdown automata, special classes of Petri nets, etc. See [Mol96, Esp96] for a survey.

[3] **Keywords:** semantics of concurrency, automated verification of programs, infinite state systems.
Thanks: This research was mainly done while the 2 authors were at the Leibniz-IMAG Lab. in Grenoble.

In [KS96a, Kou97], we proposed and studied a new infinite-state model of concurrent systems with a good balance between expressive power and decidability of behavioral questions. This formal model, called *RP schemes*, was developed as an abstract semantic foundation for the RP programming language [VEKM94]. The analysis of RP programs was in fact the real motivation behind the development of a theory of RP schemes. Now a bridge must be built, linking practical questions about real RP programs and theoretical problems about abstract RP schemes.

In this paper, we present a formal framework linking the two viewpoints. This framework provides for a systematic treatment of how results can be transfered from one viewpoint to the other. The formal framework relies on a general Preservation Theorem between two semantics. Today, it is used as a formal semantical foundation in the development of software tools for the analysis of RP programs. These tools are connected to the RP compiler currently developed by a research team of the Theor. Comp. Sci. Dept. at Yaroslavl State University.

This paper is organized as follows: Sections 1 and 2 succinctly recall the basic concepts of RP programs, RP schemes, and their abstract behavioral semantics $\mathcal{M}$. Section 3 summarizes our main decidability results. In Section 4 we define $\mathcal{M}^I$, the formal model we use for the full interpreted language and give our Preservation Theorem linking $\mathcal{M}$ and $\mathcal{M}^I$. We conclude with several examples, illustrating how specific questions can be transfered from $\mathcal{M}^I$ to the abstract $\mathcal{M}$.

1 Recursive-parallel programs and their schemes

The recursive-parallel (RP) style of programming is a specific approach to the organization of parallel computations. This section gives a short introduction to the basic concepts and notions which will be needed throughout. For a more detailed treatment we refer to [KS96a].

1.1 RP programs

RP programs are written in an imperative programming language supporting parallel coroutines (with recursion) and following a precise discipline for handling parallelism. The language has been developed around the parallel machine of the IPTC Institute in Yaroslavl [MVVK88] and it assumes a shared global memory.

An RP program is essentially a set of nested procedures with the possibility of recursive calls. Fig. 1 contains an example of an abstract RP program, "abstract" because we used abstract action names $a, b, c, \ldots$ from some uninterpreted countable alphabet A instead of the usual basic actions from imperative languages: "`x:=y+2`", ...

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

Fig. 1. An (abstract) RP program

Every call of an RP procedure (via a `pcall` statement) yields a new process that runs in parallel while execution proceeds in the caller (the *parent*). The only synchronization device is the `wait` statement. With this, at any time, a parent invocation may ask to synchronize with (i.e. to wait for) the termination of all the children invocations it has spawned earlier. It is not possible to wait for the termination of only a subset of the children invocations. It is not possible for a child invocation to wait for the termination of its parent invocation.

1.2 RP schemes

RP schemes are a graphical representation of RP programs where the structure of the control flow is made more apparent. An *RP scheme* (over A) is a finite rooted graph G with several kinds of labels that we will not define formally here (see [KS96a]). We let $RPPS_A$ denote the class of all such graphs. Fig. 2 shows the scheme associated to the abstract RP program we saw earlier. 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.

2 Behavioral semantics

In this section, we follow [KS96a] and define a formal notion of behavior for RP schemes. The intention is to formalize the operational meaning of the RP constructs `pcall`, `wait`, etc., but abstracting from the semantics of the individual actions.

We consider a given scheme $G \in RPPS_A$. The behavior of G is given via a labeled transition system $\mathcal{M}_G$. Labels are taken from $A_\tau \stackrel{\text{def}}{=} A \cup \{\tau\}$, where τ is a special name denoting silent, internal computation, and ranged over by $a, \ldots$

First we define what are the states (or configurations) of our system.

