Reachability analysis of multithreaded software with asynchronous communication

Ahmed Bouajjani[1], Javier Esparza[2], Stefan Schwoon[2], and Jan Strejček[2]

[1] LIAFA, University of Paris 7, abou@liafa.jussieu.fr
[2] Institute for Formal Methods in Computer Science, University of Stuttgart
{esparza,schwoosn,strejcek}@informatik.uni-stuttgart.de

Abstract. We introduce *asynchronous dynamic pushdown networks* (ADPN), a new model for multithreaded programs in which pushdown systems communicate via shared memory. ADPN generalizes both CPS (concurrent pushdown systems) [QR05] and DPN (dynamic pushdown networks) [BMOT05]. We show that ADPN exhibit several advantages as a program model. Since the reachability problem for ADPN is undecidable even in the case without dynamic creation of processes, we address the *bounded* reachability problem [QR05], which considers only those computation sequences where the (index of the) thread accessing the shared memory is changed at most a fixed given number of times. We provide efficient algorithms for both forward and backward reachability analysis. The algorithms are based on automata techniques for symbolic representation of sets of configurations.

1 Introduction

In recent years a number of formalisms have been proposed for modelling and analyzing procedural multithreaded programs. A well-known result states that, if recursion is allowed, checking assertions for these programs is undecidable, even if all variables are boolean (see for instance [Ram00]).

Due to this undecidability result, approximate analysis techniques have been considered. In [BET03,BET04] it is shown how to compute overapproximations of the set of reachable states. In [QR05], Qadeer and Rehof present the first nontrivial technique to compute underapproximations. In this paper we build on the ideas of [QR05], which we now describe in some more detail. Qadeer and Rehof introduce *concurrent pushdown systems* (CPS) as a model of multithreaded programs. In a nutshell, a CPS is a set of stacks with a global finite control; at each step, the CPS reads the current control state and the topmost symbol of (exactly) one of the stacks, can change the control state and replace the stack symbol by a word, like in a pushdown automaton. A *dynamic* CPS (or DCPS) can also, optionally, create a new stack as the result of a transition. Each stack of a CPS corresponds to a thread. Communication between threads is modelled through the common set of global control states. A *context* is defined as a computation in which all transitions act *on the same stack*. In [QR05] it is shown how to compute, given a fixed number k, the set of states that can be reached by *k-bounded* computations, i.e., by computations consisting of the concatenation of at most k contexts. Obviously, this set constitutes an underapproximation of the set of all reachable states.

In this paper, we show that with the help of a refined model it is possible to generalize and improve the results of [QR05] in a number of ways. We propose a generalization of CPS called *asynchronous pushdown networks* (APN); we also introduce the dynamic version of the model, called ADPN. Loosely speaking, the stacks of an APN have an additional set of local control states, different from the common global finite control; transitions are either local (dependent only on the local control), or global (depending on both the global and local control states). We also propose a new, more liberal, definition of context: a context is now a computation in which all *global* transitions act on the same stack, possibly interspersed with local transitions acting on arbitrary stacks.

In the first part of the paper (Section 2) we observe that, while the APN and CPS formalisms are equally expressive, APN can model programs more succinctly than CPS. In the dynamic case we show that, while ADPN can naturally model value passing from a called procedure to its caller, DCPS cannot.

In the second part of the paper (Section 3), we study the forward and backward k-bounded reachability problem for APN. Comparing [QR05], we propose a more general and asymptotically faster algorithm for forward reachability. We introduce a backward reachability algorithm as well.

In the third part of the paper (Sections 4 and 5), we consider the k-reachability problem for the ADPN model. We show that, due to the more liberal notion of context, the set of configurations of an ADPN reachable by k-bounded computations may be non-regular, contrary to the case of DCPSs. Using results of [BMOT05], we show that the set is always context-free and provide an algorithm to compute a context-free grammar that generates it. We then observe that the set of backwards k-bounded reachable configurations is regular, and, relying on results from [EHRS00], provide an efficient algorithm to compute it.

2 The model

2.1 Asynchronous dynamic pushdown networks

An *asynchronous dynamic pushdown network (ADPN)* is a tuple $\mathcal{N} = (G, P, \Gamma, \Delta_l, \Delta_g)$, where G is a finite set of *global states*, P is a finite set of *local states*, Γ is a finite *stack alphabet*, and

- Δ_l is a finite set of *local rules* of the form $p\gamma \hookrightarrow p_1 w_1$ or $p\gamma \hookrightarrow p_1 w_1 \rhd p_2 w_2$, where $p, p_1, p_2 \in P$, $\gamma \in \Gamma$, and $w_1, w_2 \in \Gamma^*$.
- Δ_g is a finite set of *global rules* of the form $(g, p\gamma) \hookrightarrow (g', p_1 w_1)$ or $(g, p\gamma) \hookrightarrow (g', p_1 w_1) \rhd p_2 w_2$, where $g, g' \in G$, $p, p_1, p_2 \in P$, $\gamma \in \Gamma$, and $w_1, w_2 \in \Gamma^*$.

The rules with a suffix of the form $\rhd p_2 w_2$ are called *dynamic*. A *configuration* of an ADPN is a pair $(g, \alpha) \in G \times (P\Gamma^*)^+$ of a global state g and a word $\alpha = p_1 w_1 p_2 w_2 \dots p_n w_n$, where each subword $p_i w_i \in P\Gamma^*$ represents a configuration of (a pushdown corresponding to) one *component*. A word $p_i w_i$ is called *component configuration*. The set of all configurations is denoted by $\mathcal{C}$.

