Reachability Analysis of Synchronized PA Systems

Ahmed Bouajjani[1], Javier Esparza[2], and Tayssir Touili[3]

[1] LIAFA, University of Paris 7, 2 place Jussieu, 75251 Paris cedex 5, France.
[2] University of Stuttgart, Universitätstr. 38, 70569 Stuttgart, Germany.
[3] School of Comput. Sci., Carnegie Mellon University, Pittsburgh, PA 15213, USA.

Abstract. We present a generic approach for the analysis of concurrent programs with (unbounded) dynamic creation of threads and recursive procedure calls. We define a model for such programs based on a set of term rewrite rules where terms represent control configurations. The reachability problem for this model is undecidable. Therefore, we propose a method for analyzing such models based on computing abstractions of their sets of computation paths. Our approach allows to compute such abstractions as least solutions of a system of (path language) constraints. More precisely, given a program and two regular sets of configurations (process terms) T and T', we provide (1) a construction of a system of constraints which characterizes precisely the set of computation paths leading from T to T', and (2) a generic framework, based on abstract interpretation, allowing to solve this system in various abstract domains leading to abstract analysis with different precision and cost.

1 Introduction

Analyzing and verifying multithreaded programs is nowadays one of the most important problems in program analysis and computer-aided verification. This problem is especially challenging in the case where the programming language allows (1) dynamic creation of concurrent threads, and (2) recursive calls of procedures. It is well known that as soon as synchronization and procedure calls are taken into account, the reachability problem (even of control points) is undecidable (see [Ram00]). Therefore, any analysis or verification algorithm for such programs must consider upper-approximations of the set of possible computation paths.

In a previous work [BET03], we have introduced a generic framework for computing abstractions of the set of paths for a class of multithreaded programs. We have shown that instantiations of this framework lead to several analysis procedures with different precision and cost. In that work, we considered programs without dynamic creation of threads, i.e., programs with recursive procedures but with only a fixed number of communicating threads.

In this paper, we extend our work to the more general case where threads may be created dynamically. For that we consider the approach advocated in [EP00] for modeling and analyzing parallel programs. In [EP00], a framework based on term rewrite systems and automata techniques is used for analyzing parallel programs *without* synchronization. In this paper, we model similarly programs by sets of term rewrite rules, but we take into account synchronizations. More precisely, in our model, the set of terms (defining configurations of the program) is defined by means of (1) process constants corresponding to control points, and composition operators corresponding to (2) sequential composition and (3) CCS-like parallel composition. (A restriction operator is also needed at the top level in order to forbid interleavings between synchronization actions.)

Then, the basic problem we consider is, given two sets of configurations (sets of terms) T_1 and T_2, compute a representation of the set $Paths(T_1, T_2)$ of computation paths leading from a configuration in T_1 to some configuration in T_2. (This allows in particular, but not only, to solve reachability problems by checking the emptiness of this set.) Due to the undecidability result mentioned above, this set cannot be computed precisely, in general. Therefore, our aim is to define a generic method (in the spirit of our previous work [BET03] mentioned above) for effectively computing abstractions $A(T_1,T_2)$ (upper-approximations) of the set of paths $Paths(T_1,T_2)$.

The method we propose in this paper consists in (1) characterizing the set $Paths(T_1,T_2)$ as the least solution of a system of constraints (on path languages), and (2) defining a uniform framework (based on abstract interpretation [CC77]) for computing (in a generic way) abstractions of the least solution of this system of constraints. In the appendix, we give examples of abstractions (with different precision) which can be naturally used in program analysis, and which can be defined as instances of our framework. Moreover, we illustrate the applicability of our techniques and the use of these abstractions on an example of parallel algorithm which computes minimum values of streams of inputs.

Related work: There are several works on static analysis of concurrent programs (see [Rin01] for a survey).

In [BCR01,DBR02], analysis techniques are defined for multithreaded programs without procedure calls (threads are finite-state communicating systems). These techniques are based on solving the coverability problem of Petri nets. This approach is generalized to programs with broadcast communications in [FRSB02] using Petri nets with transfer transitions.

The automata approach for program analysis has been used in [EK99,EHRS00] for programs with procedures (without concurrency). These works are based on computing reachable configurations in pushdown automata [BEM97,FWW97]. This approach has been extended in [LS02,EP00] to parallel programs with dynamic creation of processes, but without synchronization, using as models process rewrite systems called PA processes. In [BT03], we extend this approach to a larger class of processes allowing return values of procedures.

In [BET03], we use path abstractions to analyze parallel recursive programs (with synchronization). In that paper we use communicating pushdown automata as formal model of programs and build abstractions of context-free path languages based on our automata-based procedures for reachability analysis of pushdown automata [BEM97,EHRS00]. A different approach for analyzing parallel programs with procedures using path language abstractions is presented in [FQ03].

In [SS00,MO02], similar approaches to the one we propose here are defined. In both papers, the authors define sets of constraints characterizing sets of computation paths. However, these characterizations are technically different from ours, and consider a more restricted setting. (1) These works consider the problem of computing abstractions of the set of paths starting from one single initial configuration to the set of all reachable configurations, whereas in our approach, the set of initial configurations and target configurations can be any regular sets of configurations. This allows us to deal in a uniform way with the analysis problem of various properties (as it can be seen for instance in the example of Section B.2). (2) The work in [SS00] (like the one in [EP00]) does not consider synchronizations, whereas the aim of our work is to consider synchronizations in presence of dynamic creation of processes and procedure calls. Finally,(3) the work in [MO02] is focused on a particular dataflow analysis problem (constant detection), whereas our approach intends to deal uniformly with safety properties. It must also be said that we pay a price for our more general setting, namely the higher complexity of some of our abstractions.

Another work which considers the abstract analysis of concurrent programs in presence of dynamic creation of threads and procedures is [DS91]. The paper provides an (ad-hoc) approximate analysis for determining which statements can be concurrently executed. We think that the approximation used in that work could be phrased in our framework, but a careful comparison of our two approaches needs to be done.

Finally, in [QRR04], procedure summaries are used to represent the effect of executing a procedure. The approach works on the concrete multithreaded program (no abstraction is required). The analysis algorithm is only guaranteed to terminate in some specific cases.

2 Synchronized PA systems

We introduce hereafter a process algebra-based model for multithreaded programs with recursive calls. It consists in extending PA processes [BW90] with synchronization actions.

2.1 Syntax

Let $Lab = \{a, b, c, \ldots\}$ be a set of visible actions. Let *Sync* and *Async* be two disjoint sets such that $Lab = Sync \cup Async$. We assume that to each action $a \in Sync$ corresponds a co-action $\bar{a}$ in *Sync* such that $\bar{\bar{a}} = a$. Intuitively, *Sync* is the set of all synchronization actions, i.e., actions which must be performed simultaneously with their corresponding co-actions in a "handshake" between two parallel processes. Let $Act = Lab \cup \{\tau\}$ be the set of all the actions, where τ is a special internal action (as we shall see, this special action will represent the handshakes). Let $Var = \{X, Y, \ldots\}$ be a set of process constants. Then, we define $\mathcal{T}$ to be the set of process terms t given by:

$$t ::= 0 \mid X \mid t \cdot t \mid t \| t$$

Intuitively, 0 is the idle or terminated process (also called null process), and "." (resp. "$\|$") corresponds to the sequential composition (resp. parallel composition).

The set of *restricted process terms* is defined as $\mathcal{T}_r = \{t \backslash Sync \mid t \in \mathcal{T}\}$. The term "$t \backslash Sync$" corresponds to the restriction of the behavior of t to the non-synchronizing actions. Given a set of process terms T, we denote by $T \backslash Sync$ the set $\{t \backslash Sync \mid t \in T\}$.

Definition 1. *A* Synchronized PA system *(SPA for short) is is a finite set R of rules of the form $X \overset{a}{\hookrightarrow} t$, where $t \in \mathcal{T}$ and $a \in Lab$.*

2.2 Semantics

Structural equivalences on terms: Terms are considered modulo structural equivalences correponding to the following properties: neutrality of the null process "0" w.r.t. "$\cdot$" and "$\|$", the associativity of "$\cdot$" and "$\|$", and the commutativity of "$\|$". We consider the equivalence relation $\sim_0$ on $\mathcal{T}$ defined by:

A1: $\quad t \cdot 0 \sim_0 0 \cdot t \sim_0 t \| 0 \sim_0 0 \| t \sim_0 t$

We also consider the structural equivalence $\sim$ generated by (A1) and:

A2:	$(t \cdot t') \cdot t'' \sim t \cdot (t' \cdot t'')$	: associativity of ".",
A3:	$t \| t' \sim t' \| t$	: commutativity of "$\|$",
A4:	$(t \| t') \| t'' \sim t \| (t' \| t'')$	: associativity of "$\|$".

The equivalences above are extended to terms of $\mathcal{T}_r$ by considering that $t \backslash Sync \equiv t' \backslash Sync$ iff $t \equiv t'$.

Let $\equiv$ be an equivalence from the set $\{=, \sim\}$, where $=$ stands for the identity between terms. Let $t \in \mathcal{T}_r$, we denote by $[t]_\equiv$ the equivalence class modulo $\equiv$ of the process term t, i.e., $[t]_\equiv = \{t' \in \mathcal{T}_r \mid t \equiv t'\}$. A set of terms L is said to be compatible with the equivalence $\equiv$ if $[L]_\equiv = L$. We say that L' is a $\equiv$-representative of L if $[L']_\equiv = L$.

