Dealing with communication for dynamic multithreaded recursive programs

Tayssir Touili

\textit{\textsuperscript{a}LIAFA, CNRS \& University of Paris 7, Paris, France}

Abstract. This paper presents a new contribution to the model-checking of multithreaded programs with recursive procedure calls, result passing between recursive procedures, dynamic creation of parallel processes, and synchronisation between parallel threads. To represent such programs \textit{accurately}, we define the model SPAD that can be seen as the extension with synchronisation of the class PAD (the subclass of the rewrite systems PRS where parallel composition is not allowed in the left-hand sides of the rules). We consider in this paper the reachability problem of this model, which is undecidable. As in \cite{BET03a,BET04}, we reduce this problem to the computation of abstractions of the sets of execution paths of the program, and we propose a generic technique that can compute different abstractions (of different precisions and different costs) of these sets.

Keywords. Multithreaded recursive programs, Dynamic creation of processes, Result passing between recursive procedures, Synchronisation between parallel processes, Program analysis, SPAD rewrite systems, Abstraction techniques.

1. Introduction

We consider parallel recursive programs with dynamic creation of parallel processes. We suppose that the infinite types of data have been abstracted towards finite types using the standard methods of abstract interpretation \cite{CC77}. Even after such abstraction, the verification problem of these programs is undecidable in general \cite{Ric53}. We obtain this undecidability as soon as the programs contain synchronisation and recursion \cite{Ram00}. The problem then amounts to finding models that can be analysed, and that are expressive enough to represent significant classes of these parallel recursive programs.

These last years, rewrite systems have been used to model subclasses of such programs. For example, Pushdown systems have been considered for the analysis of sequential programs \cite{EK99,ES01}, Petri nets for the modeling of concurrent non recursive programs \cite{BCR01,DBR02}, PA systems for the programs without communication (by communication, we mean synchronisation between parallel processes, and result passing between recursive procedures) \cite{EK99,EP00}, etc. In our recent work, we have extended these existing approaches towards more expressive models that allow to consider simultaneously recursivity, parallelism, and synchronisation; and we have proposed symbolic analysis methods for these models. More precisely, we have proposed three different models having different expressivities:

In \cite{BT03,Tou03,BT05}, we have considered PRS systems to model concurrent recursive programs, and we have proposed for these systems symbolic analysis algorithms...
based on tree-automata techniques. A PRS is a finite set of rules of the form \( t \rightarrow t' \), where \( t \) and \( t' \) are terms built up from the null process “0”, a finite number of variables (\( X \)), the sequential composition “\( \cdot \)”, and the asynchronuous parallel composition “\( || \)”, where the operators “\( \cdot \)” and “\( || \)” are respectively associative and associative/commutative. Intuitively, the process “0” represents termination, a process variable \( X \) corresponds to a control point of the program, and a process term \( t \) describes the control structure of the program. A procedure call is represented by a rule of the form \( X \rightarrow Y \cdot Z \), where the program at control point \( X \) calls the procedure \( Y \) and goes to control point \( Z \). This control point \( Z \) becomes active when \( Y \) terminates. Suppose the behavior of \( Z \) depends on the result of the computation \( Y \), this is represented by a finite number of rules \( Y^i \cdot Z \rightarrow t^i \) meaning that \( Y \) is evaluated before passing the control to \( Z \), and if \( Y \) returns \( Y^i \), the caller becomes \( t^i \) and resumes its computation. This kind of rules having sequential composition in their left-hand sides model then result passing between recursive procedures (here, \( Y^i \) is the result of the computation of the procedure \( Y \)). Dynamic creation of parallel processes is modeled by rules of the form \( X \rightarrow Y \| Z \), expressing that a process in control point \( X \) can create two parallel processes in control points \( Y \) and \( Z \), respectively. Finally, handshakes between parallel processes are represented by rules of the form \( X \| Y \rightarrow X'||Y' \), meaning that two parallel processes in control points \( X \) and \( Y \), respectively, can synchronize and move simultaneously to control points \( X' \) and \( Y' \), respectively.

Several well known models mentioned above can be seen as subclasses of PRS such as Pushdown systems that correspond to systems without parallel composition, Petri nets are PRS systems with only parallel composition, or PA systems that are PRSs having rules of the form \( X \rightarrow t \). Another subclass of PRS that is relevant for this work is the class \( \text{PAD} \) that corresponds to rules without parallel composition in their left-hand sides, i.e., rules of the form \( X \rightarrow t \) or \( X \cdot Y \rightarrow t \). This class can model recursion and dynamic creation of processes, but not the synchronisation between parallel processes (due to the absence of rules of the form \( X \| Y \rightarrow t \)).

Despite the fact that PRS subsumes all these well known models, it is not powerful enough to represent accurately the synchronisation between parallel processes in an exact manner. Indeed, to represent the fact that two control points \( X \) and \( Y \) have to synchronise and move simultaneously to control points \( X' \) and \( Y' \) respectively, an infinite number of rules is needed: besides the rule \( X \| Y \rightarrow X'||Y' \), we need for example the rules

\[
(X \cdot X_1) \| Y \rightarrow (X' \cdot X_1) \| Y', \quad \text{and}
\]

\[
(X \cdot X_1 \cdot X_2) \| Y \rightarrow (X' \cdot X_1 \cdot X_2) \| Y', \quad \text{and}
\]

\[
(X \cdot X_1 \cdot X_2 \cdot X_3) \| Y \rightarrow (X' \cdot X_1 \cdot X_2 \cdot X_3) \| Y', \quad \text{etc}
\]

since the control point \( X \) is active in all the configurations \( X \cdot X_1 \cdot X_2 \cdots \).

Since PRS cannot model synchronisation accurately, we have considered other models that can represent synchronisation in an \textit{exact} manner. However, the models that we considered cannot represent other aspects such as dynamic creation of parallel processes, or result passing between recursive procedures. The idea was to extend subclasses of PRS with synchronisation operators according to the CCS [Mil80] style: In a first step
[BET03a,BET03b], we extended the Pushdown System subclass towards a model called *Communicating Pushdown Systems (CPDS)*. This model can represent programs containing a finite fixed number of parallel recursive processes, i.e., parallel recursive programs without dynamic creation of parallel processes. In a second step [BET04], we extended the subclass PA towards a model called *SPA* for *Synchronised PA*. Compared to CPDS, this model can deal with dynamic creation of processes, but it cannot represent result passing between recursive procedures (this is due to the limits of the expressive power of PA).

In this work, to overcome this restriction, we go one step further in the PRS hierarchy, and extend the subclass PAD (which subsumes both pushdown and PA systems) with synchronisation, which allows to model all the aspects mentioned previously in an exact manner (dynamic creation of parallel processes, recursion, and communication, i.e., result passing between recursive procedures and synchronisation between parallel processes). The idea is the same as previously: extend PAD with synchronisation and restriction operators according to the CCS style, obtaining thus a new model called *SPAD* (for *Synchronized PAD systems*) that is more powerful than all the previous models that we have considered (PRS, CPDS, and SPA).