The transition relation $\to \subseteq \mathcal{C} \times \mathcal{C}$ is defined as follows: $(g, u) \to (g', v)$ if there is

- $p\gamma \hookrightarrow p_1 w_1$ in Δ_l such that $u = u_1 p\gamma u_2$, $v = u_1 p_1 w_1 u_2$, and $g = g'$, or

- $p\gamma \hookrightarrow p_1 w_1 \triangleright p_2 w_2$ in Δ_l such that $u = u_1 p\gamma u_2$, $v = u_1 p_2 w_2 p_1 w_1 u_2$, and $g = g'$, or
- $(g, p\gamma) \hookrightarrow (g', p_1 w_1)$ in Δ_g such that $u = u_1 p\gamma u_2$ and $v = u_1 p_1 w_1 u_2$, or
- $(g, p\gamma) \hookrightarrow (g', p_1 w_1) \triangleright p_2 w_2$ in Δ_g such that $u = u_1 p\gamma u_2$ and $v = u_1 p_2 w_2 p_1 w_1 u_2$,

where $u_1 \in (P\Gamma^*)^*$ and $u_2 \in \Gamma^*(P\Gamma^*)^*$. We say that the transition has been performed by the component whose local state changes from p to p_1. The transitions generated by global and local rules are called *global* and *local transitions* respectively. A dynamic rule creates a new component starting in component configuration $p_2 w_2$.

2.2 Subclasses of ADPNs

ADPNs are an extension of several other models. An ADPN with only global states and global rules is a *dynamic concurrent pushdown systems* (DCPS). Formally, a DCPS is an ADPN $(G, P, \Gamma, \Delta_l, \Delta_g)$ satisfying $|P| = 1$ and $\Delta_l = \emptyset$. The DCPS model is studied in [QR05]. The subclasses of ADPN and DCPS without dynamic rules are called APN and CPS, respectively. Notice that in an APN or CPS all configurations reachable from an initial configuration have the same number of components. Finally, both APNs and CPSs are extensions of pushdown systems (PDS). Formally, a PDS is a CPS in which the initial configuration only has one component.

An ADPN without global variables or global rules is called a DPN. DPNs have been introduced and studied in [BMOT05]. Notice that in a DPN there is no communication between different threads.

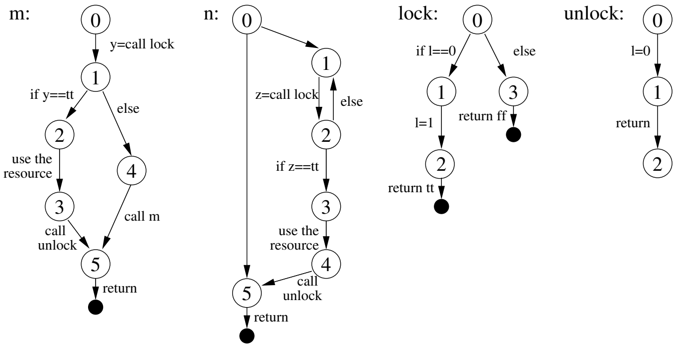
2.3 Reachability and bounded reachability

Given an ADPN $\mathcal{N}$ and a set $S \subseteq \mathcal{C}$, we denote by $post^*_{\mathcal{N}}(S)$ and $pre^*_{\mathcal{N}}(S)$ the sets of forward and backward reachable configurations from S. The *forward* and *backward reachability problem* consists of, given sets I and F of initial and final configurations, determining if $post^*_{\mathcal{N}}(I) \cap F = \emptyset$ or $pre^*_{\mathcal{N}}(F) \cap I = \emptyset$, respectively. Both problems are undecidable, even when I and F are singletons. This is a consequence of the fact that APNs (even without dynamic rules) are Turing powerful. For instance, it is straightforward to encode a 2-counter Minsky machine into an APN.

Following [QR05], we define a notion of bounded reachability. A *context* is a transition sequence where all global transitions are performed by the same component. We say that this component *controls* the context. Notice that within a context local transitions can be performed by arbitrary components. For $k \geq 1$, a sequence of transitions is *k-bounded* if it is a concatenation of at most k contexts. We denote by $post^*_{k,\mathcal{N}}(S)$ the set of all configurations reachable from S by k-bounded sequences. By analogy, $pre^*_{k,\mathcal{N}}(S)$ denotes the set of all configurations from which a configuration from S is reachable by a k-bounded sequence. We talk about *forward* and *backward k-bounded reachability*, respectively. Further, by $post^*_{0,\mathcal{N}}(S)$ and $pre^*_{0,\mathcal{N}}(S)$ we denote the sets of configurations that are forward and backward reachable only by local transitions, respectively.

2.4 APN as program model

The following example illustrates how to model programs with APNs (for simplicity, we omit thread creation here). We consider a program with procedures $m, n, lock, unlock$



$\bullet[m_0,b] \hookrightarrow \bullet lock_0[m_1,b]$	$\bullet[n_0,b] \hookrightarrow \bullet[n_1,b]$	$\blacksquare$ $(0,\bullet lock_0) \hookrightarrow (0,\bullet lock_1)$
$tt[m_1,b] \hookrightarrow \bullet[m_2,tt]$	$\bullet[n_0,b] \hookrightarrow \bullet[n_5,b]$	$\blacksquare$ $(1,\bullet lock_0) \hookrightarrow (1,\bullet lock_3)$
$ff[m_1,b] \hookrightarrow \bullet[m_4,ff]$	$\bullet[n_1,b] \hookrightarrow \bullet lock_0[n_2,b]$	$\blacksquare$ $(0,\bullet lock_1) \hookrightarrow (1,\bullet lock_2)$
$\bullet[m_2,b] \hookrightarrow \bullet[m_3,b]$	$tt[n_2,b] \hookrightarrow \bullet[n_3,tt]$	$\blacksquare$ $(1,\bullet lock_1) \hookrightarrow (1,\bullet lock_2)$
$\bullet[m_3,b] \hookrightarrow \bullet unlock_0[m_5,b]$	$ff[n_2,b] \hookrightarrow \bullet[n_1,ff]$	$\bullet lock_2 \hookrightarrow tt\varepsilon$
$\bullet[m_4,b] \hookrightarrow \bullet[m_0,-][m_5,b]$	$\bullet[n_3,b] \hookrightarrow \bullet[n_4,b]$	$\bullet lock_3 \hookrightarrow ff\varepsilon$
$\bullet[m_5,b] \hookrightarrow \bullet\varepsilon$	$\bullet[n_4,b] \hookrightarrow \bullet unlock_0[n_5,b]$	$\blacksquare$ $(0,\bullet unlock_0) \hookrightarrow (0,\bullet unlock_1)$
	$\bullet[n_5,b] \hookrightarrow \bullet\varepsilon$	$\blacksquare$ $(1,\bullet unlock_0) \hookrightarrow (0,\bullet unlock_1)$
where b ranges over $\{-,tt,ff\}$		$\bullet unlock_1 \hookrightarrow \bullet\varepsilon$

Fig. 1. A program with four procedures and two threads.

described by the flow graphs of Figure 1; y and z are local variables of the procedures m and n, respectively, and can take the values undefined ($-$), true (tt), or false (ff). The procedures m and n call procedures *lock* and *unlock* to get exclusive access to a shared resource. The *lock* action is nonblocking; it returns true if it succeeds to lock the resource, false otherwise. The variable l occurring in the procedures *lock* and *unlock* is global and ranges over $\{0,1\}$. The system consists of two concurrent threads, one starting with the execution of m, the other with the execution of n.