Transition relations and computations: An SPA R induces a transition relation $\xrightarrow{a}$ over $\mathcal{T} \cup \mathcal{T}_r$ defined by the following inference rules:

$$\theta_1 : \frac{X \overset{a}{\hookrightarrow} t_2 \in R}{X \xrightarrow{a} t_2}; \quad \theta_2 : \frac{t_1 \xrightarrow{a} t_1'}{t_1 \cdot t_2 \xrightarrow{a} t_1' \cdot t_2}; \quad \theta_3 : \frac{t_1 \sim_0 0\ , \quad t_2 \xrightarrow{a} t_2'}{t_1 \cdot t_2 \xrightarrow{a} t_1 \cdot t_2'}$$

$$\theta_4 : \frac{t_1 \xrightarrow{a} t_1'}{t_1 \| t_2 \xrightarrow{a} t_1' \| t_2}; \quad \theta_5 : \frac{t_1 \xrightarrow{a} t_1'; \quad t_2 \xrightarrow{\overline{a}} t_2'; \quad a \in Sync}{t_1 \| t_2 \xrightarrow{\tau} t_1' \| t_2'}; \quad \theta_6 : \frac{t_1 \xrightarrow{a} t_2; \quad a \notin Sync}{t_1 \backslash Sync \xrightarrow{a} t_2 \backslash Sync}$$

Each equivalence $\equiv \in \{=, \sim\}$ induces a transition relation $\xrightarrow{a}_{\equiv}$ over $\mathcal{T} \cup \mathcal{T}_r$ defined as follows:

$$\forall t, t', t \xrightarrow{a}_{\equiv} t' \text{ iff } \exists u, u' \text{ such that } t \equiv u, u \xrightarrow{a} u', \text{ and } u' \equiv t'$$

The relation $\xrightarrow{a}_{\equiv}$ is extended to sequences of actions in the usual way. For every term $t \in \mathcal{T} \cup \mathcal{T}_r$, let $Post^*_{\equiv}[w](t) = \{t' \in \mathcal{T} \cup \mathcal{T}_r \mid t \xrightarrow{w}_{\equiv} t'\}$ and let $Post^*_{\equiv}(t) = \bigcup_{w \in Act^*} Post^*_{\equiv}[w](t)$. These two definitions are extended to sets of terms as usual.

Now, we consider also a *weak* transition relation $\Rightarrow_a$ over $\mathcal{T}$ defined by the inference rules θ_1, θ_2, θ_3, and θ_4 (i.e., synchronization and restriction rules are ignored). This relation defines a semantics for SPA processes which is precisely the one of PA processes. As above, we consider also the relations $\overset{a}{\Rightarrow}_{\equiv}$ induced by the equivalences $\equiv$ defined in the obvious way, and we define for every term $t \in \mathcal{T}$, $WPost^*_{\equiv}[w](t) = \{t' \in \mathcal{T} \mid t \overset{w}{\Rightarrow}_{\equiv} t'\}$ and $WPost^*_{\equiv}(t) = \bigcup_{w \in Act^*} WPost^*_{\equiv}[w](t)$.

Given two sets of terms $T, T' \subseteq \mathcal{T} \cup \mathcal{T}_r$, the set of *computation paths* leading from T to T' is defined by $Paths_R(T, T') = \{w \in Act^* \mid \exists t \in T, \exists t' \in T', t' \in Post^*_{\sim}[w](t)\}$. We define similarly the set $WPaths_R(T, T')$, when $T, T' \subseteq \mathcal{T}$, by considering the $WPost^*$ relation instead of $Post^*$.

2.3 SPA as a model of multithreaded programs

From programs to SPA systems: Programs represented by parallel flow graph systems (see e.g., [EP00,SS00,MO02]) can be translated straightforwardly to SPA systems. (We assume as usual that infinite data types have been abstracted into finite types using standard techniques of abstract interpretation.) Nodes of the flow graphs (corresponding to control points in the programs, coupled with abstract values of local variables) are represented by process constants, and actions of the programs are modeled by means of process term rewrite rule. Rules of the form $X \overset{a}{\hookrightarrow} X_1 \cdot X_2$ correspond to procedure calls, and rules of the form $X \overset{a}{\hookrightarrow} X_1 \| X_2$ correspond to dynamic creation of parallel processes. Complementary actions a, $\overline{a}$ are used to model synchronizations between parallel processes (they correspond to send (a!) and receive (a?) statements). Therefore, we consider that the set of synchronizing actions $Sync$ is the set $\{a, \overline{a} \mid \text{a is a communication channel}\}$.

The initial configurations of a program are represented by a set T of process terms in $\mathcal{T}$. The behavior of the program corresponds to the set of computation paths of its SPA model R, starting from the set of restricted terms $T \backslash Sync$, i.e., $Paths_R(T \backslash Sync, \mathcal{T}_r)$.

Well formed systems: A quite natural requirement on programs is that complementary synchronization actions can only appear in parallel processes (they can never be executed sequentially by the same thread). This requirement is easy to guarantee for programs with a fixed number of parallel processes. It suffices to consider that each pair of processes communicate through distinguished directed channels. However, this requirement becomes hard to guarantee in the case of programs with dynamic creation of processes. We introduce hereafter a syntactical condition on SPA systems which ensures this property.

Let R be an SPA modeling a program as described above. We associate with R a dependency graph $\mathcal{G}_R$ defined as follows. Vertices are either process constants, or intermediate vertices (one for each rule in R). There is an edge $X \xrightarrow{a} Y$ for every rule $X \overset{a}{\hookrightarrow} Y$. For every rule $X \overset{a}{\hookrightarrow} X_1 op X_2$, where $op \in \{\cdot, \|\}$, there are three edges $X \xrightarrow{a} v$, $v \xrightarrow{op} X_1$, and $v \xrightarrow{op} X_2$, where v is a fresh vertex.

We say that an SPA is *well formed* if it satisfies the following condition: For every two transitions $u_1 \xrightarrow{a} u_2$ and $v_1 \xrightarrow{\overline{a}} v_2$ in $\mathcal{G}_R$, every simple path in the undirected graph corresponding to $\mathcal{G}_R$ relating u_1 and v_1 must contain an edge labelled by $\|$. It is easy to check that well formed systems satisfy the property that complementary synchronization actions can never be executed by the same sequential process. We can then show that:

Lemma 1. *If R is a well formed SPA, then for every terms t and t' in $\mathcal{T}$, we have:*

$$t \xrightarrow{\tau}_{\equiv} t' \text{ iff } \exists a \in Sync, t \overset{a\bar{a}}{\Rightarrow}_{\equiv} t'$$

Example We consider in Appendix B an example of a parallel algorithm which computes minimum values of streams of inputs, and show how it can be translated into a well-formed SPA model.

3 The Reachability Problem for SPA systems

Let R be an SPA system. The problem we consider is, given two regular (finite tree-automata definable, see definition later), potentially infinite, sets of process terms $T, T' \subseteq \mathcal{T}$, check whether:

$$Paths_R(T \backslash Sync, T' \backslash Sync) \overset{?}{=} \emptyset \tag{1}$$

Unfortunately, we can show that the halting problem of a two counter machine can be reduced to this problem.

Theorem 1. *The reachability problem of SPA systems is undecidable.*

To tackle the problem (1), we adopt an abstraction-based approach consisting as usual in checking stronger conditions, i.e., checking the emptiness of larger sets than $Paths_R(T \backslash Sync, T' \backslash Sync)$. The originality of our approach is that it allows to consider in a generic way several kinds of abstractions.

To explain our approach, we need to reformulate the problem (1) above. It is easy to see that $Paths_R(T \backslash Sync, T' \backslash Sync) = Paths_R(T, T') \cap (Async \cup \{\tau\})^*$ and therefore, solving (1) is equivalent to checking whether

$$Paths_R(T, T') \cap (Async \cup \{\tau\})^* \overset{?}{=} \emptyset \tag{2}$$

Moreover, for the class of well formed SPA systems, Lemma 1 implies that (2) is equivalent to checking whether

$$WPaths_R(T, T') \cap (Async \cup \sum_{a \in Sync} a\,\bar{a})^* \overset{?}{=} \emptyset \tag{3}$$

Due to the undecidability result above, both $Paths_R(T, T')$ and $WPaths_R(T, T')$ cannot be effectively computed as objects of any decidable class of word automata or grammars. Therefore, the question we address is how to compute *abstractions* of the path languages $Paths_R(T, T')$ and $WPaths_R(T, T')$, i.e., upper-approximations $A(T, T')$ of the set $Paths_R(T, T')$ (resp. $WPaths_R(T, T')$), such that the emptiness of the set $A(T, T') \cap (Async \cup \{\tau\})^*$ (resp. $A(T, T') \cap (Async \cup \sum_{a \in Sync} a\,\bar{a})^*$) can be decided.