As usual in program analysis, we are interested in the reachability analysis of this model. More precisely, given two sets of configurations $L$ and $L'$, the problem is to know whether $L'$ can be reached from $L$. This amounts to computing the set of execution paths $\text{Paths}(L, L')$ that lead the system from $L$ to $L'$, and checking its emptiness. However, as we mentioned previously, model checking programs with synchronisation and recursion is undecidable [Ram00]. Therefore, the set of execution paths $\text{Paths}(L, L')$ cannot be computed in an exact manner. To overcome this problem, we proceed as in [BET03a,BET03b,BET04]: Our approach is based on the computation of abstractions of the execution path language $\text{Paths}(L, L')$. To this aim, following what we did in [BET04] for SPA, we propose techniques based on (1) the representation of the sets of configurations with binary tree automata, (2) the use of these automata to compute a set of constraints whose least fixpoint characterize the set of execution paths of the program, and (3) the resolution of this set of constraints in an abstract domain. We consider in particular the case where the abstract domain does not contain an infinite ascending chain, since in this case, the set of constraints can be solved using an iterative computation. We show on an example how this kind of abstractions can be used in program analysis.

**Related Work.** Besides the works mentioned above, there are a lot of other techniques and tools that have been considered for program verification. However, none of these works can deal with communication, parallelism, dynamic process creation, and recursion at the same time. We can for example mention the works [SS00,MO02,DS91], and the tools BLAST [HJMS02], SLAM [BR01], KISS [QW04,QR05], ZING [QRR04], and MAGIC [CCG+03,CCG+04]. SLAM can only handle pure recursive programs where no concurrency is allowed. BLAST uses a different approach based on an “assume-guarantee” reasoning. Moreover, the BLAST framework is based on shared variables, whereas our concurrent components communicate via synchronization actions. KISS cannot discover errors that arise after a number of interleaveings between the parallel components greater than some finite bound. ZING is based on summarization, a technique whose termination is not guaranteed. As for MAGIC, it is based on finite-state models, and therefore, it does not take into account recursion nor dynamic creation of processes.
For other existing techniques, we refer to [Rin01] for a good survey.

The remainder of the paper is organized as follows: We present our new model in Section 2. In Section 3, we formulate our reachability problem. We describe our algorithm for computing abstractions of execution paths for SPADs in Section 4. Finally, we illustrate our method on an example of a program having both dynamic creation of processes and result passing between recursive procedures in Section 5.

2. The model: Synchronised PAD systems

We introduce in this section our SPAD model that describes recursive multithreaded programs. As mentioned earlier, this model is a kind of extension of PAD systems in the CCS style [Mil80] with synchronisation and restriction operators.

2.1. Syntax

Let $\text{Sync} = \{a, b, c, \ldots\}$ be a set of visible actions such that every action $a \in \text{Sync}$ corresponds to a co-action $\bar{a}$ in $\text{Sync}$ s.t. $\bar{a} = a$. Let $\text{Act} = \text{Sync} \cup \{\tau\}$ be the set of all the actions, where $\tau$ is a special action (as we will see, this special action represents internal actions and the handshakes).

Let $\text{Var} = \{X, Y, \cdots\}$ be a set of process variables, and $\mathcal{T}$ be the set of process terms $t$ over $\text{Var}$ defined by:

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

We define the set of restricted process terms as follows: $\mathcal{T}_r = \{t \backslash S \mid t \in \mathcal{T}, S \subseteq \text{Sync}\}$. Intuitively, the term “$t \backslash S$" corresponds to the restriction of the behavior of the term $t$ to the actions that are not in $S$.

Let $\mathcal{L}$ be a set of process terms, and $S$ be a subset of $\text{Sync}$. We denote by $\mathcal{L} \backslash S$ the set $\{t \backslash S \mid t \in \mathcal{L}\}$.

**Definition 2.1** A Synchronized PAD (SPAD for short) is a finite set of rules of the form $\overset{a}{X \rightarrow t}$, or $\overset{a}{X \cdot Y \rightarrow t}$, where $X, Y \in \text{Var}$, $t \in \mathcal{T}$, and $a \in \text{Act}$.

2.2. Semantics

2.2.1. Structural equivalences on terms:

We consider the equivalence relation $\sim_0$ on $\mathcal{T}$ defined by the neutrality of “$0$" w.r.t. “$\|$", and “$\cdot$";

$$t \cdot 0 \sim_0 0 \cdot t \sim_0 t \| 0 \sim_0 0 \| t \sim_0 t$$

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

$$\begin{align*}
\text{A2:} & \quad (t \cdot t') \cdot t'' \sim t \cdot (t' \cdot t'') \quad : \text{associativity of } "\cdot", \\
\text{A3:} & \quad t\|t' \sim t'\|t \quad : \text{commutativity of } "\|", \\
\text{A4:} & \quad (t\|t')\|t'' \sim t\|(t'\|t'') \quad : \text{associativity of } "\|".
\end{align*}$$
The equivalences above are extended to terms of \( T_r \) by considering that \( t \setminus S \equiv t' \setminus S \) iff \( t \equiv t' \), where \( \equiv \) is an equivalence from the set \( \{ \sim_0, \sim \} \). Let \( t \in T_r \cup T \), we denote by \( [t]_\equiv \) the equivalence class modulo \( \equiv \) of the process term \( t \), i.e., \( [t]_\equiv = \{ t' \in T_r \cup T \mid t \equiv t' \} \). A set of terms \( L \) is said to be \( \equiv \)-compatible if \( [L]_\equiv = L \).

2.2.2. Transition relations:

An SPAD \( R \) induces a transition relation \( \overset{a}{\rightarrow} \) over \( T \) and a transition relation \( \overset{a}{\rightarrow}_\equiv \) over \( T_r \) defined by the following inference rules:

\[
\begin{align*}
\theta_1 & : \frac{t_1 \overset{a}{\rightarrow} t_2 \in R}{t_1 \overset{a}{\rightarrow} t_2}, \\
\theta_2 & : \frac{t_1 \overset{a}{\rightarrow} t'_1, t_2 \overset{a}{\rightarrow} t'_2}{t_1 \cdot t_2 \overset{a}{\rightarrow} t'_1 \cdot t'_2}, \\
\theta_3 & : \frac{t_1 \sim_0 0, t_2 \overset{a}{\rightarrow} t'_2}{t_1 \cdot t_2 \overset{a}{\rightarrow} t_1 \cdot t'_2}, \\
\theta_4 & : \frac{t_1 \overset{a}{\rightarrow} t'_1, t_2 \overset{a}{\rightarrow} t'_2; a, \bar{a} \in \text{Sync}}{t_1||t_2 \overset{a}{\rightarrow} t'_1||t'_2}, \\
\theta_5 & : \frac{t_1 \overset{a}{\rightarrow} t'_1, t_2 \overset{a}{\rightarrow} t'_2; a, \bar{a} \in \text{Sync}}{t_1||t_2 \overset{a}{\rightarrow} t'_1||t'_2}, \\
\theta_6 & : \frac{t_1 \overset{a}{\rightarrow} t_2; a \notin S, \bar{a} \notin S}{t_1\setminus S \overset{a}{\rightarrow} t_2\setminus S}.
\end{align*}
\]

We define in a standard manner the relations \( \overset{w}{\rightarrow} \) and \( \overset{w}{\rightarrow}_\equiv \), for \( w \in \text{Act}^* \).