We model this program by the APN $\mathcal{N} = (G,P,\Gamma,\Delta_l,\Delta_g)$ as follows: Global states model the value of the global variable l, i.e. $G = \{0,1\}$. Local states are used to pass a potential return value from a callee back to the caller: The callee stores the value in the local state of the thread, from where it is read by the caller.[3] As a procedure cannot return the undefined value ($-$), we set $P = \{tt,ff,\bullet\}$, where tt and ff are used to return the corresponding values, and $\bullet$ is used elsewhere. The set Γ of stack symbols contains all program locations (p_l denotes the symbol for location l of procedure p), together with the actual values of the local variables for procedures m,n. The local and global rules corresponding to each procedure are given directly in the figure; global rules (marked with $\blacksquare$) correspond to transitions dealing with the global variable l.

The techniques developed in the next sections can show that the program does not satisfy its basic specification: exclusive access to the resource. More precisely, they

[3] In general, local states can be also used to hold values of variables that are global to a thread (if such a variable type is supported in the modeled system).

show that the program can reach a configuration of the form $(0, \bullet[m_2, b]w_1 \bullet [n3, b']w_2)$ from the initial configuration $(0, \bullet[m_0, -] \bullet [n_0, -])$, and in fact within 3 contexts.

2.5 A(D)PN versus (D)CPS

As we have seen, local states are used to model value-passing from a callee to its caller. In the CPS model there is no notion of local state of a thread, and so value passing must be simulated through a global variable. Clearly, this amounts to simulating an APN by a CPS. We show that this is possible, but involves a blow-up in size. Moreover, the translation has to fix the number n of components that the CPS can work upon. Let $\mathcal{N} = (G, P, \Gamma, \Delta_l, \Delta_g)$ be an APN. We construct a CPS $\mathcal{N}' = (G', \Gamma', \Delta'_g)$ such that the configuration graphs of $\mathcal{N}$ and $\mathcal{N}'$, defined in the usual way, are isomorphic. We take $G' = G \times P^n$, $\Gamma' = \Gamma \times \{1, \ldots, n\}$, and add to Δ'_g rules

$$((g_1, p_1, \ldots, p_{i-1}, p, p_{i+1}, \ldots, p_n), q(\gamma, i)) \hookrightarrow ((g_2, p_1, \ldots, p_{i-1}, p', p_{i+1}, \ldots, p_n), q[w, i])$$

for every $(g_1, p\gamma) \hookrightarrow (g_2, p'w)$ in Δ_g, $1 \leq i \leq n$, $p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n \in P$, and rules

$$((g, p_1, \ldots, p_{i-1}, p, p_{i+1}, \ldots, p_n), q(\gamma, i)) \hookrightarrow ((g, p_1, \ldots, p_{i-1}, p', p_{i+1}, \ldots, p_n), q[w, i])$$

for every $p\gamma \hookrightarrow p'w$ in Δ_l, $g \in G$, $1 \leq i \leq n$, and $p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n \in P$. Here, q is the only local state of $\mathcal{N}'$. Further, for $w = w_1 w_2 \ldots w_m$, $[w, i]$ stands for $(w_1, i)(w_2, i) \ldots (w_m, i)$. Observe that the size of $\mathcal{N}'$ may be larger than that of $\mathcal{N}$ by a factor of $n \cdot |G| \cdot |P|^{n-1}$.

Observe also that the transformation APN $\rightarrow$ CPS cannot be naturally extended to a transformation ADPN $\rightarrow$ DCPS. The straightforward idea of taking $G \times P^*$ as set of global states does not work, and not only because this set is infinite, but also because in order to simulate a change of local state a stack has to know its position in the current state $(g, p_1 p_2 \ldots p_n)$, which now changes as the computation proceeds because of thread creation. Currently we do not know if an ADPN can be translated into an equivalent DCPS, and we do not see any elegant way of modelling value-passing and thread creation in the DCPS formalism.

We finish with an advantage of our more liberal notion of context. In a k-bounded computation, at most k components can execute global transitions, and this has the following consequence when comparing ADPN and DCPS: While a k-bounded computation of a DCPS can create an arbitrary number of components, at most k of them can execute a transition at all. For ADPN the constraint is weaker: arbitrarily many processes can execute transitions, but at most k of them can execute global transitions. So an algorithm for exploring k-bounded computations of ADPN searches 'deeper' as the same algorithm for DCPS.

3 Reachability analysis for APN

We now consider k-bounded reachability for the APN model, i.e. the restriction of ADPN to non-dynamic rules. Let us fix an APN $\mathcal{N} = (G, P, \Gamma, \Delta_l, \Delta_g)$ and $k \in \mathbb{N}$ for the rest of this section. We investigate the case where the initial or final configurations are given by so-called aggregates:

Definition 1. *An* aggregate *is a tuple $M = (g, C_1, \ldots, C_n)$, where $g \in G$, $n \geq 1$ is the number of concurrent processes, and $C_1, \ldots, C_n \subseteq P \times \Gamma^*$ are* regular *sets of component configurations. M is used to denote the set $\{g\} \times (C_1. \cdots .C_n)$, where . is the concatenation of the component configurations.*

We now fix an aggregate $M = (g, C_1, \ldots, C_n)$ for the rest of the section, and we will present solutions for computing $post^*_{k,\mathcal{N}}(M)$ as well as $pre^*_{k,\mathcal{N}}(M)$.