We define a generic approach for computing abstractions of the sets $Paths_R(T, T')$ and $WPaths_R(T, T')$ based on (i) characterizing each of $Paths_R(T, T')$ and $WPaths_R(T, T')$ as the least solution of a system of constraints on word languages (this solution cannot be computed in general as said before), and (ii) computing the least solution of the system of constraints in an abstract domain to obtain an upper-approximation of $Paths_R(T, T')$ or $WPaths_R(T, T')$.

Remark 1. We will see later that the two formulations (2) and (3) above lead to complementary analysis approaches. Indeed, they allow to consider different abstractions with uncomparable precisions.

In the sequel, we assume that T' is a $\sim$-compatible set. In that case, it is possible to show that the sets $Paths_R(T, T')$ and $WPaths_R(T, T')$ can be precisely characterized without taking into account the structural equivalences on terms:

Proposition 1. *For every $T, T' \subseteq \mathcal{T}$, if T' is a $\sim$-compatible set, then $(W)Paths_R(T,T') = \{w \in Act^* \mid (W)Post^*_{=}[w](T) \cap T' \neq \emptyset\}$.*

Based on the proposition above, we provide a characterization of $(W)Paths_R(T,T')$ as the least solution of a set of constraints (on sets of finite words). This set of constraints is built from finite tree-automata representations of the two given sets of terms T and T'. The next section shows this characterization in detail.

4 Characterizing Path Languages

We start by introducing some preliminary definitions concerning the automata representations we use, and establish some links between operations on term processes and operations on computation paths.

4.1 Process tree automata

Terms in $\mathcal{T}$ can be seen as binary trees where the leaves are labeled with process constants, and the inner nodes with the binary operators "$\cdot$" and "$\|$". Therefore, regular sets of process terms in $\mathcal{T}$ can be represented by means of a kind of finite bottom-tree automata, called *process tree automata*, defined as follows:

Definition 2. *A* **process tree automaton** *is a tuple $A = (Q, Var, F, \delta)$ where Q is a finite set of states, Var is a set of process constants, $F \subseteq Q$ is a set of final states, and δ is a set of rules of the form (a) $f(q_1,q_2) \rightarrow_\delta q$, (b) $X \rightarrow_\delta q$, or (c) $q \rightarrow_\delta q'$, where $X \in Var$, $f \in \{\|, \cdot\}$, and $q_1,q_2,q,q' \in Q$.*

In the sequel, a term of the form $t_1 \cdot t_2$ (resp. $t_1 \| t_2$) will also be represented by $\cdot(t_1,t_2)$ (resp. $\|(t_1,t_2)$). Let t be a process term. A run of A on t is defined in a bottom-up manner as follows: first, the automaton annotates the leaves according to the rules (b), then it continues the annotation of the term t according to the rules (a) and (c): if the subterms t_1 and t_2 are annotated by the states q_1 and q_2, respectively, and if the rule $f(q_1,q_2) \rightarrow_\delta q$ is in δ then the term $f(t_1,t_2)$ is annotated by q, where $f \in \{\|, \cdot\}$. A term t is accepted by a state $q \in Q$ if A reaches the root of t in q. Let L_q be the set of terms accepted by q. The language accepted by the automaton A is $L(A) = \bigcup\{L_q \mid q \in F\}$. A set of process terms is regular if it is accepted by a process tree automaton.

Proposition 2. *[CDG+97] The class of regular process tree languages is closed under boolean operations. Moreover, it can be decided in linear time whether the language accepted by a process tree automaton is empty.*

4.2 Process Composition vs. Computation Path Composition

In order to characterize the set of computation paths, we need to associate with the operators "$\cdot$" and "$\|$" on processes corresponding operators on computation paths. The following lemma shows the link between the sequential composition of processes and the concatenation of computation paths.

Lemma 2. *For every $s_1, s_2, t_1, t_2 \in \mathcal{T}$, and every $w \in Act^*$, $s_1 \cdot s_2 \in Post^*[w](t_1 \cdot t_2)$ iff $\exists w_1, w_2 \in Act^*$ such that $w = w_1 w_2$ and, $s_1 \in Post^*[w_1](t_1)$, $s_2 \in Post^*[w_2](t_2)$, and either $s_1 \sim 0$, or $w_2 = \varepsilon$.*

Depending on which semantics we associate with the parallel operator, we must consider two different operators on paths. For the "strong" semantics, we introduce an operator "$|||$" defined inductively as follows:

$$\begin{aligned} \varepsilon ||| w = w ||| \varepsilon &= w \\ aw_1 ||| \bar{a}w_2 &= a(w_1 ||| \bar{a}w_2) + \bar{a}(aw_1 ||| w_2) + \tau(w_1 ||| w_2) \\ aw_1 ||| bw_2 &= a(w_1 ||| bw_2) + b(aw_1 ||| w_2) \;\text{ if } b \neq \bar{a} \end{aligned}$$

The proof of the following lemma can be found in the appendix:

Lemma 3. *For every $s_1, s_2, t_1, t_2 \in \mathcal{T}$, and every $w \in Act^*$, $s_1 \| s_2 \in Post^*_R[w](t_1 \| t_2)$ iff $\exists w_1, w_2 \in Act^*$ such that $w \in w_1 ||| w_2$, $s_1 \in Post^*_R[w_1](t_1)$, and $s_2 \in Post^*[w_2](t_2)$.*

In the case of the weak semantics (where $\|$ corresponds to pure interleaving without synchronization), the associated operation is the shuffle operation $\sqcup\!\sqcup$ on words. The lemma above holds when $Post^*$ is replaced by $WPost^*$, and $|||$ is replaced by $\sqcup\!\sqcup$.

4.3 Fixpoint Characterization of $(W)Paths_R(T, T')$:

Let R be a SPA system, let T and T' be two regular sets of process terms, and let $A = (Q, \Sigma, F, \delta)$ and $A' = (Q', \Sigma, F', \delta')$ be two process tree automata such that $L(A) = T$ and $L(A') = T'$. We assume w.l.o.g. that for every $s \in Q'$, there is a state $s^\perp \in Q'$ such that $L_{s^\perp} = L_s \cap \{t \in \mathcal{T} \mid t \sim_0 0\}$. (We consider that $s^{\perp^\perp}$ is the same state as $s^\perp$.) [1]

Then, let us consider the problem of characterizing $Paths_R(T, T')$. The characterization of $WPaths_R(T, T')$ can be done exactly in the same manner, by replacing everywhere $Post$ with $WPost$, and the operator $|||$ with $\sqcup\!\sqcup$.

We introduce slight extensions of the automata A and A' by adding states and rules corresponding to the terms appearing in R. For that, let us consider the set $Q^R = \{q_t \mid t$ is a subterm of a term appearing in some rule of $R\}$ and let us define δ^R to be the set of rules:

- $X \to q_X$ if $q_X \in Q^R$, for $X \in Var$,
- $\|(q_{t_1}, q_{t_2}) \to q_t$ if $t = \|(t_1, t_2)$ and $q_t \in Q^R$,
- $\cdot(q_{t_1}, q_{t_2}) \to q_t$ if $t = \cdot(t_1, t_2)$ and $q_t \in Q^R$.

It is easy to see that, for every subterm t appearing in R, we have $L_{q_t} = \{t\}$. Now, let $\mathcal{Q} = Q \cup Q^R$, $\Delta = \delta \cup \delta^R$, $\mathcal{Q}' = Q' \cup Q^R$, and $\Delta' = \delta' \cup \delta^R$. Then, given two states $q \in \mathcal{Q}$ and $s \in \mathcal{Q}'$, we define the set of paths:

$$\lambda(q, s) = \{w \in Act^* \mid Post^*_{=}[w](L_q) \cap L_s \neq \emptyset\}.$$

Clearly, the computation of the sets $\lambda(q, s)$ allows to define the set $Paths_R(T, T')$ since, due to Proposition 1, this set is simply the union of all $\lambda(q, s)$ such that $q \in F$ and $s \in F'$.

A Set of Constraints: We give hereafter a set of constraints on sets of words (path languages) and prove that these constraints define precisely the sets $\lambda(q, s)$. For that, we consider a set of variables representing sets of words defined as follows: For every state $q \in \mathcal{Q}$ and every state $s \in \mathcal{Q}'$, we consider a variable $V(q, s)$. Then, we define the following set of constraints:

(β_1) If $L_q \cap L_s \neq \emptyset$, then

$$\varepsilon \in V(q, s)$$

(β_2) If $q_1 \to q_2$ is a rule of Δ and $s_1 \to s_2$ is a rule of Δ', then

$$V(q_1, s_1) \subseteq V(q_2, s_2)$$

(β_3) If $\cdot(q_1, q_2) \to q$ is a rule of Δ and $\cdot(s_1, s_2) \to s$ is a rule of Δ', then

$$V(q_1, s_1^\perp) V(q_2, s_2) \subseteq V(q, s)$$

and, if $L_{q_2} \cap L_{s_2} \neq \emptyset$, then

$$V(q_1, s_1) \subseteq V(q, s)$$

[1] Such states can be obtained by taking a product of A' with an automaton which recognizes the set of terms $\sim_0$-equivalent to 0. The rule of this automaton are: $0 \to q_\perp$, $\cdot(q_\perp, q_\perp) \to q_\perp$, and $\|(q_\perp, q_\perp) \to q_\perp$, where $q_\perp$ is the only state of the automaton, considered as an accepting state. The state $s^\perp$ corresponds to $(s, q_\perp)$ in the product automaton.