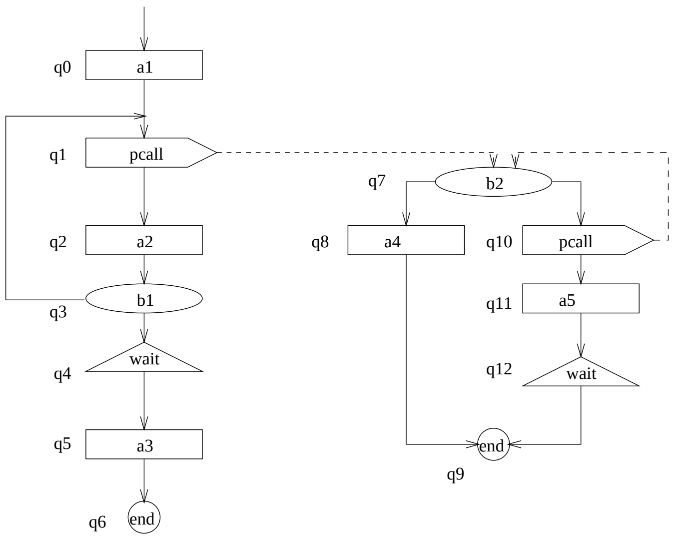

Fig. 2. The scheme associated to the RP program from Fig. 1

Definition 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, $\emptyset \in M(G)$.)

Thus hierarchical states are trees (more generally forests) of nodes from scheme G. (Trees and subtrees are unordered.) But the algebraic structure of nested multisets has its advantages: we use the customary notations "$\xi + \xi'$", "$\xi \subseteq \xi'$", ... to denote addition, inclusion, ... between multisets.

Fig. 3 displays ξ_1, a possible hierarchical state for $M(G)$. In the algebraic

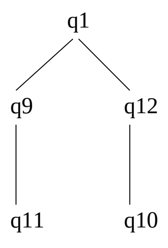

Fig. 3. ξ_1, an example hierarchical state

notation, ξ_1 is written $\{(q_1, \{(q_9, \{(q_{11}, \emptyset)\}), (q_{12}, \{(q_{10}, \emptyset)\})\})\}$. We often omit a few parenthesis and braces when no ambiguity arises and e.g. write $\xi_1 = q_1, \{q_9, \{q_{11}\};\ q_{12}, \{q_{10}\}\}$.

The intuition behind hierarchical states is that $\xi = \{(q_1, \xi_1), \ldots, (q_n, \xi_n)\}$ denotes a configuration where n completely independent concurrent activities are present. One such activity, say (q_i, ξ_i), is the invocation of a coroutine (currently in state/node q_i) together with its family of children invocations. These children are running in parallel and are currently (collectively) in state ξ_i. In our example, ξ_1 has five concurrent components. One, in state q_{11}, depends of its father (currently in state q_9) that itself depends on its father (currently in state q_1). This father invocation has another child invocation (currently in q_{12}) with its own child (currently in q_{10}).

Another view is to see a hierarchical state ξ as the marking of a Petri-like net (with tokens in the $q, q', \ldots$ nodes) *together 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. Fig. 4 displays ξ_1 as a marking of scheme G.

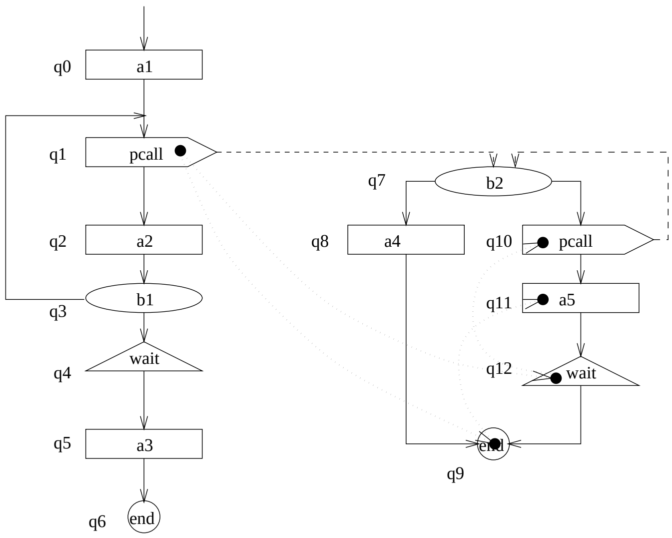

Fig. 4. ξ_1 as a marking of scheme G

The dotted lines between the 5 tokens depict the parent-child hierarchy.