For the CPS model, k-bounded reachability was considered in [QR05]. The algorithms presented in this section follow the same general idea as the solutions in [QR05] (but applied to APN). Moreover, the new solution has these benefits:

- Our algorithm avoids repeating partial computations of reachable component configurations. Even if we consider only CPSs, the algorithm runs asymptotically faster than the one presented in [QR05].
- The APN model distinguishes between local and global states, and our algorithm exploits this difference. Therefore, it is faster than a translation of a given APN to CPS (see Section 2.5) followed by the application of an algorithm for CPS.
- Some details in our algorithm are different from [QR05] and would lead to time and memory savings in an implementation. These are discussed in Section 3.3.
- We provide algorithms for both forward and backwards reachability, whereas [QR05] only covered forward reachability. The two algorithms are fairly similar – in fact we will present them as one algorithm – but their complexity analysis is a little more involved. The algorithm makes use of a procedure called CLOSURE, which stands for the *post** or *pre** procedure on PDSs [EHRS00] in case of forward and backwards reachability, respectively.

3.1 Reordering of Transitions

Our algorithms are based on the following observation: Let c be a configuration reachable from $M = (g, C_1, \ldots, C_n)$ by a k-bounded computation, and let σ be this computation. Then the transitions in σ can be rearranged to another k-bounded computation σ' that also leads from M to c. Moreover, σ' can be partitioned into $n+k$ phases, where in each phase all rules are applied to the same component:

- In the i-th phase, $1 \leq i \leq n$, component i executes all its *local* steps in σ up to, but not including, its first global step (or all steps, if it never executes a global rule).
- In the $n+i$-th phase, $1 \leq i \leq k$, the component controlling the i-th context executes the first *global* step of the i-th context in σ, followed by all its *global* and *local* steps up to, but not including, the first global step in the next context controlled by the same component (all its remaining steps, if it does not control any more contexts).

Notice that this rearrangement only requires to swap the ordering of local transitions of some component with local or global transitions of other components; but as the application of a local rule does not depend on the global state, these reorderings do not alter the final configuration of the computation.

3.2 Reduction to PDS

We now show that all $n+k$ phases reduce to reachability problems on PDS. In the following, $\textsc{Closure}_{\mathcal{P}}(C)$ denotes the set $post^*_{\mathcal{P}}(C)$ or $pre^*_{\mathcal{P}}(C)$, depending on whether forward or backward reachability is of interest.

- Let $\mathcal{P}^1_{\mathcal{N}} := (P, \Gamma, \Delta_l)$, i.e. $\mathcal{P}^1_{\mathcal{N}}$ simulates the local moves of $\mathcal{N}$. Thus, the results of the first n phases are obtained by $\textsc{Closure}_{\mathcal{P}^1_{\mathcal{N}}}(C_i)$ for $i = 1, \ldots, n$.
- For the remaining phases, we create a PDS in which the global and local states are merged. Let $\mathcal{P}^2_{\mathcal{N}} = (G \times P, \Gamma, \Delta')$, where Δ' contains all $(g_1, p_1)\gamma \hookrightarrow (g_2, p_2)w$ such that either $(g_1, p_1\gamma) \hookrightarrow (g_2, p_2 w)$ in Δ_g, or $p_1\gamma \hookrightarrow p_2 w$ in Δ_l and $g_1 = g_2$. Thus, $\mathcal{P}^2_{\mathcal{N}}$ computes the possible operations of one component in a single context. More precisely, we define $\textsc{Lift}(g, C) := \{\, ((g, p), w) \mid (p, w) \in C \,\}$ and $\textsc{Restrict}(C, g) := \{\, (p, w) \mid ((g, p), w) \in C \,\}$. Now, if a component starts a context in global state g and with component configurations C, the reachable configurations within this context that end in global state g' are $\textsc{Restrict}(\textsc{Closure}_{\mathcal{P}^2_{\mathcal{N}}}(\textsc{Lift}(g, C)), g')$.

Recall that the initial sets $C_1, \ldots, C_n$ are regular and can be represented by finite automata. Regular sets are closed under the CLOSURE operation, and algorithms for these have been provided in [EHRS00]. It is easy to see that LIFT and RESTRICT can also be implemented as operations on finite automata.

3.3 The algorithm

Figure 2 shows our algorithm, which directly implements the ideas outlined before. Line 2 computes the local phases $1, \ldots, n$ of the computations, whereas the lines from line 3 onwards implement phases $n+1, \ldots, n+k$. Essentially, the algorithm explores a 'tree' of depth k, where each node corresponds to an aggregate, and its successors are the aggregates reachable by executing one context. Each iteration of the while loop picks an aggregate and computes its successors. As hinted at before, the operations on the sets of component configurations are carried out by operations on finite automata. The algorithm uses the following data structures:

todo is a list with information on those aggregates whose successors still need to be computed. The first part of each entry in *todo* indicates the depth of the aggregate in the tree, the second is the index of the component that has controlled the previous context; the rest is the aggregate itself.

aut is a hash table. An entry $aut[g, B]$ remembers the result of applying the closure on $\textsc{Lift}(g, B)$. The motivation for this table is that, for a pair (g, B), the computation of $\textsc{Closure}_{\mathcal{P}^2_{\mathcal{N}}}(\textsc{Lift}(g, B))$ may be required in multiple branches of the 'tree'; therefore we would like to reuse the result. Notice that actually hashing over (an automaton accepting) the language B could be very time consuming. In order to achieve the desired time-saving effect (see Subsection 3.4), it suffices to approximate this effect, e.g. by giving a unique identifier to each automaton that arises from an application of CLOSURE.

reachable collects the aggregates that represent reachable configurations.

Input: An APN $\mathcal{N}$, an aggregate $M = (g, C_1, \ldots, C_n)$, and $k \in \mathbb{N}$.
Output: The set $post^*_{k,\mathcal{N}}(M)$ (or $pre^*_{k,\mathcal{N}}(M)$) given by union of the aggregates in *reachable*.