Each equivalence \( \equiv \) in \( \{ \sim, \sim_0 \} \) induces a transition relation \( \overset{a}{\rightarrow}_\equiv \) over \( T \) and a transition relation \( \overset{a}{\rightarrow}_\equiv \) over \( T_r \) defined by:

\[\forall t, t' \in T, t \overset{a}{\rightarrow}_\equiv t' \text{ iff } \exists u, u' \text{ s.t. } t \equiv u, u \overset{a}{\rightarrow} u', \text{ and } u' \equiv t',\]

and

\[\forall t, t' \in T, t \setminus S \overset{a}{\rightarrow}_\equiv t' \setminus S \text{ iff } \exists u, u' \text{ s.t. } t \equiv u, u \setminus S \overset{a}{\rightarrow} u' \setminus S, \text{ and } u' \equiv t'.\]

We extend these relations to sequences of actions in the obvious manner.

Let for \( t \in T \),

\[\text{Post}^*_{\equiv}[w](t) = \{ t' \in T \mid t \overset{w}{\rightarrow}_\equiv t' \}, \text{Post}^*_{\equiv}(t) = \bigcup_{w \in \text{Act}^*} \text{Post}^*_{\equiv}[w](t).\]

And for \( t \in T_r \),

\[\text{Post}^*_{\sim,\equiv}[w](t) = \{ t' \in T_r \mid t \overset{w}{\rightarrow}_\equiv t' \}, \text{Post}^*_{\sim,\equiv}(t) = \bigcup_{w \in \text{Act}^*} \text{Post}^*_{\sim,\equiv}[w](t).\]

We will simply write \( \text{Post}^*[w](t), \text{Post}^*(t), \text{Post}^*_{\sim}[w](t), \) and \( \text{Post}^*_{\sim}(t) \) to denote \( \text{Post}^*[w](t), \text{Post}^*(t), \text{Post}^*_{\sim}[w](t) \) and \( \text{Post}^*_{\sim}(t) \), respectively.

These definitions are extended to sets of terms of \( T \) and \( T_r \) in the standard way.

\(^1\)Note that the transition rules \( \theta_1 - \theta_4 \) define the semantics of a PAD system [May98].
2.3. From a program to an SPAD

As described previously, termination, procedure calls, result passing between recursive procedures, and process creation are respectively represented by the PAD rules $X \xrightarrow{a} 0$, $X \xrightarrow{a} Y \cdot Z, Y \cdot Z \xrightarrow{a} t$, and $X \xrightarrow{a} Y \mid Z$. As for handshaking, it is modeled as follows: Let $X$ and $Y$ be two control points that have to synchronize and move simultaneously to control points $X'$ and $Y'$. We represent this synchronisation by the following SPAD rules, where $a \in \text{Sync}$:

$$X \xrightarrow{a} X', \quad \text{and} \quad Y \xrightarrow{\bar{a}} Y'$$

Observe that with this modeling, the rules $\theta_5$ represent synchronisation in an exact manner: if the parallel processes $t_1$ and $t_2$ perform the synchronizing actions $a$ and $\bar{a}$, respectively, they get synchronized and evolve simultaneously to $t'_1$ and $t'_2$, respectively. However, the rules $\theta_4$ allow to every process to evolve independently of the other parallel processes. These rules introduce then additional unwanted behaviors in the system. To avoid this, the rules $\theta_4$

$$\theta_4 : 
\frac{t_1 \xrightarrow{a} t'_1}{t_1 \mid t_2 \xrightarrow{a} t'_1 \mid t_2} ; \quad t_2 \xrightarrow{a} t_2 \xrightarrow{a} t'_1}$$

should be applied only if $a$ is an internal action, i.e., if $a \notin \text{Sync}$ (i.e., $a = \tau$). Therefore, two processes get synchronised correctly iff they execute the action $\tau$. With the SPAD semantics, this amounts to considering the restricted process terms $t \mid \text{Sync}$. Therefore, the set of reachable configurations of the program, starting from a set of configurations given by a set of process terms $L$, is represented by $\text{Post}^*_\rightarrow (L \setminus \text{Sync})$.

3. The reachability problem for SPAD

Let $\mathcal{R}$ be an SPAD, and let $\mathcal{L}$ and $\mathcal{L}'$ be two sets of process terms over $\mathcal{T}$. The problem consists in checking whether

$$\text{Post}^*_\rightarrow (\mathcal{L} \setminus \text{Sync}) \cap \mathcal{L}' \setminus \text{Sync} = \emptyset. \quad (1)$$

Unfortunately, this question is undecidable [BET04]. To tackle this problem, we adopt the same approach as in [BET04], and we translate it into a problem on the sets of execution paths. Let

$$\text{Paths}(\mathcal{L}, \mathcal{L}') = \{ w \in \text{Act}^* \mid \exists t \in \mathcal{L}, t' \in \mathcal{L}', t' \in \text{Post}^*[w](t) \}.$$ 

This set corresponds to all the execution paths that lead from terms of $\mathcal{L}$ to terms of $\mathcal{L}'$ even by application of the rules ($\theta_4$) when $a \neq \tau$. Since the behaviors of the terms of $\mathcal{L} \setminus \text{Sync}$ are restricted to $\tau$ actions, the execution paths that can lead from $\mathcal{L} \setminus \text{Sync}$ to $\mathcal{L}' \setminus \text{Sync}$ is equal to $\text{Paths}(\mathcal{L}, \mathcal{L}') \cap \tau^*$. Therefore, the problem (1) amounts to check whether
However, the set \( \text{Paths}(L, L') \) cannot be computed because of the undecidability result. As in [BET04], our approach consists in computing an over-approximation \( A(L, L') \) of \( \text{Paths}(L, L') \) such that if \( A(L, L') \cap \tau^* \) is empty, then so is \( \text{Paths}(L, L') \cap \tau^* \).