The "hierarchical states as markings" viewpoint helps understand how one moves from a current hierarchical state to a next one. Basically, the tokens move independently (according to the graph structure of G), are created at `pcall` nodes and disappear at `end` nodes. The parent-child relationship between tokens is maintained and updated at all times. A token can only pass a `wait` node when it has no child anymore (i.e. they are terminated, see Prop. 3).

Fig. 5 shows a possible evolution of ξ_1, exemplifying the updating of the parent-child hierarchy. ξ_1 transforms into ξ_2 when its invocation currently in q_{10}

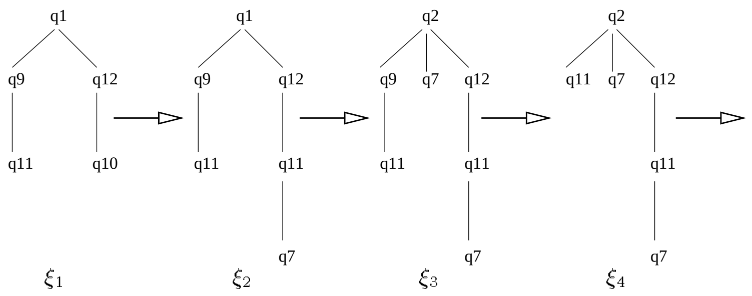

Fig. 5. Example behaviour for hierarchical states

evolves into q_{11}, thereby invoking a new child in q_7. Then ξ_2 transforms into ξ_3 when the parent in q_1 evolves into q_2, thereby invoking a new child in q_7. Then the invocation in q_9 (an `end` node) terminates and disappears, leading from ξ_3 to ξ_4.

We now formally define what are the transitions between the hierarchical states.

Definition 2. The transition system $\mathcal{M}_G \stackrel{\text{def}}{=} \langle M(G), A_\tau, \rightarrow, \xi_0\rangle$ 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 "$\xi \xrightarrow{a} \xi'$") satisfying the following rules:

action: *If q is an a-labeled action (or test) node in G, and q' is a successor node of q, then $q, \xi \xrightarrow{a} q', \xi$ 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 $\xi \xrightarrow{a} \xi'$ then $\xi + \xi'' \xrightarrow{a} \xi' + \xi''$ for all ξ''.*
paral2: *If $\xi \xrightarrow{a} \xi'$ and q is a node of G, then $q, \xi \xrightarrow{a} q, \xi'$.*

Rules for parallelism express that any activity $\xi \xrightarrow{a} \xi'$ 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 children are all terminated (and then not present anymore). The other rules state how children invocations are created and kept around.

We can now formally state that RP schemes have no deadlock:

Proposition 3 [KS96a]. $\xi \not\rightarrow$ *iff* $\xi = \emptyset$.

The expressive power of our RP schemes and their hierarchical states semantics is in some way larger than Petri nets because RP schemes allow a distinction between parent and child invocations. On the other hand, they do not allow arbitrary synchronization between concurrent components.

Formally, we proved in [Kou97, KS96a] that RP schemes and finite PA programs [BK89, BW90] generate the same class of languages while Petri nets and RP schemes generate incomparable classes. (When branching time behavior is taken into account, RP schemes seem more expressive that PA programs but we have no formal proof of this.)

PA is currently under heavy scrutiny [Esp96, Mol96] because it is both quite expressive and still amenable to formal analysis. Thus RP schemes seem to be just expressive enough to offer challenging analysis questions.

3 The analysis of RP schemes

Our first investigations of RP schemes allowed several decidability results [KS96a]. Of course, the expressive power of these schemes is high enough so that many natural questions (language equality, ...) are not decidable.

In this section, we are only interested into recalling the positive decidability results we use in Section 4.