1 $\mathit{reachable} \leftarrow \emptyset$;
2 $\mathit{todo} \leftarrow \{(0, 0, g, \textsc{Closure}_{\mathcal{P}^1_{\mathcal{N}}}(C_1), \ldots, \textsc{Closure}_{\mathcal{P}^1_{\mathcal{N}}}(C_n))\}$;
3 **while** $\mathit{todo} \neq \emptyset$ **do**
4   pop $(\mathit{level}, \mathit{last}, g, B_1, \ldots, B_n)$ with minimal *level* from *todo*;
5   **if** $\mathit{level} = k$ **then**
6     $\mathit{reachable} \leftarrow \mathit{reachable} \cup \{(g, B_1, \ldots, B_n)\}$;
7   **else**
8   **for all** $i = 1, \ldots, n$ such that $i \neq \mathit{last}$ **do**
9   **if** $\mathit{aut}[g, B_i]$ undefined **then**
10         $\mathit{aut}[g, B_i] \leftarrow \textsc{Closure}_{\mathcal{P}^2_{\mathcal{N}}}(\textsc{Lift}(g, B_i))$;
11   **for all** $g' \in G$ **do**
12         $\mathit{todo} \leftarrow \mathit{todo} \cup \{(\mathit{level}+1, i, g', B_1, \ldots, B_{i-1}, \textsc{Restrict}(\mathit{aut}[g, B_i], g'), B_{i+1}, \ldots, B_n)\}$;

Fig. 2. Algorithm computing k-bounded reachability on APN.

The basic idea of exploring a tree of depth k is similar to the CPS algorithm in [QR05]. However, the algorithm in Figure 2 also contains some improvements:

– When adding a new item to *todo*, the algorithm reuses all previous local automata except for B_i (unlike [QR05], where all n automata are changed in every step). This makes the algorithm more memory-efficient, because the automata that have not changed from one context to another can be shared.
– Using *aut* allows to reuse results of computations made in other parts of the tree.
– A trivial improvement is that no component is allowed to execute two contexts in a row (the second context would yield nothing new due to closure properties).
– Another simple, but important optimization (not shown) is that line 11 should only be executed for those global states g' such that $\mathit{aut}[g, B_i]$ accepts at least one configuration of the form $\langle g', w \rangle$ for some $w \in \Gamma^*$.

3.4 Complexity analysis

We now examine the complexity of our algorithm for both directions. Recall that the set-theoretic operations in Figure 2 are implemented using operations on finite automata. We first state some facts about these:

Definition 2 ([EHRS00]). *Let $\mathcal{P} = (P', \Gamma', \Delta)$ be a pushdown system. A quintuple $A = (Q, \Gamma', \delta, P', F)$ is called a $\mathcal{P}$-*automaton *if A is a finite automaton where the set of states Q subsumes P', and where the elements of P' are the initial states of the automaton. We say that A* accepts *a set $C \subseteq P' \times \Gamma^*$, where C consists of the pairs (p, w) such that w is accepted in A by a path starting at p.*

Notice that we can equivalently represent $M = (g, C_1, \ldots, C_n)$ by a tuple $(g, A_1, \ldots, A_n)$, where all A_i ($i = 1, \ldots, n$) represent the configurations of a single component. In other words, A_i is a $\mathcal{P}^1_{\mathcal{N}}$-automaton accepting C_i, for $i = 1, \ldots, n$.

All operations required by the algorithm can be efficiently implemented on regular sets of languages using automata. Using the algorithms from [EHRS00], automata accepting the sets $post^*_{\mathcal{P}}(L(A))$ and $pre^*_{\mathcal{P}}(L(A))$ can be efficiently computed, with the following complexities:

Theorem 1 ([EHRS00]). *Let $\mathcal{P} = (P', \Gamma, \Delta)$ be a PDS and $A = (Q, \Gamma, \delta, P', F)$ be a $\mathcal{P}$-automaton.*

(a) An automaton accepting $post^*_{\mathcal{P}}(L(A))$ can be constructed in time $O(|P'| \cdot |\Delta| \cdot (|Q_0| + |\Delta|) + |P'| \cdot |\delta_0|)$, where $Q_0 = Q \setminus P'$ and $\delta_0 \subseteq \delta$ is the set of all transitions leading from states in Q_0. Moreover, the automaton has at most $|Q| + |\Delta|$ states and at most $|\delta_0| + |\Delta| \cdot (|\Delta| + |Q_0|)$ transitions leading from states that are not in P'.

(b) An automaton accepting $pre^*_{\mathcal{P}}(L(A))$ can be constructed in time $O(|Q|^2|\Delta|)$ and with $|Q|$ states.

Moreover, we need the following operations:

- Given a $\mathcal{P}^1_{\mathcal{N}}$-automaton A, we can generate a $\mathcal{P}^2_{\mathcal{N}}$-automaton accepting the set $\textsc{Lift}(g, A)$ by modifying A as follows: Change the name of every state $p \in P$ to a pair (g, p) and add initial states $(g', p) \in (G \setminus \{g\}) \times P$.
- Given a $\mathcal{P}^2_{\mathcal{N}}$-automaton A, we can create a $\mathcal{P}^1_{\mathcal{N}}$-automaton accepting $\textsc{Restrict}(A, g)$ by making all states in $(G \setminus \{g\}) \times P$ non-initial and renaming every state of the form (g, p) to p.

Looking at the algorithm in Figure 2, it is straightforward to see that the bulk of the work is done in the cycle starting at line 3. The number of iterations of this cycle equals the number of different tuples in *todo*. Let t_j denotes the number of these tuples with j as the first component. Then $t_0 = 1$, $t_j \leq t_{j-1} \cdot |G| \cdot (n-1)$, or $t_j \leq (n-1)^j \cdot |G|^j$, for $j \geq 1$. Line 6 is thus executed $t_k = (n-1)^k \cdot |G|^k$ times, while the number of executions of the **else** branch starting at line 7 is

$$t = \sum_{j=0}^{k-1} t_j \;\leq\; \frac{(n-1)^k \cdot |G|^k - 1}{(n-1) \cdot |G| - 1} \;\in\; O(n^{k-1} \cdot |G|^{k-1}).$$

Let us compute the number of times line 10 is executed. Thanks to the *aut* structure, this number is less than t.

Lemma 1. *Let $S_{lev,j}$ be the set of pairs (g, B_j) such that some element of the form $(lev, last, g, B_1, \ldots, B_j, \ldots, B_n)$ is added to todo during the algorithm run. Then the number of distinct pairs in $S_{lev,j}$, denoted $|S_{lev,j}|$, is $O(|G|^{lev})$.*

We prove Lemma 1 by induction on *lev*.