To compute such over-approximations, we extend the technique presented in [BET03a,BET03b,BET04] to SPADs. More precisely, we define a generic approach based on (1) the characterisation of the set \( \text{Paths}(L, L') \) as the least solution of a system of constraints over word languages (this solution cannot be computed in general), and (2) the computation of the least solution in an abstract domain.

4. Characterizing the path languages

We present in this section the approach that we adopt to characterize \( \text{Paths}(L, L') \) as the least solution of a system of constraints over word languages. Following the approach presented in [BET04], (1) we use binary trees to represent the process terms of \( T \), (2) we use binary tree automata to represent regular sets of process terms, (3) we restrict ourselves to the case where \( L \) and \( L' \) are \( \sim \)-compatible, and (4) we reduce the problem to the characterization of the set of sequences of execution paths \( \{ w \in \text{Act}^* \mid \text{Post}^\sim_0[w](L) \cap L' \neq \emptyset \} \). Indeed, if \( L \) and \( L' \) are \( \sim \)-compatible then this set is equal to \( \text{Paths}(L, L') \):

**Proposition 4.1** If \( L \) and \( L' \) are \( \sim \)-compatible then

\[
\text{Paths}(L, L') = \{ w \in \text{Act}^* \mid \text{Post}^\sim_0[w](L) \cap L' \neq \emptyset \}
\]

**Proof:** This is due to the fact that if \( L \) is \( \sim \)-compatible then

\[
[\text{Post}^\sim_0[w](L)]_{\sim} = \text{Post}^* [w](L)
\]

The proof of this fact can be found in [Tou03] (Proposition 5.2.6) since the relation \( \sim \) defines a semantics for SPAD that is the same than the one defined for PAD.

The different steps of our technique follow the lines of the approach presented in [BET04] except that in the current case, we need to take into account the neutrality of “0”, whereas for SPA, we were able to forget about the structural equivalences and characterize just \( \text{Paths}(L, L') = \{ w \in \text{Act}^* \mid \text{Post}^\sim_0[w](L) \cap L' \neq \emptyset \} \), where \( = \) denotes term equality. This difference is due to the presence of the sequential composition in the left-hand sides of the rules for SPAD.

4.1. Process tree automata

Terms in \( T \) can be seen as binary trees where the leaves are labeled with process constants, and the inner nodes with the binary operators “·” and “∥”. Therefore, regular sets of process terms in \( T \) can be represented by means of a kind of finite bottom-up tree automata [CDG+97], called process tree automata, defined as follows:
Definition 4.1 A process tree automaton is a tuple $A = (Q, \text{Var}, F, \delta)$ where $Q$ is a finite set of states, $\text{Var}$ is a set of process variables, $F \subseteq Q$ is a set of final states, and $\delta$ is a set of rules of the form (a) $f(q_1, q_2) \rightarrow q$, (b) $X \rightarrow q$, or (c) $q \rightarrow q'$, where $X \in \text{Var}$, $f \in \{\|, \cdot\}$, 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 $\cdot_t(1, 2)$ (resp. $\|_t(1, 2)$). 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 $f(q_1, q_2) \rightarrow q$ is in $\delta$ 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.

4.2. Formulation of the problem in terms of process tree automata

Let $\mathcal{L}$ and $\mathcal{L}'$ be two regular $\sim$-compatible sets of terms. Let $A = (Q, \Sigma, F, \delta)$ and $A' = (Q', \Sigma, F', \delta')$ be two process tree automata that recognize $\mathcal{L}$ and $\mathcal{L}'$, respectively. We suppose w.l.o.g. that $Q'$ is such that if $s \in Q'$, then there exists a state $s^\text{null}$ in $Q'$ that recognizes exactly the null terms of $L_s$, i.e., that recognizes the language $L_s \cap \{u \in T \mid u \sim 0\}$. The state $s'^\text{null}$ is obviously equal to $s^\text{null}$.

Besides, let $s_0$ be a new state, and let $\delta_0$ be the following rules:

- $0 \rightarrow s_0$,
- $\cdot(s_0, s_0) \rightarrow s_0$, and
- $\| (s_0, s_0) \rightarrow s_0$.

It is easy to see that $s_0$ recognizes the set of all the null terms of $T$.

Let $Q^R = \{q_t \mid t$ is a subterm of the members of the rules of $\mathcal{R}\}$, and $\delta^R$ be the following set of transition rules:

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

It is then clear that for every subterm $t$ of a member of a rule in $\mathcal{R}$, $L_{q_t} = \{t\}$. Let $Q = Q \cup Q^R$, $\Delta = \delta \cup \delta^R$, $Q' = Q' \cup Q^R \cup \{s_0\}$, and $\Delta' = \delta' \cup \delta^R \cup \delta_0$. Let $q \in Q$ and $s \in Q'$. We define the set of execution paths $\lambda(q, s)$ by:

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

We get then from Proposition 4.1:

---

2Such states can be obtained by computing the product of $A'$ with the automaton recognizing the null terms, and containing the rules $0 \rightarrow q^\text{null}$, $\cdot(q^\text{null}, q^\text{null}) \rightarrow q^\text{null}$, and $\| (q^\text{null}, q^\text{null}) \rightarrow q^\text{null}$. The state $(s, q^\text{null})$ corresponds then to $s'^\text{null}$. 

---
Proposition 4.2 If \( \mathcal{L} \) and \( \mathcal{L}' \) are \( \sim \)-compatible, then

\[
\text{Paths}(\mathcal{L}, \mathcal{L}') = \bigcup_{q \in \mathcal{P}} \lambda(q, s).
\]

Therefore, our aim is this section is to characterize the sets \( \lambda(q, s) \). For that, we define a system of constraints whose least solution corresponds to these sets. To compute this system, we need to consider the operator \( \Pi \) defined in [BET04] that is a word operator corresponding to the parallel composition between terms. This operator is defined inductively as follows:

\[
\begin{align*}
\epsilon \Pi w &= w \\
aw_1 \Pi \bar{a}w_2 &= a(w_1 \Pi \bar{a}w_2) + \bar{a}(aw_1 \Pi w_2) + \tau(w_1 \Pi w_2) \\
aw_1 \Pi bw_2 &= a(w_1 \Pi bw_2) + b(aw_1 \Pi w_2) \text{ if } b \neq \bar{a}
\end{align*}
\]

With this definition, we have that if \( u_1 \) is reachable from \( v_1 \) with the sequence of actions \( w_1 \), and \( u_2 \) is reachable from \( v_2 \) with the sequence of actions \( w_2 \), then \( u = ||(u_1, u_2) \) is reachable from \( v = ||(v_1, v_2) \) with the sequence of actions \( w_1 \Pi w_2 \):

Lemma 4.1 Let \( u_1, u_2, v_1, v_2 \in \mathcal{T} \) and \( w \in \text{Act}^* \). Then

\[
||(u_1, u_2) \in \text{Post}^*_\sim[w](||(v_1, v_2)) \text{ s.t. the root } || \text{ was not rewritten}
\]

iff there exist \( w_1, w_2 \in \text{Act}^* \text{s.t. } w \in w_1 \Pi w_2, u_1 \in \text{Post}^*_\sim[w_1](v_1), \text{ and } u_2 \in \text{Post}^*_\sim[w_2](v_2).