(β_4) If $\|(q_1,q_2) \to q$ is a rule of Δ and $\|(s_1,s_2) \to s$ is a rule of Δ', then

$$V(q_1,s_1)|||V(q_2,s_2) \subseteq V(q,s)$$

(β_5) If $X \overset{a}{\hookrightarrow} t \in R$, then

$$V(q,q_X)aV(q_t,s) \subseteq V(q,s)$$

Let us explain the meaning of the constraints above. Remember that the variables $V(q,s)$ are meant to represent the sets $\lambda(q,s)$. The rule (β_1) says that if there is a term which belongs to both the source set L_q and the target set L_s, then the empty sequence ε must be in the set of paths $\lambda(q,s)$. The rules (β_2) express the fact that if $L_{q_1} \subseteq L_{q_2}$ and $L_{s_1} \subseteq L_{s_2}$, then $\lambda(q_1,s_1) \subseteq \lambda(q_2,s_2)$. The rule ($\beta_3$) says that: (1) for every $u_1 \in L_{s_1}$ such that u_1 is equivalent to 0 (accepted at $s_1^{\perp}$), and which is reachable from some $v_1 \in L_{q_1}$ by some computation path w_1, and for every $u_2 \in L_{s_2}$ which is reachable from some $v_2 \in L_{q_2}$ by some computation path w_2, the term $\cdot(u_1,u_2)$ (accepted at s due to the rule $\cdot(s_1,s_2) \to s$) is reachable from the term $\cdot(v_1,v_2)$ (accepted at q due to the rule $\cdot(q_1,q_2) \to q$) by the computation path w_1w_2. Moreover, (2) for every $u_1 \in L_{s_1}$ which is not equivalent to 0, and which is reachable from some $v_1 \in L_{q_1}$ by some computation path w, and for every term u_2 which belongs to both L_{q_2} and L_{s_2}, the term $\cdot(u_1,u_2)$ (accepted at s) is reachable from the term $\cdot(v_1,u_2)$ (accepted at q) by the computation path w. (This corresponds to the case where no rewriting step can be performed at the right of the "$\cdot$" operator since its left operand is not reduced to the terminated process). The rule (β_4) is similar to (β_3), but concerns parallel terms. It is simpler because parallel composition is commutative, and hence, rewriting steps can be performed on both sides of the "$\|$" operator. Finally, the rule (β_5) says that, if the constant X is reachable from some term v in L_q by some path w_1 (i.e., $w_1 \in \lambda(q,q_X)$), and if there is a path w_2 from the term t to some term $u \in L_s$ (i.e., $w_2 \in \lambda(q_t,s)$), then u is reachable by the path w_1aw_2 from v.

Correctness: We show that (i) the least solution of the previous set of constraints exists, and (ii) that this solution corresponds precisely to the defi nition of the sets $\lambda(q,s)$.

Proposition 3. *The least solution of the set of constraints* (β_1)–(β_5) *exists.*

Indeed, let $x_1,\ldots,x_m$ be an arbitrary numbering of the variables $V(q,s)$ for $q \in \mathcal{Q}$ and $s \in \mathcal{Q}'$. Then, the system (β_1)–(β_5) is a set of inclusion constraints of the form

$$f_i(x_1,\ldots,x_m) \subseteq x_i, \quad 1 \le i \le m \tag{4}$$

where the $f_i(x_1,\ldots,x_m)$'s are functions built up from the variables x_i's, and the operators of word concatenation, $|||$, and $\cup$. (Observe that two different inclusions of the form $e_1 \subseteq x_i$ and $e_2 \subseteq x_i$ can be replaced by the inclusion $e_1 \cup e_2 \subseteq x_i$.)

Let then $\mathbf{X} = (x_1,\ldots,x_m)$, and F be the function

$$F(\mathbf{X}) = \big(f_1(x_1,\ldots,x_m),\ldots,f_m(x_1,\ldots,x_m)\big).$$

The least solution of (4) is the least pre-fi xpoint of F. Let $\mathcal{L}$ be the complete lattice of languages over Act, i.e., $\mathcal{L} = (2^{Act^*},\subseteq,\cup,\cap,\emptyset,Act^*)$. It can be shown that the operators $\cdot$ and $|||$ are $\cup$-continuous. It follows that F is monotonic and $\cup$-continuous. Therefore, by Tarski's theorem, the least pre-fi xpoint of F exists and is equal to its least fi xpoint, and by Kleene's theorem this fi xpoint is equal to:

$$\bigcup_{i\ge 0} F^i(\emptyset,\ldots,\emptyset). \tag{5}$$

Let $\big(L(q,s)\big)_{q\in\mathcal{Q},s\in\mathcal{Q}'}$ be the least solution of the system (β_1)–(β_5). The proof of this theorem is in the appendix:

Theorem 2. *For every $q \in \mathcal{Q}$ and every $s \in \mathcal{Q}'$, we have $L(q,s) = \lambda(q,s)$.*

5 Abstracting Path Languages

The iterative computation (5) of the least solution of the system (4) does not terminate in general (since the reachability problem is undecidable for SPAs). As explained before, instead of computing the exact languages $\lambda(q,s)$, our approach consists in computing abstractions of them. To describe these abstractions, we define a formal framework based on abstract interpretation [CC77].

5.1 A Generic Framework

Let $\mathcal{L}$ be the complete lattice of languages over *Act*, i.e., $\mathcal{L} = (2^{Act^*}, \subseteq, \cup, \cap, \emptyset, Act^*)$. Formally, an abstraction requires an *abstract lattice* $\mathcal{D} = (D, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$, where D is some abstract domain, and a *Galois connection* (α,γ) between $\mathcal{L}$ and $\mathcal{D}$, i.e., a pair of mappings $\alpha : 2^{Act^*} \to D$ and $\gamma : D \to 2^{Act^*}$ such that

$$\forall x \in 2^{Act^*},\ \forall y \in D.\ \alpha(x) \sqsubseteq y \Longleftrightarrow x \subseteq \gamma(y)\,.$$

In our framework, $\sqcup$ is associative, commutative, and idempotent. We assume also that this operator can be extended to countably infinite sets (i.e., countably infinite joins are also elements of D). Moreover, we consider two abstract operations $\otimes$ and $\odot$, and one element $\overline{1}$ such that: $\otimes$ is associative and commutative, $\odot$ is associative, $\overline{1}$ is the neutral element of $\odot$, and $\odot$ and $\otimes$ are $\sqcup$-continuous. Notice, that the requirements above imply that $(D, \sqcup, \odot, \bot, \overline{1})$ is an idempotent closed semiring.

Intuitively, the abstract operations $\sqcup$, $\odot$, and $\otimes$ of $\mathcal{D}$ correspond to union, concatenation, and word parallel composition ($\|\!|$ or $\sqcup\!\sqcup$, depending on the adopted semantics) in the lattice $\mathcal{L}$. $\bot$ and $\overline{1}$ are the abstract objects corresponding to the empty language and to $\{\varepsilon\}$, respectively. Moreover, the top element $\top \in D$ and the meet operation $\sqcap$ correspond in the lattice $\mathcal{L}$ to Act^* and to language intersection, respectively.

We consider abstractions where the domain D is generated by $\bot$, $\overline{1}$ and an element v_a for each $a \in Act$. We always take $v_\tau = \overline{1}$. Intuitively, the element v_a corresponds to the language $\{a\}$ if $a \neq \tau$.

To define a Galois connection between the concrete and the abstract domains, we consider a mapping α that satisfies the following: $\alpha(\varepsilon) = \overline{1}$, and for every word languages L_1, L_2, we have: $\alpha(L_1 \cdot L_2) = \alpha(L_1) \odot \alpha(L_2)$, $\alpha(L_1 \cup L_2) = \alpha(L_1) \sqcup \alpha(L_2)$, and $\alpha(L_1 \|\!| L_2) = \alpha(L_1) \otimes \alpha(L_2)$ (or $\alpha(L_1 \sqcup\!\sqcup L_2) = \alpha(L_1) \otimes \alpha(L_2)$ if we are in the weak semantics case). It follows that

$$\alpha(L) = \bigsqcup_{a_1 \cdots a_n \in L} v_{a_1} \odot \cdots \odot v_{a_n}.$$

Furthermore, we define the concretization function γ by

$$\gamma(x) = \{a_1 \cdots a_n \in 2^{Act^*} \mid v_{a_1} \odot \cdots \odot v_{a_n} \sqsubseteq x\}.$$

It can be checked that (α,γ) is indeed a Galois connection between $\mathcal{L}$ and $\mathcal{D}$.