Theorem 4 [Sch96]. *The Reachability Problem:*

input: a scheme G and two states $\xi, \xi' \in M(G)$. **output:** true iff there exists a sequence of transitions in $\mathcal{M}_G$ reaching ξ' from ξ,

the Node Reachability Problem:

input: a scheme G, a node q and a state $\xi \in M(G)$. **output:** true iff from ξ we can reach a state where q occurs,

the Mutual Exclusion Problem:

input: a scheme G, two nodes q, q' and a state $\xi \in M(G)$. **output:** true iff from ξ we never reach a state where both q and q' occur,

and the Boundedness Problem:

input: a scheme G and a state $\xi \in M(G)$.
output: true iff $Reach(\xi)$, the set of all states reachable from ξ, is finite,

are decidable.

We say that a hierarchical state ξ is embedded into ξ', written $\xi \preceq \xi'$, if there exists an embedding between ξ and ξ' seen as labeled forests, i.e. if ξ can be obtained from ξ' by deleting some nodes, preserving the transitive ancestor relationships between the remaining nodes (see [KS96a]). This yields a well-founded partial ordering, with $\emptyset$ as minimum element. By Kruskal's Tree Theorem [Kru60], this is also a well-quasi-ordering.

We say that a set $I \subseteq M(G)$ is *upward-closed* iff $\xi \succeq \xi' \in I$ entails $\xi \in I$. The *upward-closure* of I_0 is the set of all states larger (w.r.t. $\preceq$) than states in I_0. If I_0 is finite, it is a *basis* of its upward-closure. Because $\preceq$ is a well-ordering, any upward-closed set has a finite basis.

Theorem 5 [Kou97, KS96a]. *The Sup-Reachability Problem:*

input: a scheme G and a state $\xi \in M(G)$.
output: a finite basis of the upward-closure of $Reach(\xi)$.

is effectively solvable.

[KS96a] introduces $\perp$-embedding between hierarchical states. This is a finer variant of the earlier tree embedding, this time with gap-conditions. It also yields a decidable well-quasi-ordering.

Theorem 6 [Kou97, KS96a]. *The Inevitability Problem:*

input: a scheme G, a state $\xi \in M(G)$, a finite basis $I_0 \subseteq M(G)$.
output: true iff all computations starting from ξ eventually reach a state not in the upward-closure (w.r.t. $\perp$-embedding) of I.

is decidable.

Corollary 7. *The halting problem (whether all computations starting from a given ξ eventually terminate) is decidable.*

4 A model for interpreted RP programs

In the abstract RP schemes model, basic actions were left uninterpreted. A formal semantics for the full language with interpreted actions can be obtained simply by enriching the hierarchical states viewpoint with memory states and the effect of basic actions upon memory. This way we obtain a formal definition which can be used for the full language and which is based on the same ideas we developed and studied in the simpler abstract framework.

4.1 Interpretation of the basic actions

In the RP language, the memory states have two components: (1) a shared global memory, and (2) for each invocation, a local memory.

Formally, we consider a set (infinite, in general) $GMem = \{u, \ldots\}$ of *shared global memory states*, and a set (infinite, in general) $LMem = \{v, \ldots\}$ of *local memory states* (that is, local to a given coroutine invocation).

The local and global memory states are read and modified by the basic actions of our programs. Formally, all basic actions $a \in A$ are interpreted as mappings $\xmapsto{a}$ from $GMem \times LMem$ into itself. We write $u, v \xmapsto{a} u', v'$ rather than "$\xmapsto{a} (u, v) = (u', v')$". This means that performing a in a configuration where the global (resp. local) memory is in state u (resp. v) changes them (deterministically) to u' and v'. Additionaly, there are some similar $\xmapsto{\texttt{pcall}}$, $\xmapsto{\texttt{wait}}$ and $\xmapsto{\texttt{end}}$ (see § 4.3).

We say that $I = \langle GMem, LMem, (\xmapsto{a})_{a \in A \cup \cdots} \rangle$ is an *interpretation* of the underlying language A. I is *finite* if $GMem$ and $LMem$ are. We let $\mathbf{I}_A = \{I, \ldots\}$ denote the class of all recursive interpretations. Obviously finite interpretations are recursive.

In summary, our basic assumption is that all atomic actions in an RP program (1) have a *deterministic* effect upon the local+shared memory, (2) always terminate properly, and (3) are effective. It would be possible to allow non-deterministic interpretations but we feel this does not respect the basic RPC idea where non-determinism only comes from parallelism.