Base $lev = 0$. Obvious, because there is exactly one tuple (produced in line 2) with $lev = 0$.

Induction step. Assume $|S_{lev,j}| = O(|G|^{lev})$ holds. Let us see what happens in lines 8 to 12. For all executions with $i \neq j$, the additions in line 12 only change the global states, but not the automaton for component j, resulting in at most $|G| \cdot |S_{lev,j}|$ different contributions to $S_{lev+1,j}$. Every execution with $i = j$ results in up to $|G|$ contributions, i.e. $(g', \text{RESTRICT}(A, g'))$ for all $g' \in G$, if A is the result of the closure. Thus, $|S_{lev+1,j}| = O(|G|^{lev+1})$.

It is easy to see that the number of times line 10 is executed is

$$\sum_{lev=0}^{k-1} \sum_{j=1}^{n} |S_{lev,j}| = O(n \cdot |G|^{k-1}).$$

Line 12 is executed $t \cdot (n-1) \cdot |G|$ times. However, it is sufficient to execute the RESTRICT operation only $|G|$ times for every $aut[g, B_i]$ (and store the result for later use). The cost of RESTRICT operation is linear in the number of states of the automaton it is executed on. If $C(A)$ is the maximal cost of computing the closure and $|A|$ is the maximal number of states of an automaton arising in the computation, then the overall time required by the computation can be bounded by

$$O(n^k \cdot |G|^k + n \cdot |G|^{k-1} \cdot (C(A) + |G| \cdot |A|)). \tag{1}$$

We are now ready to state the main result of this section. Let $A_1, \ldots, A_n$ be automata representing $C_1, \ldots, C_n$.

Theorem 2. *Let $M = (g, C_1, \ldots, C_n)$ be an aggregate of an APN $\mathcal{N} = (G, P, \Gamma, \Delta_l, \Delta_g)$ and let $k \in \mathbb{N}$ be a number. Then there exist aggregates $M_0, \ldots, M_m$ such that $post^*_{k,\mathcal{N}}(M)$ (or $pre^*_{k,\mathcal{N}}(M)$, resp.) has the form $M_0 \cup M_1 \cup \ldots \cup M_m$ and all these aggregates are effectively computable. Moreover,*

(a) computing $post^*_{k,\mathcal{N}}(M)$ takes $O(n^k \cdot |G|^k + n \cdot |G|^k \cdot |P| \cdot (d + |\Delta| \cdot k \cdot q + |\Delta|^2 \cdot k^2))$ time, where $|\Delta| = |G| \cdot |\Delta_l| + |\Delta_g|$ and q, d are the largest numbers of non-initial states and transitions leading out of non-initial states in $A_1, \ldots, A_n$, respectively;

(b) $pre^*_{k,\mathcal{N}}(M)$ can be computed in time $O(n^k \cdot |G|^k + n \cdot |G|^{k-1} \cdot (q + k \cdot |P| \cdot |G|)^2 \cdot |\Delta|)$ where $|\Delta| = |G| \cdot |\Delta_l| + |\Delta_g|$ and q is the maximal number of states in $A_1, \ldots, A_n$.

Proof: Equation 1 provides the basis for the theorem; all we need is to determine the parameters $C(A)$ and $|A|$ for both forward and backward reachability.

(a) *Forward reachability:* Notice that $|\Delta| = |G| \cdot |\Delta_l| + |\Delta_g|$ and $|P'| = |P| \cdot |G|$ in line 10. Let q be the largest number of non-initial states and d be the largest number of transitions leading out of non-initial states in any of the initial automata A_i ($1 \leq i \leq n$). Each application of LIFT adds $|P| \cdot (|G| - 1)$ states. Recall that these added states are initial. The subsequent application of RESTRICT makes the same number of added states non-initial, and (since the $post^*$ construction ensures that they do not have any incoming transitions) unreachable, so they can be removed, and each iteration grows the automaton only by $|\Delta|$ states. A component can control at most $\lceil k/2 \rceil$ contexts, so the maximal size of Q_0 will be $q + \lceil k/2 \rceil \cdot |\Delta|$. Moreover, the

$post^*$ algorithm adds $|\Delta|\cdot(|\Delta|+|Q_0|)$ transitions leading out of non-initial states. Due to the bound on $|Q_0|$, the maximal size of δ_0 is $O(d+|\Delta|\cdot k\cdot q+|\Delta|^2\cdot k^2)$. The maximal number of all states in an is $|A|=|Q_0|+|P|\cdot|G|$. Plugging these results into Theorem 1(a) and into Equation 1 yields the result stated in part (a) of the theorem.

(b) *Backward reachability:* Again, we have $|\Delta|=|G|\cdot|\Delta_l|+|\Delta_g|$ in line 10. This time, let q be the largest number of states and d be the largest number of transitions in any of the initial A_i ($1\le i\le n$) automata. For each application of LIFT, we need to add $|P|\cdot(|G|-1)$ states, so the automaton size for every component grows with every context it controls. Again, each component controls at most $\lceil k/2\rceil$ contexts, so the maximal size of Q is $O(q+\lceil k/2\rceil\cdot|P|\cdot|G|)$. This estimate in combination with Theorem 1(b) and Equation 1 prove part (b) of the theorem. □

Note that the complexity given for k-bounded forward CPS reachability in [QR05] has (among others) the factors k^3 and $|G|^{k+5}$. Seeing as APNs are an extension of CPSs, Theorem 2 provides a better upper bound for k-bounded reachability even on CPSs. (This issue should not be confused with the fact that APN can also be a more compact model than CPS, as has been pointed out in Section 2.5.)

4 Forward reachability analysis of ADPN

Even in the DPN case, the $post^*$ image of a regular set of configurations is not always regular [BMOT05]. However, it can be shown that this image is always context-free, and [BMOT05] provides a construction that, given a DPN and an initial configuration $p_0\gamma_0$, computes a context-free grammar $\mathcal{G}$ such that $L(\mathcal{G})=post^*(p_0\gamma_0)$.

