Reachability Analysis of Synchronized PA Systems

Ahmed Bouajjani

LIAFA, *University of Paris 7, 2 place Jussieu, 75251 Paris cedex 5, France*
abou@liafa.jussieu.fr

Javier Esparza

Institute for Formal Methods in Computer Science,
University of Stuttgart, Universitätstr. 38, 70569 Stuttgart, Germany
esparza@informatik.uni-stuttgart.de

Tayssir Touili

LIAFA, *University of Paris 7, 2 place Jussieu, 75251 Paris cedex 5, France*
touili@liafa.jussieu.fr

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 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.

Key words: Multithreaded programs with procedure calls, process algebra, program analysis, verification.

1 Introduction

Analyzing and verifying multithreaded programs is nowadays one of the most important problems in program analysis and computer-aided verifi cation. This problem is especially challenging in the case where the programming language allows

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

(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 [20]). Therefore, any analysis or verification algorithm for such programs must consider upper-approximations of the set of possible computation paths.

In a previous work [4], 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 [12] for modeling and analyzing parallel programs. In [12], 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 [4] 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 [7]) for computing (in a generic way) abstractions of the least solution of this system of constraints. In the full paper, we give examples of abstractions 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 (arbitrary length) streams of inputs.

Related work:

There are several works on static analysis of concurrent programs (see [21] for a survey).

In [2,8], 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 [13] using Petri nets with transfer transitions.

The automata approach for program analysis has been used in [11,10] for programs with procedures (without concurrency). These works are based on computing reachable configurations in pushdown automata [3,14]. This approach has been extended in [17,12] to parallel programs with dynamic creation of processes, but without synchronization, using as models process rewrite systems called PA processes. In [5], we extend this approach to a larger class of processes allowing return values of procedures.

In [4], 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 [3,10]. A different approach for analyzing parallel programs with procedures using path language abstractions is presented in [15].

In [22,18], 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. (2) The work in [22] (like the one in [12]) 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 [18] 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 [9]. 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 [19], 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 a process algebra-based model for multithreaded programs with recursive calls which is an extension of PA [1] 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 $\overline{a}$ in *Sync* such that $\overline{\overline{a}} = a$. 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 "$\cdot$" (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, let $T \backslash Sync$ denote $\{t \backslash Sync \mid t \in T\}$.

Definition 2.1 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

2.2.1 Structural equivalences on terms:

Terms are considered modulo the equivalence $\sim$ which correponds to the algebraic properties: neutrality of the null process "0" w.r.t. "$\cdot$" and "$\|$", the associativity of "$\cdot$" and "$\|$", and the commutativity of "$\|$". We also need to consider the equivalence relation $\sim_0$ on $\mathcal{T}$ corresponding to the properties of 0 (neutrality w.r.t. "$\cdot$" and "$\|$").

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 $[L']_\equiv = L$.

2.2.2 Transition relations and computations:

An SPA R induces a transition relation $\overset{a}{\rightarrow}$ over $\mathcal{T} \cup \mathcal{T}_r$ defined by:

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

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

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

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

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 $\stackrel{a}{\Rightarrow}_{\equiv}$ induced by the equivalences $\equiv$ defined in the obvious way, and we define for every term $t \in \mathcal{T}$, $WPost^*_{\equiv}[w](t) = \{t' \in \mathcal{T} \mid t \stackrel{w}{\Rightarrow}_{\equiv} t'\}$ 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

2.3.1 From programs to SPA systems:

Programs represented by parallel flow graph systems (see e.g., [12,22,18]) 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 $X \stackrel{a}{\hookrightarrow} X_1 \cdot X_2$ correspond to procedure calls, and rules of the form $X \stackrel{a}{\hookrightarrow} X_1 \| X_2$ correspond to dynamic creation of parallel processes. Complementary actions a, $\bar{a}$ 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 $\{a, \bar{a} \mid \text{a is a communication channel}\}$.

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., $Paths_R(T \backslash Sync, \mathcal{T}_r)$.

2.3.2 Well formed systems:

A 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 $X \xrightarrow{a} \nu$, $\nu \xrightarrow{op} X_1$, and $\nu \xrightarrow{op} X_2$, where ν 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 $v_1 \xrightarrow{\bar{a}} v_2$ 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.

Lemma 2.2 *If R is a well formed SPA, then for every terms t and t' in $\mathcal{T}$, we have: $t \xrightarrow{\tau}_{\equiv} t'$ iff $\exists a \in Sync, t \overset{a\bar{a}}{\Rightarrow}_{\equiv} t'$*

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:

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

However, it is not difficult to prove that the reachability problem of SPA systems is undecidable (using a reduction of the halting problem of 2-counter machines). Therefore, 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 2.2 implies that (2) is equivalent to checking whether

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

Since the reachability problem of SPA is undecidable, 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. $A(T,T') \cap (Async \cup \sum_{a \in Sync} a\ \overline{a})^*$) 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 3.1 We will see later that the two formulations (2) and (3) above lead to complementary analysis approaches: they allow to consider different abstractions with uncomparable precisions (see Remark 5.1).