4.2 Interpreted states

A *state* for an interpreted RP program has the same tree-like structure between parallel invocations we used for abstract RP schemes. Now each invocation is equipped with its local memory state. On top of this, there is one single shared global memory state.

Definition 8. Given an interpreted scheme G, I, the set of its *interpreted hierarchical states* is the least set $M^I(G)$ s.t. if $q_1, \ldots, q_n$ are nodes of G, $v_1, \ldots, v_n \in LMem$ are local memory states, and $\xi_1, \ldots, \xi_n \in M^I(G)$, then the multiset $\xi = \{(q_1, v_1, \xi_1), \ldots, (q_n, v_n, \xi_n)\}$ is in $M^I(G)$. The set of its *interpreted global states* is $GMem \times M^I(G)$.

4.3 Transitions between interpreted states

The transitions between global states are given by rules extending the rules for uninterpreted schemes. The transition rule for action nodes becomes

> **action:** If q is an a-labeled *action* node in G, and q' is the successor node of q, and $u, v \xmapsto{a} u', v'$ then $\langle u, (q, v, \xi)\rangle \xrightarrow{a} \langle u', (q', v', \xi)\rangle$ for all ξ.

Of course, this must be used in combination with rules for parallelism, which become

paral1: If $\langle u, \xi\rangle \xrightarrow{a} \langle u', \xi'\rangle$ then $\langle u, \xi + \xi''\rangle \xrightarrow{a} \langle u', \xi' + \xi''\rangle$ for all ξ''.
paral2: If $\langle u, \xi\rangle \xrightarrow{a} \langle u', \xi'\rangle$ and q is a node of G and v a state of $LMem$, then $\langle u, (q, v, \xi)\rangle \xrightarrow{a} \langle u', (q, v, \xi')\rangle$.

Invocation of coroutines through `pcall` has some effect on the local and the global memory states, but it also yields a new local memory for the invoked process, so that $\overset{\texttt{pcall}}{\longmapsto}$ is a mapping from $GMem \times LMem$ into $(GMem \times LMem) \times LMem$. The transition rule for invocations becomes

call: If q is a `pcall` node in G, with successor node q', invoked node q'' and if $u, v \overset{\texttt{pcall}}{\longmapsto} u', v', v''$ then $\langle u, (q, v, \xi)\rangle \xrightarrow{\tau} \langle u', (q', v', (\xi + (q'', v'', \emptyset)))\rangle$ for all ξ.

The rules for the `wait` and `end` constructs follow the same logic:

wait: If q is a `wait` node in G, and q' is a successor node of q, and $u, v \overset{\texttt{wait}}{\longmapsto} u', v'$ then $\langle u, (q, v, \emptyset)\rangle \xrightarrow{\tau} \langle u', (q', v', \emptyset)\rangle$.
end: If q is an `end` node in G, and $u, v \overset{\texttt{end}}{\longmapsto} u'$ then $\langle u, (q, v, \xi)\rangle \xrightarrow{\tau} \langle u', \xi\rangle$ for all ξ.

The test instructions are not considered as non-deterministic basic actions anymore. Rather, they yield a definite boolean result (depending on u, v) so that if b is a test, $\overset{b}{\longmapsto}$ is a mapping from $GMem \times LMem$ into $GMem \times LMem \times \{\texttt{true}, \texttt{false}\}$. The rule is

test: If q is a b-labeled *test* node in G, and q', q'' are its two successor nodes, and $u, v \overset{b}{\longmapsto} u', v', \texttt{true}$ then $\langle u, (q, v, \xi)\rangle \xrightarrow{b} \langle u', (q', v', \xi)\rangle$ for all ξ. If instead $u, v \overset{b}{\longmapsto} u', v', \texttt{false}$ then $\langle u, (q, v, \xi)\rangle \xrightarrow{b} \langle u', (q'', v', \xi)\rangle$.

Finally, all constructs now behave deterministically and it is the arbitrary interleaving of parallel components which produces a non-deterministic global behavior.