The fact that $\alpha(\emptyset) = \bot$ and $\gamma(\bot) = \emptyset$, implies that

$$\forall L_1, L_2. \alpha(L_1) \sqcap \alpha(L_2) = \bot \Rightarrow L_1 \cap L_2 = \emptyset.$$

This property is necessary for our approach: To solve the problems (2) and (3) we are interested in, it suffices to check, respectively, whether

$$\alpha\big(Paths_R(T,T')\big) \sqcap \alpha\big((Async \cup \{\tau\})^*\big) \stackrel{?}{=} \bot \tag{6}$$

or

$$\alpha\big(WPaths_R(T,T')\big) \sqcap \alpha\big((Async \cup \sum_{a \in Sync} a\,\overline{a})^*\big) \stackrel{?}{=} \bot \tag{7}$$

where $\alpha\big(Paths_R(T,T')\big)$ (resp. $\alpha\big(WPaths_R(T,T')\big)$) is the least solution of the *abstract* system of constraints:

$$f_i^{\alpha}(x_1,\dots,x_m) \sqsubseteq x_i, \quad 1 \leq i \leq m, \tag{8}$$

obtained from the "*concrete*" system (β_1)–(β_5), where $f_i^{\alpha}(x_1,\dots,x_m)$ is an expression obtained by substituting in $f_i(x_1,\dots,x_m)$ of (4) word concatenation with $\odot$, the operator $|||$ (resp. $\sqcup\!\sqcup$) with $\otimes$, and the operator $\cup$ with $\sqcup$.

5.2 Computing the abstractions

To be able to solve the system (8), we consider two types of abstractions. We give in the appendix instances of these types of abstractions, and show how they can be used in the analysis of a practical example.

Finite-chain abstractions: A *Finite-chain abstraction* is an abstraction such that the semilattice $(D,\sqcup)$ has no infi nite ascending chains. Particular cases of such abstractions are *finite abstractions* where the abstract domain D is fi nite. In this case, the iterative computation of the least fi xpoint of the system (8)always terminates.

Finite abstractions can be used for both strong and weak semantics of parallel composition to compute upper approximations of the sets $Paths_R(T,T')$ or $WPaths_R(T,T')$.

Commutative Kleene algebraic abstractions: We introduce now a particular class of abstractions which can be used in the weak semantics case, i.e., in order to abstract the set $WPaths_R(T,T')$.

We consider abstractions defi ned as above, but satisfying (i) $\odot = \otimes$, and (ii) $\odot$ is commutative. Intuitively, this means that both sequential word composition and the $\sqcup\!\sqcup$ operator are abstracted by $\odot$ (see remark below).

In this case, the structure $(D,\sqcup,\odot,\perp,\overline{1})$ is a *commutative* idempotent closed semiring. As usual, we defi ne $a^0 = \overline{1}$, $a^{n+1} = a \odot a^n$, and $a^{\star} = \bigsqcup_{n\geq 0} a^n$. Adding the $\star$-operation transforms the structure above into a *commutative Kleene algebra* $\mathcal{K} = (D,\sqcup,\odot,\star,\perp,\overline{1})$. Then, the system (8) can be solved using the algorithm of Hopkins and Kozen [HK99] for solving systems of polynomial constraints in commutative Kleene algebras (see also [BET03]).

Remark 2. Notice that to be able to use the framework of commutative Kleene algebras, we need to consider that $\odot$, i.e., the abstract sequential composition, is commutative. It can be seen that if sequential composition is considered as commutative, it coincides precisely with the shuffle operator $\sqcup\!\sqcup$. However, in Kleene algebras we cannot have an additional operator $\otimes$ in addition to $\odot$. So, the only case we can deal with is when this operator of parallel composition concides with $\odot$, which means that it should represent $\sqcup\!\sqcup$. This is the reason why this approach based on commutative Kleene algebras can only be applied in the case of the weak semantics.

6 Conclusion

We have presented a generic approach for the static analysis of concurrent programs with (unbounded) dynamic creation of threads and recursive procedure calls. We use a formal model for such systems called SPA where program confi gurations are defi ned as process terms in an adequate process algebra, and program actions are modeled by means of term rewrite rules. Then, our methodology allows to compute effectively abstractions of the set of computation paths of a program as the least solution of a system of constraints.

An interesting issue we plan to investigate in the future is the extension of our approach (based on a combination of automata-theoretic techniques and path language abstractions) to models with different/richer communication policies such as communication through shared variables and broadcast communication.

References

[BCR01] T. Ball, S. Chaki, and S. K. Rajamani. Parameterized verification of multithreaded software libraries. In *TACAS'01*. LNCS 2031, 2001.

[BEH95] A. Bouajjani, R. Echahed, and P. Habermehl. On the Verification Problem of Nonregular Properties for Nonregular Processes. In *LICS'95*. IEEE, 1995.

[BEM97] A. Bouajjani, J. Esparza, and O. Maler. Reachability Analysis of Pushdown Automata: Application to Model Checking. In *CONCUR'97*. LNCS 1243, 1997.

[BET03] A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *POPL'03*. ACM, 2003.

[BT03] Ahmed Bouajjani and Tayssir Touili. Reachability Analysis of Process Rewrite Systems. In *FSTTCS'03*. LNCS, 2003.

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

[CC77] P. Cousot and R. Cousot. Static Determination of Dynamic Properties of Recursive Procedures. In *IFIP Conf. on Formal Description of Programming Concepts*, 1977.

[CDG$^+$97] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: http://www.grappa.univ-lille3.fr/tata, 1997.

[CLR90] Th. Cormen, Ch. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[DBR02] G. Delzanno, L. Van Begin, and J.-F. Raskin. Toward the automated verification of multithreaded java programs. In *TACAS'02*. LNCS, 2002.

[DS91] E. Duesterwald and M.L. Soffa. Concurrency analysis in the presence of of procedures using a data-fbw framework. In *Proc. of the Symposium on Testing, Analysis, and Verifi cation, Victoria, Canada*, pages 36–48. ACM Press, 1991.

[EHRS00] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithm for model checking pushdown systems. In *CAV'00*, volume 1885 of *LNCS*, 2000.

[EK99] J. Esparza and J. Knoop. An automata-theoretic approach to interprocedural data-fbw analysis. In Wolfgang Thomas, editor, *FoSSaCS'99*, LNCS. Springer, 1999.

[EP00] J. Esparza and A. Podelski. Efficient algorithms for pre* and post* on interprocedural parallel fbw graphs. In *POPL'00*, pages 1–11. ACM Press, 2000.

[FQ03] Cormac Flanagan and Shaz Qadeer. Assume-Guarantee Model Checking. Technical report, Microsoft Research, January 2003.

[FRSB02] A. Finkel, J.-F. Raskin, M. Samuelides, and L. Van Begin. Monotonic extensions of petri nets: Forward and backward search revisited. In *INFINITY'2002*, 2002.

[FWW97] A. Finkel, B. Willems, and P. Wolper. A Direct Symbolic Approach to Model Checking Pushdown Systems. *ENTCS*, 9, 1997.

[HK99] M.W. Hopkins and D.C. Kozen. Parikh's Theorem in Commutative Kleene Algebra. In *Proc. IEEE Conf. Logic in Computer Science (LICS'99)*. IEEE, 1999.

[LS02] D. Lugiez and P. Schnoebelen. The Regular Viewpoint on PA-processes. In *TCS*, volume 274(1-2), 2002.

[MO02] Markus Müller-Olm. Variations on Constants. Habilitation Thesis. University of Dortmund, 2002.

[QRR04] S. Qadeer, S.K. Rajamani, and J. Rehof. Procedure Summaries for Model Checking Multithreaded Software. In *POPL'04*, 2004.

[Ram00] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM TOPLAS*, 22:416–430, 2000.

[Rin01] M. Rinard. Analysis of multithreaded programs. In Patrick Cousot, editor, *SAS'01*, volume 2126 of *LNCS*, 2001.

[SS00] H. Seidl and B. Steffen. Constraint-based inter-procedural analysis of parallel programs. *Nordic Journal of Computing*, 7(4):375–400, 2000.

A Instances of our Abstraction Framework

Finite-chain abstractions: We give hereafter examples of finite-chain abstractions. We illustrate on one of these abstractions the fact that they can be defined as instances of our generic framework. The expression of the two others in the framework is not difficult.

Forbidden and required sets: The abstract object is a pair $[F,R]$, where $F,R \subseteq Lab$. F, the *forbidden* set, contains the labels a that do not occur in any sequence of $Paths_R(T,T')$. R, the *required* set, contains the labels a that appear in all sequences of $Paths_R(T,T')$. $[F,R]$ represents the language of all sequences containing no occurrence of letters in F and at least one occurrence of each letter in R.

This abstraction is defined in our framework as follows:

- $D = \{\bot\} \cup \{[F,R] \in 2^{Lab} \times 2^{Lab} \mid F \cap R = \emptyset\}$, i.e., the set of all pairs of sets of actions generated by the elements v_a for each $a \in Lab$, where $v_a(a) = [Lab \setminus \{a\}, \{a\}]$, augmented with a special element $\bot$,
- $[F_1,R_1] \sqsubseteq [F_2,R_2]$ iff $F_1 \supseteq F_2$ and $R_1 \supseteq R_2$,
- $[F_1,R_1] \sqcup [F_2,R_2] = [F_1 \cap F_2, R_1 \cap R_2]$,
- $[F_1,R_1] \odot [F_2,R_2] = [F_1 \cap F_2, R_1 \cup R_2]$,
- $[F_1,R_1] \otimes [F_2,R_2] =$
$$[F_1 \cap F_2, \big((R_1 \cup R_2) \cap Async\big) \cup \{a \in R_1 \mid \overline{a} \in F_2\} \cup \{a \in R_2 \mid \overline{a} \in F_1\}]$$
- $\overline{1} = [Lab, \emptyset]$.