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 3.2 *$\forall T,T' \subseteq \mathcal{T}$, if T' is $\sim$-compatible, 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

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 4.1 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 $f(q_1,q_2) \to_\delta q$ 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. From [6], the class of regular process tree languages is closed under boolean operations. Moreover, the emptiness problem of process tree automata is decidable in linear time.

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. Let us start by the case of sequential composition.

Lemma 4.2 *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 $w = w_1w_2$ 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:

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

Lemma 4.3 *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 $s^{\perp\perp} = s^\perp$) [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 $\sqcup\!\sqcup$.

[1] We show in the full paper how to transform A' in order to satisfy this assumption.

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 $Q^R = \{q_t \mid t \text{ is a subterm of a term appearing in some rule of } R\}$ and let us define δ^R to be the set of rules: (1) $X \to q_X$ if $q_X \in Q^R$, for every $X \in Var$, (2) $\|(q_{t_1}, q_{t_2}) \to q_t$ if $t = \|(t_1, t_2)$ and $q_t \in Q^R$, and (3) $\cdot(q_{t_1}, q_{t_2}) \to q_t$ 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 $Paths_R(T,T')$ since, by Proposition 3.2, this set is the union of all $\lambda(q,s)$ such that $q \in F$ and $s \in F'$.

4.3.1 A Set of Constraints:

We define hereafter a set of constraints on path languages and prove that it characterizes precisely the sets $\lambda(q,s)$. Let us consider a set of variables (representing sets of paths) defined as follows: For every state $q \in \mathcal{Q}$ and every state $s \in \mathcal{Q}'$, we define a variable $V(q,s)$. Then, we consider 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

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

(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

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

4.3.2 Correctness:

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

Proposition 4.4 *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 $q \in Q$ and $s \in Q'$. Then, the system (β_1)–(β_5) is a set of inclusion constraints

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

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

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

The least solution of (4) is the least pre-fixpoint 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-fixpoint of F exists and is equal to its least fixpoint, and by Kleene's theorem this fixpoint is equal to:

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

Theorem 4.5 *Let* $\big(L(q,s)\big)_{q \in Q, s \in Q'}$ *be the least solution of the system* (β_1)–(β_5). *Then, for every* $q \in Q$ *and every* $s \in Q'$, *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 [7].

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

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

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:

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

obtained from the "*concrete*" system (β_1)–(β_5), where $f_i^\alpha(x_1, \ldots, x_m)$ is an expression obtained by substituting in $f_i(x_1, \ldots, x_m)$ 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.

5.2.1 Finite-chain abstractions:

A *Finite-chain abstraction* is an abstraction such that the semilattice $(D, \sqcup)$ has no infinite ascending chains. Particular cases of such abstractions are *finite abstractions* where the abstract domain D is finite. In this case, the iterative computation of the least fixpoint 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 $WPaths_R(T, T')$.

5.2.2 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 defined 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 $(D, \sqcup, \odot, \bot, \overline{1})$ is a *commutative* idempotent closed semiring. As usual, we define $d^0 = \overline{1}$, $a^{n+1} = a \odot a^n$, and $a^\star = \bigsqcup_{n \geq 0} a^n$. Adding the $\star$-operation transforms the structure above into a *commutative Kleene algebra* $\mathcal{K} = (D, \sqcup, \odot, \star, \bot, \overline{1})$. Then, the system (8) can be solved using the algorithm of Hopkins and Kozen [16] for solving systems of polynomial constraints in commutative Kleene algebras (see also [4]).

Remark 5.1 Notice that to be able to use the framework of commutative Kleene algebras, we need to consider that $\odot$ 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.

References

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

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

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

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

[5] Ahmed Bouajjani and Tayssir Touili. Reachability Analysis of Process Rewrite Systems. In *FSTTCS'03*. LNCS 2914, 2003.

[6] 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.

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

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

[9] E. Duesterwald and M.L. Soffa. Concurrency analysis in the presence of of procedures using a data-fbw framework. In *Proc. of the Symp. on Testing, Analysis, and Verification*. ACM Press, 1991.

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

[11] J. Esparza and J. Knoop. An automata-theoretic approach to interprocedural data-fbw analysis. In *FoSSaCS'99*. LNCS 1578, 1999.

[12] J. Esparza and A. Podelski. Efficient algorithms for pre* and post* on interprocedural parallel fbw graphs. In *POPL'00*, 2000.

[13] 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.

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

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

[16] M.W. Hopkins and D.C. Kozen. Parikh's Theorem in Commutative Kleene Algebra. In *LICS'99*. IEEE, 1999.

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

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

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

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

[21] M. Rinard. Analysis of multithreaded programs. In *SAS'01*. LNCS 2126, 2001.

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