\textbf{Proof:} The proof follows the lines of the proof of Lemma 4.3 in [BET04]. \qed

4.3. The system of constraints

We associate to the states \( q \in \mathcal{Q} \) and \( s \in \mathcal{Q}' \) variables \( l(q, s) \) representing subsets of \( \text{Act}^* \), and we define a set of constraints over these variables. We show that the least solution of this system corresponds to the sets \( \lambda(q, s) \). The system of constraints is defined as follows:

\begin{enumerate}
\item[(\( \beta_1 \))] if \([L_q]_{\sim} \cap L_s \neq \emptyset \), then

\[
e \in l(q, s)
\]

\item[(\( \beta_2 \))] if \( q_1 \rightarrow q_2 \) is a rule of \( \Delta \) and \( s_1 \rightarrow s_2 \) is a rule of \( \Delta' \), then

\[
l(q_1, s_1) \subseteq l(q_2, s_2)
\]

\item[(\( \beta_3 \))] if \( \cdot(q_1, q_2) \rightarrow q \) is a rule of \( \Delta \) and \( \cdot(s_1, s_2) \rightarrow s \) is a rule of \( \Delta' \), then

\[
l(q_1, s_1^{null}) : l(q_2, s_2) \subseteq l(q, s)
\]
\end{enumerate}
and
\[ l(q_1, s_1) \subseteq l(q, s) \text{ if } [L_{q_2}]_{\sim_0} \cap L_{s_2} \neq \emptyset \]

(\(\beta_4\)) if \(((q_1, q_2) \rightarrow q)\) is a rule of \(\Delta\) and \(((s_1, s_2) \rightarrow s)\) is a rule of \(\Delta'\), then
\[ l(q_1, s_1) \Pi l(q_2, s_2) \subseteq l(q, s) \]

(\(\beta_5\)) if \(t_1 \xrightarrow{a} t_2 \in \mathcal{R}\), then
\[ l(q, q_{t_1}) \cdot a \cdot l(q_{t_2}, s) \subseteq l(q, s) \]

(\(\beta_6\)) if \((q_1, q_2) \rightarrow q)\) is a rule of \(\Delta\), then
\[ l(q_1, s_0) \cdot l(q_2, s) \subseteq l(q, s) \]

(\(\beta_7\)) if \(((q_1, q_2) \rightarrow q)\) is a rule of \(\Delta\), then
\[ l(q_1, s_0) \Pi l(q_2, s) \subseteq l(q, s) \]

and
\[ l(q_1, s) \Pi l(q_2, s_0) \subseteq l(q, s) \]

We explain in what follows the intuition behind these rules. Recall that the variables 
\(l(q, s)\) are meant to represent the sets \(\lambda(q, s)\). The rules (\(\beta_1\)) express that if \([L_0]_{\sim_0} \cap L_{s_2} \neq \emptyset\), then \(e \in \lambda(q, s)\). The rules (\(\beta_2\)) express 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 rules (\(\beta_3\)) express that if \((q_1, q_2) \rightarrow q\) is a rule of \(\Delta\), and \((s_1, s_2) \rightarrow s)\) is a rule of \(\Delta'\), then \(u_1 \in L_{s_1}, u_2 \in L_{s_2}, v_1 \in L_{q_1}\), and \(v_2 \in L_{q_2}\) are such that:

- if \(u_1\) is null \((u_1 \sim_0 0)\), and therefore recognized by \(s_1^{null}\), and if it is reachable from \(v_1\) by applying the sequence of actions \(w_1 (w_1 \in \lambda(q_1, s_1^{null}))\), and \(u_2\) is reachable from \(v_2\) by applying the sequence of actions \(w_2 (w_2 \in \lambda(q_2, s_2))\), then \(u = (u_1, u_2)\) \((u\) is in \(L_{s}\)) is reachable from \(v = (v_1, v_2)\) \((v\) is in \(L_{q})\) by applying \(w_1 w_2 (w_1 w_2 \in \lambda(q, s))\).
- if \(u_1\) is reachable from \(v_1\) by applying the sequence of actions \(w_1 (w_1 \in \lambda(q_1, s_1))\), and \(v_2 \in [L_{q_2}]_{\sim_0} \cap L_{s_2}\), then \(u = (u_1, u_2)\) \((u\) is in \(L_{s}\)) is reachable from \(v = (v_1, v_2)\) \((v\) is in \(L_{q})\) by applying \(w_1 (w_1 \in \lambda(q, s))\).

The rules (\(\beta_4\)) express that if \(((q_1, q_2) \rightarrow q)\) is a rule of \(\Delta\), and \(((s_1, s_2) \rightarrow s)\) is a rule of \(\Delta'\), then \(u_1 \in L_{s_1}, u_2 \in L_{s_2}, v_1 \in L_{q_1}\), and \(v_2 \in L_{q_2}\) are such that \(u_1\) is reachable from \(v_1\) by applying the sequence of actions \(w_1 (w_1 \in \lambda(q_1, s_1))\), and \(u_2\) is reachable from \(v_2\) by applying the sequence of actions \(w_2 (w_2 \in \lambda(q_2, s_2))\), then \(u = (u_1, u_2)\) \((u\) is in \(L_{s}\)) is reachable from \(v = (v_1, v_2)\) \((v\) is in \(L_{q})\) by applying any sequence of actions \(w\) in \(w_1 \Pi w_2 (w_1 \Pi w_2 \subseteq \lambda(q, s))\). The rules (\(\beta_5\)) express that if \(t_1\) is reachable from \(L_{q}\) with the sequence of actions \(w_1 (w_1 \in \lambda(q, q_{t_1}))\), if \(t_1 \xrightarrow{a} t_2 \in \mathcal{R}\), and if there
exists a term \( u \) in \( L_s \) that can be reached from \( t_2 \) with the sequence \( w_2 \) \((w_2 \in \lambda(q_{t_2}, s))\), then \( u \) can be reached from \( L_q \) with the sequence \( w_1 \cdot a \cdot w_2 \) \((w_1 \cdot a \cdot w_2 \in \lambda(q, s))\).