This construction leads to a transition system $\mathcal{M}_G^I$ once we agree on a precise starting state $\langle u_0, (q_0, v_0, \emptyset)\rangle$. Pursuing the "RP schemes as high-level nets" analogy, we could say that $\mathcal{M}_G^I$ is a *colored* version of $\mathcal{M}_G$, where colors are the memory states. However, this coloring mechanism is quite powerful, even if we only consider finite interpretations:

Theorem 9. *RP schemes with finite interpretations are Turing powerful.*

This relies on the fact that the global memory component can be used to synchronize *à la Petri nets* parallel invocations. Thus we obtain the combined power of Petri Nets and BPA synchronization.

Proof of Theorem 9. One can encode any Minsky counter machine into an RP scheme with finite interpretation. For a counter C, an RP procedure is written. This procedure counts by spawning children invocations. When we want to increment the counter, we ask it (through u) to spawn a new child. These children can testify (through u) that C is not zero. Through u, we can ask one (any) of them to terminate, decrementing the value of C. C can implement a (blocking) test for emptiness by using the `wait` construct to check that it has no children anymore.

There exist some strong links between the interpreted model and the abstract model.

Theorem 10 (Preservation Theorem) [Kou97, KS96b]. $\mathcal{M}_G^I \sqsubseteq_d \mathcal{M}_G$

where $\sqsubseteq_d$ is a divergence preserving version of the classical τ-simulation quasi-ordering [Wal88].

This result shows that the abstract model is a correct approximation of the more realistic interpreted model. The interesting point is that it only requires general hypothesis upon the semantics of the basic actions (e.g. they never raise errors). The proof does not need to know what are $GMem$ and $LMem$ precisely, nor did we have to worry about the precise definition of e.g. $\xmapsto{\mathtt{x:=y+3}}$.

It is also interesting to observe that the same $\sqsubseteq_d$ approximation criterion is used in [KS96b, Kou97] to relate $\mathcal{M}_G$ and $\mathcal{P}_G$ (also $\mathcal{M}_G^I$ and $\mathcal{P}_G^I$). $\mathcal{P}_G$ is a third model for RP programs, in which we formally describe the specific implementation strategy for controlling and assigning priorities to a potentially unbounded number of parallel processes on the IPTC parallel machine with only a fixed number of processors.

A corollary of Theorem 10 is that it is possible to investigate properties of some real program by analyzing its abstract model. Of course, only properties *compatible with* $\sqsubseteq_d$ can be handled that way:

Definition 11. A property φ is *compatible* with $\sqsubseteq_d$ iff, for any transition systems $\mathcal{P}, \mathcal{P}'$

$$\mathcal{P} \sqsubseteq_d \mathcal{P}' \text{ and } \mathcal{P}' \models \varphi \text{ entail } \mathcal{P} \models \varphi$$

where $\mathcal{P} \models \varphi$ means that φ holds of $\mathcal{P}$.

Proposition 12. *1. All safety properties are compatible with $\sqsubseteq_d$.*
2. Termination is compatible with $\sqsubseteq_d$.

This gives a general methodology for analyzing RP programs. If a given property is compatible with $\sqsubseteq_d$, it is sufficient to establish it on the abstract $\mathcal{M}_G$ model. Of course the method is not complete (anyway, RP programs are Turing-powerful) and the property may fail on $\mathcal{M}_G$ and still hold of $\mathcal{M}_G^I$.

Properties not compatible with $\sqsubseteq_d$ (e.g. normedness) are mostly interesting if one wants to analyze the uninterpreted model, without aiming at transferring the information to the interpreted model.

5 Analyzing full RP programs

In this section we list a few examples of analysis questions a programmer may ask about his RP programs (i.e. about a given $\mathcal{M}_G^I$) and which may be answered by looking at the abstract $\mathcal{M}_G$ for which decidability results exist.

For each example, it is important to check that (1) the question is relevant from a programmer point of view, (2) it can be translated into a decidable problem for the abstract model, and (3) a positive answer for $\mathcal{M}_G$ entails a positive answer for $\mathcal{M}_G^I$.

Point (3) often involves a slightly more general notion of compatibility where we allow properties to be translated when moving from $\mathcal{M}_G^I$ to $\mathcal{M}_G$. In all examples, establishing the compatibility is quite simple and does not have to consider fine details of the semantics of interpreted actions.