In this paper we show how to compute $post^*_{k,\mathcal{N}}(c_0)$ for an ADPN $\mathcal{N}$, a configuration $c_0=(g_0,p_0\gamma_0)$ and an arbitrary $k\ge 0$. (The algorithm can be extended from one configuration c_0 to a regular set of configurations.) The key of the result is a construction which, given a sequence $\sigma=g_1\dots g_k$ of global states of $\mathcal{N}$, constructs a DPN $\mathcal{N}_\sigma$, a configuration c, a regular set S, and a homomorphism π (as we shall see, S, c, and π are independent from σ) such that:

$$post^*_{k,\mathcal{N}}(c_0)=\pi(S\cap\bigcup_{\sigma=g_1\cdots g_k\in G^k}(g_k,post^*_{\mathcal{N}_\sigma}(c)))$$

By the result of [BMOT05], the sets $post^*_{\mathcal{N}_\sigma}(c)$ are effectively context-free, and so $post^*_{k,\mathcal{N}}(c_0)$ is effectively context-free as well.

Informally, given $\sigma=g_1\dots g_k$ the DPN $\mathcal{N}_\sigma$ is able to simulate those execution sequences of $\mathcal{N}$ in which, for every $1\le i\le k$, the i-th context-switch occurs at a configuration of $\mathcal{N}$ with global state g_i. During the simulation, each pushdown component of $\mathcal{N}_\sigma$ maintains a guess about the index of the current context. (Notice that, due to the lack of communication between components of a DPN, a component cannot know how many context-switches have occurred). The component can at any point increase its guess, but cannot decrease it. A wrong guess leads to an unfaithful simulation (see below how to 'filter them away'). Moreover, the component can at any point decide to control the current context (more precisely, the context it guesses is the current one). In such a case, the current global state is mantained as a part of the corresponding local state. Since components cannot communicate, this may lead to an unfaithful simulation, where zero, two or more different components claim to control the same context.

The problem of the unfaithful simulations is solved with the help of the set S and the homomorphism π. We define $\mathcal{N}_\sigma$ so that if a component completes the simulation of a context it claims to have controlled, then it must create an inactive 'marker' (a new component that can do nothing) witnessing this claim. At the end of the simulation we can inspect the inactive markers, and check if every context was indeed controlled by one and at most one component. If this is so, the simulation is faithful, otherwise it is unfaithful. The set S is the set of configurations where every marker appears exactly once, and so intersection with S 'filters out' all the configurations reached by unfaithful simulations. The homomorphism π is used to 'clean up' the configurations so obtained by disposing of the markers and other auxiliary symbols used along the simulation.

Formally, let $\mathcal{N} = (G, P, \Gamma, \Delta_l, \Delta_g)$ be an ADPN, and let $\sigma = g_1 \ldots g_k \in G^k$. The DPN $\mathcal{N}_\sigma = (P_\sigma, \Gamma', \Delta_\sigma)$ is defined as follows. The set P_σ contains:

- a state $[p,i]$ for every $p \in P$, and $1 \le i \le k+1$;
- a state $[\mathbf{g},\mathbf{p},\mathbf{i}]$ for every $g \in G$, $p \in P$, and $1 \le i \le k$;
- a state $\mathbf{i}$ for every $1 \le i \le k$.

The set of stack symbols $\Gamma' = \Gamma \cup \{\bot\}$ contains a fresh symbol $\bot$ denoting a bottom of a stack. This added symbol enables us to rewrite a state even if the correponding stack is empty.

Intuitively, a component in state $[p,i]$ guesses that the simulation is currently in the i-th context. In addition, a component in state $[\mathbf{g},\mathbf{p},\mathbf{i}]$ claims to be in control of the i-th context by global state g. The configuration c is given by $c = [\mathbf{g_0},\mathbf{p_0},\mathbf{1}]\gamma_0\bot$.

The rules Δ_σ follow easily from the intended meaning of $[p,i]$ and $[\mathbf{g},\mathbf{p},\mathbf{i}]$.

- $[p,i]\gamma \hookrightarrow [p,i+1]\gamma$ for every $p \in P$, $1 \le i < k$, $\gamma \in \Gamma$;
  (a component increase its guess on the current context)
- $[p,i]\gamma \hookrightarrow [\mathbf{g_{i-1}},\mathbf{p},\mathbf{i}]\gamma$ for every $p \in P$, $2 \le i \le k$, $\gamma \in \Gamma$, where g_{i-1} is given by σ;
  (a component claims control of the i-th context)
- $[\mathbf{g_i},\mathbf{p},\mathbf{i}]\gamma \hookrightarrow [p,i+1]\gamma \rhd \mathbf{i}$ for every $p \in P$, $1 \le i \le k$, $\gamma \in \Gamma'$, where g_i is given by σ;
  (a component claiming to control the i-th context signals a context-switch leaving a marker)
- the rules corresponding to the original rules of ADPN $\mathcal{N}$:
  - $[\mathbf{g},\mathbf{p},\mathbf{i}]\gamma \hookrightarrow [\mathbf{g},\mathbf{p_1},\mathbf{i}]w_1$ and $[p,i]\gamma \hookrightarrow [p_1,i]w_1$ for every $p\gamma \hookrightarrow p_1 w_1 \in \Delta_l$, $1 \le i \le k$, $g \in G$;
  - $[\mathbf{g},\mathbf{p},\mathbf{i}]\gamma \hookrightarrow [\mathbf{g},\mathbf{p_1},\mathbf{i}]w_1 \rhd [p_2,i]w_2\bot$ and $[p,i]\gamma \hookrightarrow [p_1,i]w_1 \rhd [p_2,i]w_2\bot$ for every $p\gamma \hookrightarrow p_1 w_1 \rhd p_2 w_2 \in \Delta_l$, $1 \le i \le k$, $g \in G$;
  - $[\mathbf{g},\mathbf{p},\mathbf{i}]\gamma \hookrightarrow [\mathbf{g}',\mathbf{p_1},\mathbf{i}]w_1$ for every $(g,p\gamma) \hookrightarrow (g',p_1 w_1) \in \Delta_g$, $1 \le i \le k$;
  - $[\mathbf{g},\mathbf{p},\mathbf{i}]\gamma \hookrightarrow [\mathbf{g}',\mathbf{p_1},\mathbf{i}]w_1 \rhd [p_2,i]w_2\bot$ for every $(g,p\gamma) \hookrightarrow (g',p_1 w_1) \rhd p_2 w_2 \in \Delta_g$, $1 \le i \le k$.