Rules \((\beta_0)\) mean that if \( (q_1, q_2) \rightarrow q \) is a rule of \( \Delta \), \( u_1 \sim_0 0, u \in L_{t_2}, v_2 \in L_{q_2} \), and \( v_2 \in L_{q_2} \), such that \( u_1 \) is reachable from \( v_1 \) by applying the sequence of actions \( w_1 \) \((w_1 \in \lambda(q_1, s_0))\), and \( u \) is reachable from \( v_2 \) by applying the sequence of actions \( w_2 \) \((w_2 \in \lambda(q_2, s))\), then \( u \sim_0 (u_1, v_2) \) is reachable from \( v = (v_1, v_2) \) by applying \( w_1 \cdot w_2 \) \((w_1 \cdot w_2 \in \lambda(q, s))\). Finally, rules \((\beta_7)\) are similar to rules \((\beta_0)\).

As in [BET04], we can show using Tarski’s and Kleene’s Theorems that the least solution of the system above exists. Let \( \{e(q, s)\} \in Q \) be this solution. We show that for every \( q \in Q \) and every \( s \in Q' \), \( e(q, s) \) is equal to the path language \( \lambda(q, s) \). The proof can be found in the full version of the paper.

**Theorem 4.1** For every \( q \in Q \) and every \( s \in Q' \),

\[ e(q, s) = \lambda(q, s). \]

### 4.4. Computing the solution of the system of constraints

As mentioned previously, it is not always possible to compute the languages \( \lambda(q, s) \). Therefore, we need to compute abstractions of these languages as done in [BET03a, BET03b,BET04]. Let us summarize this approach.

Consider an abstract lattice \((D, \leq, \cap, \cup, \perp, \top, 0, 1)\) associated with a structure \((D, \oplus, \odot, \otimes, 0, 1)\) such that \( \oplus = \cup \) is an associative, commutative, and idempotent \((a \oplus a = a)\) operation; \( \odot \) is an associative operation; \( \top = \perp; 0 \) and \( 1 \) are neutral elements for \( \oplus \) and \( \odot \), respectively; \( 0 \) is an annihilator for \( \odot (a \odot 0 = 0 \odot a = 0) \); and \( \odot \) distributes over \( \oplus \). Finally, \( \leq \) is such that \( x \leq x \oplus a \).

\( D \) is related to the concrete domain \( 2^{Act^*} \) as follows:

- It contains an element \( v_a \) for every letter \( a \in Act \).
- It is associated with an abstraction function \( \alpha \) and a concretization function \( \gamma \) defined as follows:

\[
\alpha(L) = \bigoplus_{a_1 \cdots a_n \in L} v_{a_1} \odot \cdots \odot v_{a_n}
\]

and

\[
\gamma(x) = \{ a_1 \cdots a_n \in 2^{Act^*} \mid v_{a_1} \odot \cdots \odot v_{a_n} \leq x \}
\]

It is easy to see that for every language \( L \subseteq Act^* \); \( \alpha(L) \in D \), and \( \gamma(\alpha(L)) \supseteq L \).

In other words, \( \gamma(\alpha(L)) \) is an over-approximation of \( L \) that is finitely represented in the abstract domain \( D \) by the element \( \alpha(L) \). Intuitively, the abstract operations \( \odot, \otimes, \) and \( \oplus \) correspond to concatenation, \( \cup \), and union respectively, \( \leq \) and \( \cap \) correspond to inclusion and intersection respectively, and the abstract elements \( 0 \) and \( 1 \) correspond to the empty language and \( \{ \epsilon \} \) respectively.

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

\[ \forall L_1, L_2, \alpha(L_1) \cap \alpha(L_2) = \perp \implies L_1 \cap L_2 = \emptyset. \]
Therefore, to determine whether \( \text{Paths}(L, L') \cap \tau^* \neq \emptyset \), it suffices to check whether
\[
\alpha(\text{Paths}(L, L')) \sqcap \alpha(\tau^*) \neq \bot
\]

Let a finite-chain abstraction be an abstraction such that \( D \) does not contain an infinite ascending chain. Then, an iterative computation of the least solution of the above set of constraints will terminate. In [BET03a, BET03b, BET04], we have given different examples of “natural” finite-chain abstractions that can be used in program analysis, and that offer different analysis algorithms having different degrees of precision and cost. For example, we have defined the “First occurrence ordering” abstraction (that we will need to analyse the example of the next section) as follows:

**First occurrence ordering**  Let \( W = \{ w \in \text{Act}^* \mid \forall a \in \text{Act}, |w|_a \leq 1 \} \), i.e., the set of words where each letter occurs at most once. We consider the abstraction given by:

1. \( D = 2^W \); this set is generated by the elements \( v_a \) for each \( a \) in \( \text{Act} \), where \( v_a = \{ a \} \).
2. \( \leq = \subseteq \).
3. \( \oplus = \cup \).
4. \( U \circ V = \{ u_1 \cdot v'_2 \mid u_1 \in U \text{ and } \exists v_2 \in V. v'_2 \text{ is the projection of } v_2 \text{ on the set of letters which do not occur in } u_1 \} \).
5. \( \otimes = \sqcup \).
6. \( 0 = \emptyset \).
7. \( 1 = \{ \epsilon \} \).
8. \( \top = \text{Act} \), and \( \sqcap = \cap \).

5. An example

To illustrate our technique, let us consider the example of the program represented in Figure 1 that involves dynamic creation of processes (at point \( n_0 \)) and result passing between recursive procedures (at point \( m_4 \)). The figure represents the flow graph of a program having two procedures \( \pi_1 \) and \( \pi_2 \) such that:

- \( \pi_1 \) calls itself in parallel with another procedure \( \pi_2 \).
- \( \pi_2 \) calls itself recursively.
- \( \pi_1 \) and \( \pi_2 \) communicate via the synchronizing actions \( a \) and \( b \), and their corresponding co-actions \( \bar{a} \) and \( \bar{b} \), and
- the program starts at point \( n_0 \).

![Diagram](image)

This program can be modeled by the SPAD \( \mathcal{R} \) that has the following rules:
Using the “First occurrence ordering abstraction”, we can show that starting from $n_0$, the program can never reach a configuration where the control point $m_1$ is active. To do so, let us consider the tree languages $L = \{n_0\}$, and $L'$ the set of binary trees where $m_1$ is active. Our problem amounts then to check whether

$$\alpha(Paths(L, L')) \cap \alpha(\tau^*) = \emptyset.$$ 

I.e., since in this case $\alpha(\tau^*) = \{\tau, \epsilon\}$, to check whether

$$\alpha(Paths(L, L')) \cap \{\tau, \epsilon\} = \emptyset$$

To apply the technique described above, we need to define process tree automata that recognize $L$ and $L'$. Let then $A = (Q, \Sigma, F, \delta)$ be a process tree automaton that recognizes $L$, where $Q = F = \{p\}$ and $\delta = n_0 \rightarrow p$; and let $A' = (Q', \Sigma, F', \delta')$ be a process tree automaton that recognizes $L'$ s.t. $Q = \{s, s_1\}$, $F = \{s_1\}$ and $\delta'$ is the following set of rules:

- $m \rightarrow s$ for every $m \in \{n_0, n_1, n_2, m_0, m_1, m_2, m_3, m_4\}$;
- $m_1 \rightarrow s_1$;
- $\|$($s$, $s$) $\rightarrow s$, $\|$($s_1$, $s$) $\rightarrow s_1$, $\|$($s$, $s_1$) $\rightarrow s_1$, and $\cdot$($s_1$, $s$) $\rightarrow s_1$.