5.1 Node Reachability

A user having written some RP program G is interested into knowing whether all nodes are reachable. Formally, a node q is reachable if q occurs into a reachable state. If q is not reachable then this may be an indication that there is something wrong in the program. We saw in Section 3 that this question is decidable for the abstract model $\mathcal{M}_G$.

Proposition 13. (Correctness) *If node q is not reachable in $\mathcal{M}_G$, it is not reachable in any $\mathcal{M}_G^I$.*

(Completeness) *Conversely, if node q is reachable in $\mathcal{M}_G$, there exists a finite I such that q is reachable in $\mathcal{M}_G^I$.*

Proof. Correctness is clear because when we forget the memory components of a behavior of $\mathcal{M}_G^I$, we get a behavior of $\mathcal{M}_G$.

For completeness, assume that $(\sigma =)\xi_0 \to \ldots \xi_n$ is a behavior of $\mathcal{M}_G$ reaching q (i.e. q occurs in ξ_n). We now build a simple interpretation I where the local memory states are empty and where the global memory state u just stores a natural number, registering the current number of performed steps. So that any action simply increments u. If we try to mimic σ in $\mathcal{M}_G^I$, the only difficulty is to always pick the right branches after test instructions. Because $\xrightarrow{b}$ depends on u, we can code in the $\xrightarrow{b}$'s the "left or right" choice which was actually taken in σ. Because σ is finite, u can be bounded.

5.2 Persistent nodes

We say a set of nodes $P = \{q_1, \ldots, q_n\}$ is *persistent* iff all states reachable (from a given initial state in a given program G) have at least one occurrence of one node in P. In practice, P can be the set of nodes of a given procedure, or the set of nodes in which a given resource is used. If P is persistent, then the procedure is never terminated, the resource is never free.

Persistence is decidable in RP schemes, as a corollary of Theorem 5.

Proposition 14. (Correctness) *If P is persistent in $\mathcal{M}_G$, it is persistent in $\mathcal{M}_G^I$.*
(Completeness) *Conversely, if P is not persistent in $\mathcal{M}_G$, there exists a finite I such that P is not persistent in $\mathcal{M}_G^I$.*

Proof. As Proposition 13.

5.3 Mutual exclusion

The node exclusion problem is very important for RP programs. This has applications in compiler techniques. E.g. listing all nodes of G where a given global variable is assigned new values, and checking that these nodes cannot occur simultaneously in a hierarchical state, we know there will be no write-conflict in the machine hardware. When they can occur simultaneously, this is often an indication of bad programming. We saw in Section 3 that node exclusion is decidable for the abstract model $\mathcal{M}_G$. Now we just need:

Proposition 15. (Correctness) *If nodes q_1 and q_2 are mutually exclusive in $\mathcal{M}_G$, they are mutually exclusive in $\mathcal{M}_G^I$.*
(Completeness) *Conversely, if q_1 and q_2 are not mutually exclusive in $\mathcal{M}_G$, there exists a finite I such that q_1 and q_2 are not mutually exclusive in $\mathcal{M}_G^I$.*

Proof. As Proposition 13.

5.4 Boundedness

Boundedness means that there only exists a finite number of different reachable states, and then that the full behavior of the system can be represented as a finite graph. For many protocols, or controllers, it is expected that only a finite number of configurations can be reached. Boundedness is decidable for abstract RP schemes.

Proposition 16. .

(Correctness) *When $LMem$ and $GMem$ are finite, boundedness of $\mathcal{M}_G$ entails boundedness of $\mathcal{M}_G^I$.*
(Completeness) *Conversely, if $\mathcal{M}_G$ is not bounded, there exist a finite I s.t. $\mathcal{M}_G^I$ is not bounded.*

Proof. Correctness is obvious. For completeness, we assume $(\sigma =)\xi_0 \to \xi_1 \to \cdots$ is an unbounded behavior of $\mathcal{M}_G$ (which must exist by König's Lemma). It is possible [Sch96] to assume σ is of one of the following two forms:

- σ is some
$$\xi_0 \to \cdots \xi_k = \zeta[q, \xi] \to \zeta[..] \to \cdots \xi_l = \zeta[q, \xi + \xi']$$
(with ξ' not empty) and unboundedness comes from iterating indefinitely the $q, \xi \to \cdots q, \xi + \xi'$ part in the $\zeta[.]$ context, or
- σ is some
$$\xi_0 \to \cdots \xi_k = \zeta[q, \xi] \to \zeta[..] \to \cdots \xi_l = \zeta[\zeta'[q, \xi]]$$
(with $\zeta'[.]$ not empty) and unboundedness comes from pumping indefinitely the $\zeta'[.]$ context.

In both cases, one can build a finite interpretation allowing $\mathcal{M}_G^I$ to mimic this unbounded behavior. We only need an empty local memory and a finite global memory storing only a natural number bounded by l. With this, it is easy to encode the necessary "left or right" choices in the $\overset{b}{\longmapsto}$ parts of I.

5.5 Termination

Termination is a compatible property, decidable for abstract RP schemes.

Proposition 17. (Correctness) *If $\mathcal{M}_G$ halts, $\mathcal{M}_G^I$ halts.*
(Completeness) *Conversely, if $\mathcal{M}_G$ does not halt, there exists a finite I such that $\mathcal{M}_G^I$ does not halt.*

Proof. As Proposition 16.

6 Conclusions

We defined a formal semantics for interpreted RP programs and connected the analysis of abstract RP schemes (from [KS96a]) to the analysis of real RP programs. This provides a formal foundation for the software tools currently developed around the RP compiler at Yaroslavl State University.

Today the theory of our RP schemes model has not yet been fully investigated. There exist literally hundreds of behavioral problems for which we do not know yet whether they are decidable or not. Up to now, our approach has been to propose general methods for answering these questions and possibly many related variants. The analysis framework we presented in this paper is a further guide, helping see which questions are more important. Questions about RP schemes are mostly relevant when they can be linked successfully to questions about interpreted RP programs.

References

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[BK89] 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.

[BW90] J. C. M. Baeten and W. P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge Univ. Press, 1990.

[Esp96] J. Esparza. More infinite results. In *Proc. Int. Workshop Verification of Infinite State Systems, Pisa*, pages 4–20, August 1996.

[Kou97] O. Kouchnarenko. *Sémantique des programmes récursifs-parallèles et méthodes pour leur analyse.* Thèse de Doctorat, Univ. Joseph Fourier-Grenoble I, France, February 1997.

[Kru60] J. B. Kruskal. Well-quasi-ordering, the Tree Theorem, and Vazsonyi's conjecture. *Trans. Amer. Math. Soc.*, 95:210–225, 1960.

[KS96a] O. Kouchnarenko and Ph. Schnoebelen. A model for recursive-parallel programs. In *Proc. Int. Workshop Verification of Infinite State Systems, Pisa*, pages 127–138, August 1996.

[KS96b] O. Kouchnarenko and Ph. Schnoebelen. Modèles formels pour les programmes récursifs-parallèles. In *Proc. RENPAR'8, Bordeaux*, pages 85–88, May 1996.

[Mol96] F. Moller. Infinite results. In *Proc. CONCUR'96, Pisa, Italy, LNCS 1119*, pages 195–216. Springer-Verlag, August 1996.

[MVVK88] Y. Mamatov, V. Vasilchikov, S. Volchenkov, and V. Kurchidis. Multiprocessor computer system with dynamic parallelism. Technical Report 7160, VINITI, Moscow, Russia, September 1988.

[Sch96] Ph. Schnoebelen. On the analysis of RP schemes. Unpublished notes, November 1996.

[VEKM94] V. Vasilchikov, V. Emielyn, V. Kurchidis, and Y. Mamatov. *Recursive-parallel programming and work in RPMSHELL.* IPVT RAN, Iaroslavl, Russia, 1994.

[Wal88] D. J. Walker. Bisimulations and divergence. In *Proc. 3rd IEEE Symp. Logic in Computer Science, Edinburgh*, July 1988.

This article was processed using the LaTeX macro package with LLNCS style