The abstract lattice is obtained by taking $\top = [\emptyset,\emptyset]$, and defining $\sqcap$ by:

- $[F_1,R_1] \sqcap [F_2,R_2] = [F_1 \cup F_2, R_1 \cup R_2]$ if $(F_1 \cap R_2) \cup (F_2 \cap R_1) = \emptyset$,
- $[F_1,R_1] \sqcap [F_2,R_2] = \bot$ otherwise,
- $\bot \sqcap x = x \sqcap \bot = \bot$.

Label bitvectors: The abstract object is now a set S of bitvectors $Lab \to \mathbb{B}$. A bitvector b belongs to S if there is a sequence in $Paths_R(T,T')$ such that $b(a) = 1$ if a occurs in the sequence and $b(a) = 0$ otherwise.

First occurrence ordering: This is the most precise finite-chain abstraction we consider here. The abstract object is a set S of words w such that for every $a \in Lab$, $|w|_a \leq 1$ ($|w|_a$ denotes the number of occurrences of the letter a in w). A word $w = a_1 \cdots a_n$ belongs to S if there is a path in $Paths_R(T,T')$ such that the set of letters occurring in this path is precisely $\{a_1,\ldots,a_n\}$, and moreover, the first occurrences of these letters in the path occur in the ordering defined by w (i.e., for every $i < j$, a_i occurs for the first time before the first occurrence of a_j).

Remark 3. Finer abstractions can be obtained by considering the k first occurrences, for a fixed natural number k.

Commutative abstractions: Examples of commutative abstractions are the forbidden/required sets and the label bitvector sets abstractions defined above.

An interesting (more precise) abstraction in this class is the Parikh image abstraction. Intuitively, in this abstraction a path is abstracted to its Parikh image, a vector that counts for each letter the number of times it occurs in the word. Formally, abstract elements are semilinear sets of integer vectors in $[Lab \to \mathbb{N}]$. We recall that semilinear sets are finite unions of sets of the form $\{\mathbf{u} + k_1\mathbf{v}_1 + \ldots k_n\mathbf{v}_n \mid k_1 \ldots k_n \in \mathbb{N}\}$, where $\mathbf{u}, \mathbf{v}_1, \ldots, \mathbf{v}_n \in [Lab \to \mathbb{N}]$ ($\mathbf{u}$ is the basis, and the $\mathbf{v}_i$'s are the periods). It is easy to define this abstraction in our framework (see [BET03]).

Remark 4. Notice that none of the classes of finite-chain and commutative abstractions is included in the other. Indeed, the first-occurrence abstraction is not commutative whereas the Parikh image abstraction is not finite-chain. We can now explain why we have introduced two different formulations of the reachability problem, namely the expressions (2) and (3). Formulation (2) does not lead to a generic procedure for computing arbitrary commutative abstractions: the reason is that the $|||$ operator corresponding to the *Paths* semantics does not lead to a commutative Kleene algebra, and the algorithms of [HK99] only work for commutative Kleene algebras. On the other hand, formulation (3) does allow to consider arbitrary commutative abstractions, like the Parikh images, however only for well-formed SPA systems.

B Example: Parallel minimum computation

B.1 The application and its SPA model

We describe an SPA model of a system that accepts a stream of inputs $x_1 \dots x_n$ (in some data domain), and returns $Min_{i=1}^{n} f(x_i)$, where f is some function (defined for all values in the domain of inputs), and *Min* computes the minimum value w.r.t. a total ordering on the data domain of inputs. The system computes the *Min* value of the input stream in parallel. It dynamically generates threads for this parallel computation.

Initially, the system consists of an input-output interface *IO*, an input stack to store the inputs, a result stack to store intermediate results, and a process, the distributor, that distributes inputs and results to new processes. Here by a stack we mean a recursive procedure which accepts a value and calls itself, or delivers the value and terminates. The parallel composition of these processes is defined by the process term:

$$IO \parallel Dist \parallel Input\_stack \parallel Result\_stack$$

The input and result stacks have very similar behaviours:

$$\begin{aligned}
Input\_stack &\xrightarrow{i1?x} I[x] \cdot Input\_stack \\
I[x] &\xrightarrow{i1?y} I[y] \cdot I[x] \\
I[x] &\xrightarrow{i2!x} 0
\end{aligned}$$

$$\begin{aligned}
Result\_stack &\xrightarrow{r1?x} R[x] \cdot Result\_stack \\
R[x] &\xrightarrow{r1?y} R[y] \cdot R[x] \\
R[x] &\xrightarrow{r2!x} 0
\end{aligned}$$

Here, we think of an expression $I[x]$ as a process constant. Since an SPA system must have finitely many constants, we are restricting ourselves to finitely many different inputs. Notice, however, that this is not a problem once one assumes that f and *Min* are computed correctly: under this assumption, it is easy to see that if the system works correctly for all streams with two different inputs 0 and 1, then it works correctly for all streams of inputs. The justification of this fact is based on the so-called 0-1-principle for sorting networks (see e.g., [CLR90]).

The distributor creates for each input x a new process *Comp_f* that computes $f(x)$. Since for n inputs, $n-1$ *Min* operations are required, the distributor also creates a *Comp_Min* process for each input, but starting from the second one.

$$\begin{aligned}
Dist &\xrightarrow{i2?x} Dist_1 \parallel Comp\_f[x] \\
Dist_1 &\xrightarrow{i2?x} Dist_1 \parallel Comp\_f[x] \parallel Comp\_Min
\end{aligned}$$

The $Comp\_f[x]$ process just returns $f(x)$:

$$Comp\_f[x] \xrightarrow{r1!f(x)} 0$$

The *Comp_Min* processes get their two inputs x, y from the result stack, and return $Min(x, y)$. Before computing, they send a *report* signal to indicate that they have already received their two inputs. The purpose of this will be clear in a moment.

$$\begin{array}{rcl} Comp\_Min & \xrightarrow{r2?x} & Comp\_Min[x] \\ Comp\_Min[x] & \xrightarrow{r2?y} & Comp\_Min[x,y] \\ Comp\_Min[x,y] & \xrightarrow{report!} & Comp\_Min_1[x,y] \\ Comp\_Min_1[x,y] & \xrightarrow{r1!Min(x,y)} & 0 \end{array}$$

The *IO* process is in charge of sending inputs to the input stack, and collecting the final result from the result stack. However, how does it know that the result has been computed? For this, it counts the number of inputs minus the number of *Min* operations that the system has already initiated. When the number of inputs exceeds the number of operations by one, it knows that only the final result is in the result stack, and it picks it up. Let us give the rules defining this process. The first two rules just read the first input and put it in the input stack

$$\begin{array}{rcl} IO & \xrightarrow{in\_x} & IO[x] \\ IO[x] & \xrightarrow{i1!x} & IO_1 \end{array}$$

The next two rules do the same, but the process now also creates a *Get_Result* process at the bottom of the stack.

$$\begin{array}{rcl} IO_1 & \xrightarrow{in\_x} & IO_1[x] \\ IO_1[x] & \xrightarrow{i1!x} & IO_2 \cdot Get\_Result \end{array}$$

The next two rules add one IO_2 process to the stack for each new input, and remove one for each *Min* computation:

$$\begin{array}{rcl} IO_2 & \xrightarrow{in\_x} & IO_2[x] \\ IO_2[x] & \xrightarrow{i1!x} & IO_2 \cdot IO_2 \\ IO_2 & \xrightarrow{report?} & 0 \end{array}$$

Finally, the *Get_Result* process just gets a value from the output stack, sends it, and terminates.

$$\begin{array}{rcl} Get\_Result & \xrightarrow{r2?x} & Output\_Result[x] \\ Output\_Result[x] & \xrightarrow{out\_x} & Terminated \end{array}$$

Now, the behavior of the system is defined by the set of computation paths starting from the restricted process term:

$$(IO \parallel Dist \parallel Input\_stack \parallel Result\_stack) \backslash Sync$$

where *Sync* is the set of all synchronization actions appearing in the rules above. It can be checked that the SPA system defined above is well formed.

B.2 Analysis of the example

We now show how our analysis technique can be applied to derive some useful information about the system described above. For this, we take T as the singleton set containing the SPA term $IO \parallel Dist \parallel Input\_stack \parallel Result\_stack$. Notice that we have $Async = \{in\_0, in\_1, out\_0, out\_1\}$.

Correctness: First of all, we wish to know if the system computes the right output. For that, we take for T' the set of all terms containing *Terminated*, and we use the label bitvector abstraction: We consider that a wrong output corresponds to a bitvector indicating that *in*_0 is present in the path (meaning that at least one of the inputs was 0) and that *out*_1 is also present in the same path (meaning that the computed minimum value is 1). It is not diffi cult to show that for this abstaction the question (6) has a positive answer.

Deadlock freeness: We would also like to check that the system cannot have the following kind of deadlock: the input and result stacks are empty, all the *Comp_f* processes have terminated, and there is at least one $Comp\_Min[x]$ around, which has no possibility to terminate. For this, we consider all the processes exhibiting this situation. This is the regular set of process terms T' given by:

$$(IO_2 \cdot Get\_Result + Get\_Result) \| Input\_stack \| Result\_stack \| Dist_1 (\| Comp\_Min[x])^+$$

(We abuse notation here by giving a "regular expression"-like description of this set of terms. The interpretation, however, should be straightforward.)