We need to compute $\alpha(\lambda(p, s_1))$. After computing the system of constraints as described previously, and solving it in the case of the “First occurrence ordering abstraction”, we obtain that $\alpha(\lambda(p, s_1))$ is equal to the following set of words: $\{\tau b, \tau b\bar{a}, \tau ab, \tau ba, \tau b\bar{a}a, \tau ab\bar{a}, \tau a, \tau a\bar{b}b, \tau a\bar{b}b\bar{a}, \tau b, \tau b\bar{a}, \tau b\bar{a}a, \tau b\bar{a}b, \tau\}$. Since the intersection of this set with $\{\tau, \epsilon\}$ is empty, we infer that $L'$ is not reachable from $L$. This means that $m_1$ cannot be reached from $n_0$.

6. Conclusion

In this paper, we have defined a new model called SPAD that can represent in an accurate manner recursive procedure calls, result passing between recursive procedures, dynamic creation of processes, and synchronisation between parallel processes. The reachability question of this model being undecidable, we sidestep this problem by computing abstractions of execution paths of SPAD processes. To this aim, we extended the approach of [BET04] towards a generic technique for the analysis of SPADs that allows the computation of different abstractions (with different precisions and different costs) of the set of possible execution paths of the system.
References


A. Proof of Theorem 4.1

Theorem 4.1. For every $q \in Q$ and every $s \in Q'$,

$$e(q, s) = \lambda(q, s).$$

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

$$Post_{s_0}^*[w](L_q) \cap L_s \neq \emptyset \iff w \in e(q, s).$$

We start with the implication $\Rightarrow$: Let $q \in Q$, $s \in Q'$, $w \in Act^*$, and $u \in Post_{s_0}^*[w](L_q) \cap L_s$. Let then $v \in L_q$ s.t. $u \in Post_{s_0}^*[w](v)$. We proceed by induction on $|w|$.

- $|w| = 0$, i.e., $w = \epsilon$. Then $u \in [L_q]_{s_0} \cap L_s$, which means that $\epsilon \in e(q, s)$ (from $(\beta_1)$).

- $|w| > 0$. There are two cases:
  1. The root of $v$ has been rewritten. There are three cases:
     (a) There exists a rule $t_1 \xrightarrow{a} t_2 \in R$, $w_1, w_2 \in Act^*$ s.t.

     $$t_1 \in Post_{s_0}^*[w_1](v), u \in Post_{s_0}^*[w_2](t_2),$$

     and $w = w_1aw_2$. Since $|w_1| < |w|$ and $|w_2| < |w|$, the induction hypothesis implies that $w_1 \in e(q, q_{t_1})$, and $w_2 \in e(q_{t_2}, s)$. Therefore, we obtain from $(\beta_2)$ that

     $$w = w_1 \cdot a \cdot w_2 \in e(q, q_{t_1}) \cdot a \cdot e(q_{t_2}, s) \subseteq e(q, s)$$

     (b) $v = \cdot(v_1, v_2)$, and there exists $w_1, w_2, u' \in Post_{s_0}^*[w_1](v_1)$ such that $w_1w_2 = w$, $u \in Post_{s_0}^*[w_2](v_2)$, and $u' \sim_0 0$. We proceed by structural induction on $v$. Let $\cdot(q_1, q_2) \rightarrow q$ be a rule of $\Delta$ such that $v_1 \in L_{q_1}$ and $v_2 \in L_{q_2}$. By structural induction, we have that $w_1 \in e(q_1, s_0)$ and $w_2 \in e(q_2, s)$, $(\beta_3)$ implies that $w = w_1w_2 \in e(q, s)$.

     (c) The case where $v = ||(v_1, v_2)$ is symmetrical. In this case, we need the rules $(\beta_7)$.

  2. The root of $v$ was not rewritten. We proceed by structural induction on $v$:

     * $v = \cdot(v_1, v_2)$ 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 $\cdot(q_1, q_2) \rightarrow q$ is a rule of $\Delta$ and $\cdot(s_1, s_2) \rightarrow s$ is a rule of $\Delta'$ s.t.:

       (a) $u_1$ is equivalent to 0, i.e., $u_1 \in L_{s_{null}}$. Let then $w_1, w_2$ s.t. $w = w_1 \cdot w_2$, $u_1 \in Post_{s_0}^*[w_1](v_1)$, and $u_2 \in Post_{s_0}^*[w_2](v_2)$. By structural induction, we have $w_1 \in e(q_1, s_{null}^1)$ and $w_2 \in e(q_2, s_2)$. We obtain then by $(\beta_4)$ that

       $$w = w_1 \cdot w_2 \in e(q_1, s_{null}^1) \cdot e(q_2, s_2) \subseteq e(q, s).$$
(b) \( u_1 \) is not equivalent to 0, in this case \( u_2 \sim_0 v_2 \in L_{s_2} \cap \{L_{q_2}\} \sim_0 \) and \( u_1 \in Post^*_{\sim_0}[w](v_1) \). By structural induction, we obtain \( w \in e(q_1, s_1) \), and by \((\beta_3)\), we obtain that

\[
w \in e(q_1, s_1) \subseteq e(q, s).
\]

* \( v = ||(v_1, v_2) \) and \( u = ||(u_1, u_2) \). Lemma 4.1 infers that there exists \( w_1, w_2 \) s.t. \( w \in w_1 \parallel w_2, u_1 \in Post^*_{\sim_0}[w_1](v_1), \) and \( u_2 \in Post^*_{\sim_0}[w_2](v_2) \). Let \( q_1, q_2 \in \mathcal{Q} \) s.t. \(||(q_1, q_2) \rightarrow q \) is a rule of \( \Delta \), \( v_1 \in L_{q_1} \), and \( v_2 \in L_{q_2} \). Moreover, let \( s_1, s_2 \in \mathcal{Q}' \) s.t. \(||(s_1, s_2) \rightarrow s \) is a rule of \( \Delta' \) s.t. \( u_1 \in L_{s_1} \) and \( u_2 \in L_{s_2} \). Then, by structural induction we infer that \( w_1 \in e(q_1, s_1) \) and \( w_2 \in e(q_2, s_2) \), and thanks to \((\beta_4)\), it follows that

\[
w \in w_1 \parallel w_2 \subseteq e(q_1, s_1) \parallel e(q_2, s_2) \subseteq e(q, s).
\]

Consider the other direction. Let \( w \in e(q, s) \), we show that \( Post^*_{\sim_0}[w](L_q) \cap L_s \neq \emptyset \). Since the labels \( e(q, s) \) are built using a saturation procedure, we consider the sequences \( (e_i(q, s))_{0 \leq i \leq n} \) where \( e_0(q, s) = 0, e_n(q, s) = e(q, s) \), and \( e_i(q, s) \) is the label obtained after the \( i \)th iteration. We prove by induction on \( i \) that if \( w \in e_i(q, s) \), then \( Post^*_{\sim_0}[w](L_q) \cap L_s \neq \emptyset \). The case where \( i = 0 \) is direct. The same for the case where \( i = 1 \), since in this case, it’s the rule \( \beta_1 \) which is applied. Let then \( i > 1 \). Let \( w \in e_i(q, s) \), then :

- Either there exists \( (q', s') \) s.t. \( w \in e_{i-1}(q', s') \), and \( e_{i-1}(q', s') \subseteq e_i(q, s) \) (rules \( \beta_2 \)). In this case, we have necessarily \( q' \rightarrow q \in \Delta \) and \( s' \rightarrow s \in \Delta' \), and hence \( L_{q'} \subseteq L_q \), and \( L_{s'} \subseteq L_s \). By induction, we have

\[
Post^*_{\sim_0}[w](L_{q'}) \cap L_{s'} \neq \emptyset.
\]

It follows that \( Post^*_{\sim_0}[w](L_q) \cap L_s \neq \emptyset \) 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. \( (q_1, q_2) \rightarrow q \in \Delta \), \( (s_1, s_2) \rightarrow s \in \Delta' \), \( w_1 \in e_{i-1}(q_1, s_1^\text{null}) \), \( w_2 \in e_{i-1}(q_2, s_2) \), \( w \in e_i(q, s) \) (rules \( \beta_3 \)). By induction, we obtain that

\[
Post^*_{\sim_0}[w_1](L_{q_1}) \cap L_{s_1^{\text{null}}} \neq \emptyset
\]

and

\[
Post^*_{\sim_0}[w_2](L_{q_2}) \cap L_{s_2} \neq \emptyset.
\]

Let then \( u_1 \) and \( u_2 \) s.t.

\[
u_1 \in Post^*_{\sim_0}[w_1](L_{q_1}) \cap L_{s_1^{\text{null}}}
\]

and
\[ u_2 \in Post^*_\omega [w_2](L_{q_2}) \cap L_{s_2}. \]

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

\[ \cdot (u_1, u_2) \in Post^*_\omega [w](L_q) \cap L_s. \]

- Either there exist \( q_1, s_1, q_2, s_2 \) s.t. \( \cdot (q_1, q_2) \rightarrow q \in \Delta, \cdot (s_1, s_2) \rightarrow s \in \Delta', w \in e_{i-1}(q_1, s_1), \) and \([L_{q_2}]_\omega \cap L_{s_2} \neq \emptyset \) (rules \( \beta_3 \)). Let then \( u_2 \in [L_{q_2}]_\omega \cap L_{s_2}. \) Since \( w \in e_{i-1}(q_1, s_1) \), it follows by induction that there exists \( u_1 \in Post^*_\omega [w](L_{q_1}) \cap L_{s_1}. \) It is then clear that

\[ \cdot (u_1, u_2) \in Post^*_\omega [w](L_q) \cap L_s. \]

- Either there exist \( w_1, w_2 \) s.t. \( w \in w_1 \sqcup w_2, \) and there exist \( q_1, s_1, q_2, s_2 \) s.t. \( ||(q_1, q_2) \rightarrow q \in \Delta, \|| (s_1, s_2) \rightarrow s \in \Delta', w_1 \in e_{i-1}(q_1, s_1), \) and \( w_2 \in e_{i-1}(q_2, s_2) \) (rules \( \beta_4 \)). By induction, we obtain that

\[ Post^*_\omega [w_1](L_{q_1}) \cap L_{s_1} \neq \emptyset \]

and

\[ Post^*_\omega [w_2](L_{q_2}) \cap L_{s_2} \neq \emptyset. \]

Let then \( u_1 \) and \( u_2 \) s.t. \( u_1 \in Post^*_\omega [w_1](L_{q_1}) \cap L_{s_1} \) and \( u_2 \in Post^*_\omega [w_2](L_{q_2}) \cap L_{s_2}. \) We obtain from Lemma 4.1 that

\[ || (u_1, u_2) \in Post^*_\omega [w](L_q) \cap L_s. \]

- Either there exist a rule \( t_1 \xleftarrow{a} t_2 \in R, w_1, w_2 \) s.t. \( w = w_1aw_2, w_1 \in e_{i-1}(q, q_1), \) \( w_2 \in e_{i-1}(q_2, s) \) (rules \( \beta_5 \)). By induction, we obtain that

\[ t_1 \in Post^*_\omega [w_1](L_q) \]

and there exists \( u \in L_s \) s.t.

\[ u \in Post^*_\omega [w_2](t_2). \]

It is then clear that

\[ u \in Post^*_\omega [w_1aw_2](L_q). \]

- Either there exist \( w_1, w_2 \) s.t. \( w = w_1w_2, \) and there exist \( q_1, q_2 \) s.t. \( \cdot (q_1, q_2) \rightarrow q \in \Delta, w_1 \in e_{i-1}(q_1, s_0), w_2 \in e_{i-1}(q_2, s), w \in e_i(q, s) \) (rules \( \beta_6 \)). By induction, we obtain that

\[ Post^*_\omega [w_1](L_{q_1}) \cap L_{s_0} \neq \emptyset \]

and
Let then \( u_1 \) and \( u_2 \) s.t.

\[ u_1 \in \text{Post}^*_{\sim_0} [w_1](L_{q_1}) \cap L_{s_0} \]

and

\[ u_2 \in \text{Post}^*_{\sim_0} [w_2](L_{q_2}) \cap L_s. \]

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

\[ (u_1, u_2) \in \text{Post}^*_{\sim_0} [w](L_q) \cap L_s \]

and since \( u_1 \sim_0 0 \), that

\[ u_2 \in \text{Post}^*_{\sim_0} [w](L_q) \cap L_s \]

- Either there exist \( w_1, w_2 \) s.t. \( w \in w_1 \) or \( w_2 \), and there exist \( q_1, q_2 \) s.t. \( ||(q_1, q_2) \rightarrow q \in \Delta, w_1 \in e_{i-1}(q_1, s_0), w_2 \in e_{i-1}(q_2, s), w \in e_i(q, s) \) (rules \( \beta_7 \)). This case is similar to the previous one.