We use now the Parikh image abstraction: the paths leading to T' must perform some extra non synchronizing transitions $i2?x$ creating more *Comp_Min* processes than necessary. These paths are discarded by taking the intersection with the Parikh image of $(Async \cup \sum_{a \in Sync} a\,\bar{a})^*$, as indicated in (7), which is precisely the set of all paths with the same number of a and $\bar{a}$ for each synchronization action.

It can be seen that to check the deadlock property, we really need to count, and all the other abstractions described above are too imprecise.

Remark 5. These two examples show the relevance of (1) having a generic framework where different kinds of abstractions can be defi ned and computed, and of (2) having a general procedure for reachability analysis allowing to defi ne the source and target sets of confi gurations. Indeed, to deal with the two properties above, we consider different abstractions as well as two different regular sets T' of target confi gurations.

C Proof of Theorem 1

Theorem 1. The reachability problem of SPAs is undecidable.

Proof:

We encode the halting problem of a two-counter machine. The reduction is similar to the one presented in [BEH95] used to prove the undecidability of Model checking LTL for PA systems.

Let $\mathcal{M}$ be a two-counter machine with m instructions. Let

– $Var = \{X, Y, X_1^1, X_2^1, X_1^2, X_2^2\} \cup \{X_i^c \mid 1 \leq i \leq m\}$,
– $Sync = \{i_1, i_2, i_1, i_2, d_1, d_2, d_1, d_2, z_1, z_2, \bar{z}_1, \bar{z}_2\}$,
– $Async = \{halt, a\}$.

Let R be the SPA having the following rules, where $j \in \{1,2\}$ and $k \in \{1,\ldots,m\}$:

1. $X \overset{a}{\hookrightarrow} X_1^1 || X_1^2 || X_1^c$,
2. $X_1^j \overset{z_j}{\hookrightarrow} X_1^j$,
3. $X_1^j \overset{i_j}{\hookrightarrow} X_2^j \cdot X_1^j$,
4. $X_2^j \overset{i_j}{\hookrightarrow} X_2^j \cdot X_2^j$,
5. $X_2^j \overset{d_j}{\hookrightarrow} 0$,

6. $X_k^c \overset{\overline{i_j}}{\hookrightarrow} X_h^c$, if $(s_k : c_j := c_j + 1;\ \text{goto}\ s_h)$ is an instruction,
7. $X_k^c \overset{\overline{z_j}}{\hookrightarrow} X_{h_1}^c$ and $X_k^c \overset{\overline{d_j}}{\hookrightarrow} X_{h_2}^c$ if the following instruction exists:

$$(s_k :\ \text{if}\ c_j = 0\ \text{then go to}\ s_{h_1}\ \text{else}\ c_j := c_j - 1;\ \text{goto}\ s_{h_2})$$

8. $X_m \overset{halt}{\hookrightarrow} Y$.

Intuitively, the process variable X_1^j represents the value 0 of the counter j, and the term $X_2^j \cdots X_2^j \cdot X_1^j$ with n X_2^j's represents the value n of the counter j. The second rules simulate the test of equality of the counter c_j to 0, the third and the fourth rules to the incrementation of c_j, whereas the fifth rules simulate the decrementation of c_j. The three last rules simulate the finite control of the machine. The last rule simulates the halt of the machine. The actions i_j and $\overline{i_j}$ represent the incrementation of c_j. The synchronization between these two co-actions imposes that the counters can be incremented only if the controller allows it. The same intuition holds for the other actions where d_j and $\overline{d_j}$ represent the decrementation of c_j, and z_j and $\overline{z_j}$ the test to zero of c_j.

Then it is clear that $\mathcal{M}$ halts iff

$$Post^*_{\sim}(X \backslash Sync) \cap L \backslash Sync \neq \emptyset,$$

where L is the set of terms of the form $t_1||t_2||Y$, where t_j is a term of the form $X_2^j \cdots X_2^j \cdot X_1^j$ that represents the value of the counter j.

□

D Proof of Lemma 3

Lemma 3.
For every $s_1, s_2, t_1, t_2 \in \mathcal{T}$, and every $w \in Act^*$, $s_1 \| s_2 \in Post^*[w](t_1 \| t_2)$ iff $\exists w_1, w_2 \in Act^*$ such that $w = w_1 ||| w_2$ and, $s_1 \in Post^*[w_1](t_1)$, and $s_2 \in Post^*[w_2](t_2)$.

Proof: Let us consider the direction $\Rightarrow$ first. Let $u_1, u_2, v_1, v_2 \in \mathcal{T}$, and $w \in Act^*$ s.t.

$$||(u_1, u_2) \in Post^*_{=}[w]\big(||(v_1, v_2)\big).$$

We proceed by induction on $|w|$:

– $|w| = 0$. Then $w = \varepsilon$, and the property holds with $w_1 = w_2 = \varepsilon, u_1 = v_1$, and $u_2 = v_2$.
– $|w| > 0$, i.e., $w = bw'$ for $b \in Act$. Let then $u_1', u_2' \in \mathcal{T}$ s.t. $||(u_1', u_2') \in Post^*_{=}[b]\big(||(v_1, v_2)\big)$, and $||(u_1, u_2) \in Post^*_{=}[w']\big(||(u_1', u_2')\big)$. It follows by induction that it exist two sequences w_1' and w_2' s.t. $w' \in w_1' ||| w_2'$, $u_1 \in Post^*_{=}[w_1'](u_1')$, and $u_2 \in Post^*_{=}[w_2'](u_2')$. There are two cases depending on the nature of b:
  1. $b \neq \tau$. Let then $v_1 = u_1'$ and $v_2 \xrightarrow{b}_{=} u_2'$, let $v_2 = u_2'$ and $v_1 \xrightarrow{b}_{=} u_1'$. The two cases are symmetrical. Suppose for example the first case. Let then $w_1 = w_1'$ and $w_2 = bw_2'$. It is easy to see that $w \in w_1 ||| w_2$, $u_1 \in Post^*_{=}[w_1](v_1)$, and $u_2 \in Post^*_{=}[w_2](v_2)$.
  2. $b = \tau$. Let then $a \in Sync$ s.t. $v_1 \xrightarrow{a}_{=} u_1'$ and $v_2 \xrightarrow{\overline{a}}_{=} u_2'$. Consider $w_1 = aw_1'$ and $w_2 = \overline{a}w_2'$. It is easy to see that $w \in w_1 ||| w_2$, $u_1 \in Post^*_{=}[w_1](v_1)$, and $u_2 \in Post^*_{=}[w_2](v_2)$.

We show now the other direction. Let then $w, w_1, w_2 \in Act^*$ and $u_1, u_2, v_1, v_2 \in \mathcal{T}$ s.t. $w \in w_1 ||| w_2$, $u_1 \in Post^*_{=}[w_1](v_1)$, and $u_2 \in Post^*_{=}[w_2](v_2)$. We show by induction on $|w_1| + |w_2|$ that $||(u_1, u_2) \in Post^*_{=}[w]\big(||(v_1, v_2)\big)$:

– $|w_1| + |w_2| = 0$. Then $w = w_1 = w_2 = \varepsilon$, and the property holds.
– $|w_1| + |w_2| > 0$. Let $w \in w_1 ||| w_2$, we will show that

$$||(u_1, u_2) \in Post^*_{=}[w]\big(||(v_1, v_2)\big)$$

Let then w'_1, w'_2, b, and b' s.t. $w_1 = bw'_1$ and $w_2 = b'w'_2$. The case where $w_1 = \varepsilon$ (or $w_2 = \varepsilon$) is direct since in this case $w_1 ||| w_2 = w_2$ ($w_1 ||| w_2 = w_1$). Let u'_1 and u'_2 s.t. $v_1 \xrightarrow{b}_{=} u'_1$, $v_2 \xrightarrow{b'}_{=} u'_2$, $u_1 \in Post^*_{=}[w'_1](u'_1)$, and $u_2 \in Post^*_{=}[w'_2](u'_2)$. There are two cases depending on the natures of b and b':

1. $b' \neq \bar{b}$. In this case we have $w_1 ||| w_2 = b(w'_1 ||| w_2) + b'(w_1 ||| w'_2)$. Let then $w' \in w'_1 ||| w_2$ s.t. $w = bw'$ (the case where $w = b'w'$ for a w' in $w_1 ||| w'_2$ is symmetrical). Since $u_1 \in Post^*_{=}[w'_1](u'_1)$, $u_2 \in Post^*_{=}[w_2](u_2)$, and $|w'_1| + |w_2| < |w_1| + |w_2|$, we obtain by induction that $||(u_1, u_2) \in Post^*_{=}[w']\big(||(u'_1, v_2)\big)$, which infers that $||(u_1, u_2) \in Post^*_{=}[w]\big(||(v_1, v_2)\big)$.
2. $b' = \bar{b}$. In this case, we have $w_1 ||| w_2 = b(w'_1 ||| w_2) + \bar{b}(w_1 ||| w'_2) + \tau(w'_1 ||| w'_2)$ Let then $w' \in w'_1 ||| w'_2$ s.t. $w = \tau w'$ (the other cases are handled as previously). By induction, it follows that

$$||(u_1, u_2) \in Post^*_{=}[w']\big(||(u'_1, u'_2)\big)$$

Therefore, we obtain that $||(u_1, u_2) \in Post^*_{=}[w]\big(||(v_1, v_2)\big)$ by applying the rules θ_5 since $v_1 \xrightarrow{b}_{=} u'_1$ and $v_2 \xrightarrow{\bar{b}}_{=} u'_2$.

□

E Proof of Theorem 2

Theorem 2.
For every $q \in Q$ and every $s \in Q'$, we have $L(q, s) = \lambda(q, s)$.

Proof: We show that for every $q \in Q$ and every $s \in Q'$,

$$Post^*_{=}[w](L_q) \cap L_s \neq \emptyset \Leftrightarrow w \in L(q, s).$$

We start with the implication $\Rightarrow$. Let $q \in Q$, $s \in Q'$, $w \in Act^*$, and $u \in Post^*_{=}[w](L_q) \cap L_s$. Let then $v \in L_q$ s.t. $u \in Post^*_{=}[w](v)$. We proceed by induction on $|w|$.

– $|w| = 0$, then $w = \varepsilon$, and $u \in L_q \cap L_s$, which means that $\varepsilon \in L(q, s)$ (from (β_1)).
– $|w| > 0$. There are two cases:
  1. The root of v has been rewritten. Let then $X \overset{a}{\hookrightarrow} t \in R$, $w_1, w_2 \in Act^*$ s.t.

$$X \in Post^*_{=}[w_1](v), u \in Post^*_{=}[w_2](t),$$

and $w = w_1 a w_2$. Since $|w_1| < |w|$ and $|w_2| < |w|$, the induction hypothesis implies that $w_1 \in L(q, q_X)$, and $w_2 \in L(q_t, s)$. Therefore, we obtain from (β_5) that

$$w = w_1 \cdot a \cdot w_2 \in L(q, q_X) \cdot a \cdot L(q_t, s) \subseteq L(q, s)$$

  2. The root of v was not rewritten. We proceed by structural induction on u:
     • $v = \cdot(v_1, v_2)$ and $u = \cdot(u_1, u_2)$ s.t. $v_1 \in L_{q_1}$, $v_2 \in L_{q_2}$, $u_1 \in L_{s_1}$, $u_2 \in L_{s_2}$, where $\cdot(q_1, q_2) \to q$ is a rule of Δ and $\cdot(s_1, s_2) \to s$ is a rule of Δ' s.t.:

    * u_1 is equivalent to 0, i.e., $u_1 \in L_{s_1^\perp}$. Let then w_1, w_2 s.t. $w = w_1 \cdot w_2$, $u_1 \in Post_{=}^{*}[w_1](v_1)$, and $u_2 \in Post^{*}[w_2](v_2)$. By structural induction, we have $w_1 \in L(q_1, s_1^\perp)$ and $w_2 \in L(q_2, s_2)$. We obtain then by (β_3) that

$$w = w_1 \cdot w_2 \in L(q_1, s_1^\perp) \cdot L(q_2, s_2) \subseteq L(q, s).$$

    * u_1 is not equivalent to 0, in this case $u_2 = v_2 \in L_{s_2} \cap L_{q_2}$ and $u_1 \in Post_{=}^{*}[w](v_1)$. By structural induction, we obtain $w \in L(q_1, s_1)$, and by (β_3), we obtain that

$$w \in L(q_1, s_1) \subseteq L(q, s).$$

- $v = ||(v_1, v_2)$ and $u = ||(u_1, u_2)$. Lemma 3 infers that there exists w_1, w_2 s.t. $w \in w_1 ||| w_2$, $u_1 \in Post_{=}^{*}[w_1](v_1)$, and $u_2 \in Post_{=}^{*}[w_2](v_2)$. Let $q_1, q_2 \in \mathcal{Q}$ s.t. $||(q_1, q_2) \to q$ is a rule of Δ, $v_1 \in L_{q_1}$, and $v_2 \in L_{q_2}$. Moreover, let $s_1, s_2 \in \mathcal{Q}'$ s.t. $||(s_1, s_2) \to s$ is a rule of Δ' s.t. $u_1 \in L_{s_1}$ and $u_2 \in L_{s_2}$. Then, by structural induction we infer that $w_1 \in L(q_1, s_1)$ and $w_2 \in L(q_2, s_2)$, and thanks to (β_4), it follows that

$$w \in w_1 ||| w_2 \subseteq L(q_1, s_1) ||| L(q_2, s_2) \subseteq L(q, s).$$

Consider the other direction. Let $w \in L(q, s)$, we show that $Post_{=}^{*}[w](L_q) \cap L_s \neq \emptyset$. Since the labels $L(q, s)$ are built using a saturation procedure, we consider the seqences $\big(L_i(q, s)\big)_{0 \leq i \leq n}$ where $L_0(q, s) = \emptyset$, $L_n(q, s) = L(q, s)$, and $L_i(q, s)$ is the label obtained after the ith iteration. We prove by induction on i that if $w \in L_i(q, s)$, then $Post_{=}^{*}[w](L_q) \cap L_s \neq \emptyset$. The case where $i = 0$ is direct. The same for the case where $i = 1$, since in this case, it's the rule β_1 which is applied. Let then $i > 1$. Let $w \in L_i(q, s)$, then :

– Either there exists (q', s') s.t. $w \in L_{i-1}(q', s')$, and $L_{i-1}(q', s') \subseteq L_i(q, s)$ (rules β_2). In this case, we have necessarily $q' \to q \in \Delta$ and $s' \to s \in \Delta'$, and hence $L_{q'} \subseteq L_q$, and $L_{s'} \subseteq L_s$. By induction, we have

$$Post_{=}^{*}[w](L_{q'}) \cap L_{s'} \neq \emptyset.$$

It follows that $Post_{=}^{*}[w](L_q) \cap L_s \neq \emptyset$ since $L_{q'} \subseteq L_q$ and $L_{s'} \subseteq L_s$.

– Either there exist w_1, w_2 s.t. $w = w_1 w_2$, and there exist q_1, s_1, q_2, s_2 s.t. $\cdot(q_1, q_2) \to q \in \Delta$, $\cdot(s_1, s_2) \to s \in \Delta'$, $w_1 \in L_{i-1}(q_1, s_1^\perp)$, $w_2 \in L_{i-1}(q_2, s_2)$, $w \in L_i(q, s)$ (rules β_3). By induction, we obtain that

$$Post_{=}^{*}[w_1](L_{q_1}) \cap L_{s_1^\perp} \neq \emptyset$$

and

$$Post_{=}^{*}[w_2](L_{q_2}) \cap L_{s_2} \neq \emptyset.$$

Let then u_1 and u_2 s.t.

$$u_1 \in Post_{=}^{*}[w_1](L_{q_1}) \cap L_{s_1^\perp}$$

and

$$u_2 \in Post_{=}^{*}[w_2](L_{q_2}) \cap L_{s_2}.$$

It is then clear that (since $u_1 \sim_0 0$)

$$\cdot(u_1, u_2) \in Post_{=}^{*}[w](L_q) \cap L_s$$

– Either there exist q_1, s_1, q_2, s_2 s.t. $\cdot(q_1, q_2) \to q \in \Delta$, $\cdot(s_1, s_2) \to s \in \Delta'$, $w \in L_{i-1}(q_1, s_1)$, and $L_{q_2} \cap L_{s_2} \neq \emptyset$ (rules β_3). Let then $u_2 \in L_{q_2} \cap L_{s_2}$. Since $w \in L_{i-1}(q_1, s_1)$, it follows by induction that there exists $u_1 \in Post_{=}^{*}[w](L_{q_1}) \cap L_{s_1}$. It is then clear that

$$\cdot(u_1, u_2) \in Post_{=}^{*}[w](L_q) \cap L_s.$$

– Either there exist w_1, w_2 s.t. $w \in w_1 \| w_2$, and there exist q_1, s_1, q_2, s_2 s.t. $||(q_1, q_2) \to q \in \Delta$, $||(s_1, s_2) \to s \in \Delta'$, $w_1 \in L_{i-1}(q_1, s_1)$, and $w_2 \in L_{i-1}(q_2, s_2)$ (rules β_4). By induction, we obtain that
$$Post^*_{=}[w_1](L_{q_1}) \cap L_{s_1} \neq \emptyset$$
and
$$Post^*_{=}[w_2](L_{q_2}) \cap L_{s_2} \neq \emptyset.$$
Let then u_1 and u_2 s.t. $u_1 \in Post^*_{=}[w_1](L_{q_1}) \cap L_{s_1}$ and $u_2 \in Post^*_{=}[w_2](L_{q_2}) \cap L_{s_2}$. We obtain from Lemma 3 that
$$||(u_1, u_2) \in Post^*_{=}[w](L_q) \cap L_s$$
– Either there exist a rule $X \overset{a}{\hookrightarrow} t \in R$, w_1, w_2 s.t. $w = w_1 a w_2$, $w_1 \in L_{i-1}(q, q_X)$, $w_2 \in L_{i-1}(q_t, s)$ (rules β_5). By induction, we obtain that
$$X \in Post^*_{=}[w_1](L_q)$$
and there exists $u \in L_s$ s.t.
$$u \in Post^*_{=}[w_2](t).$$
It is then clear that
$$u \in Post^*_{=}[w_1 a w_2](L_q).$$
□