We still have to define the set S and the homomorphism π. The set S consistss of all words without any letter of the form $[\mathbf{g},\mathbf{p},\mathbf{i}]$ and in which each of the markers $\mathbf{1},\ldots,\mathbf{k}$

appears exactly once. This means that exactly one component claimed control of each context. Since in principle there are no restriction on the order in which the markers may appear in a configuration of $\mathcal{N}_\mathfrak{S}$ at the end of a faithful simulation, the size of an automaton accepting S is $O(2^k)$. Finally, the homomorphism π is defined by $\pi([p,i]) = p$ for every $p \in P$, and $1 \leq i \leq k+1$, $\pi(\mathbf{i}) = \pi(\perp) = \varepsilon$ for every $1 \leq i \leq k$, and $\pi(\gamma) = \gamma$ otherwise.

$\mathcal{N}_\mathfrak{S}$ has $O(|P| \cdot |G| \cdot k)$ states and $O(k \cdot (|P| \cdot |\Gamma| + |\Delta_l| \cdot |G| + |\Delta_g|))$ rules. The construction shown in [BMOT05] takes as input a DPN $\mathcal{N} = (P,\Gamma,\Delta)$ and yields a context-free grammar with $O(|P|^2 \cdot |\Delta|)$ productions. So we obtain a context-free grammar of size $O((|\Delta_l| + |\Delta_g|) \cdot |P|^3 \cdot |G|^3 \cdot k^3 \cdot |\Gamma| \cdot |G|^k)$ accepting the set $\bigcup_{\sigma = g_1 \ldots g_k \in G^k} (g_k, post^*_{\mathcal{N}_\mathfrak{S}}(c))$. From these grammars and the automaton accepting S we obtain the final context-free grammar accepting $post^*_{k,\mathcal{N}}(c_0)$ by means of standard constructions. So we have the following result:

Theorem 3. *Let $\mathcal{N} = (G,P,\Gamma,\Delta_l,\Delta_g)$ be an ADPN and let $c_0 = (g_0, p_0\gamma_0)$ be a configuration of $\mathcal{N}$. The set $post^*_{k,\mathcal{N}}(c_0)$ is context-free. A context-free grammar generating it can be constructed in time $O(k^3 \cdot |G|^{k+3} \cdot |P|^3 \cdot |\Gamma| \cdot (|\Delta_l| + |\Delta_g|))$.*

5 Backward reachability analysis of ADPN

We consider here the problem of constructing the pre^*_k images of a regular set of configurations, under the assumption of at most k contexts. We provide a reduction of this problem to the problem of computing pre^* images in the case of DPNs (or in other words to the problem of computing pre^*_1 images), and we provide and efficient algorithm for solving the latter problem. This algorithm improves the complexity of the basic saturation-based procedure proposed in [BMOT05] for symbolic backward reachability analysis of DPN.

5.1 Regular symbolic representations

Our algorithms use a class of automata-based representations for regular sets of configurations (mass configurations) which have been introduced in [BMOT05] for DPN analysis. These representations are finite-state automata in a *special form* defined below.

Let $\mathcal{N} = (G,P,\Gamma,\Delta_l,\Delta_g)$ be an ADPN. Then, a finite-state automaton $A = (Q,\Sigma,\delta,q_0,F)$ is called *$\mathcal{N}$-automaton* if and only if it satisfies the following conditions:

- $\Sigma = P \cup \Gamma$,
- Q can be partitioned into three mutually disjoint subsets Q_0, Q_1, Q_2 such that for all $q \in Q_0, p \in P$ there exists a unique state $q_p \in Q_1$,
- transition relation δ can be partitioned into three disjoint relations $\delta_0, \delta_1, \delta_2$ such that $\delta_0 = \{(q,p,q_p) \mid q \in Q_0, p \in P, q_p \in Q_1\}$, $\delta_1 \subseteq (Q_1 \cup Q_2) \times \Gamma \times Q_2$, and $\delta_2 \subseteq (Q_1 \cup Q_2) \times \{\varepsilon\} \times Q_0$,
- $q_0 \in Q_0$, and $F \subseteq Q_1 \cup Q_2$.

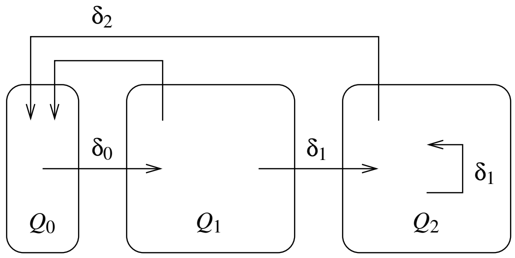


Fig. 3. An automaton in the special form.

An automaton in the above special form is schematically depicted in Figure 3. Notice that $\mathcal{N}$-automata recognize languages which are regular subsets of $(P\Gamma^*)^+$. It is easy to see that, conversely, every finite-state automaton over the alphabet $\Sigma = P \cup \Gamma$ recognizing a language included in $(P\Gamma^*)^+$ can be transformed into a language equivalent $\mathcal{N}$-automaton. Notice also that this definition depends obviously on the model $\mathcal{N}$ under consideration, but only on his set of control states P and his stack alphabet Γ and not on the fact whether global variables and rules are considered.

Following the common habit, we write $q \xrightarrow{a}_\delta q'$ meaning $(q,a,q') \in \delta$. We also extend this notation to finite words in standard way: for every $q,q' \in Q$, $a \in \Sigma$ and $u \in \Sigma^*$ we set $q \xrightarrow{\varepsilon}_\delta q$ and $q \xrightarrow{au}_\delta q'$ iff there is $q'' \in Q$ such that $q \xrightarrow{a}_\delta q''$ and $q'' \xrightarrow{u}_\delta q'$.

5.2 Computing pre^* images for DPN

Let $\mathcal{N} = (P,\Gamma,\Delta)$ be a DPN and $A = (Q,\Sigma,\delta,q_0,F)$ be an $\mathcal{N}$-automaton. We describe a simple procedure proposed in [BMOT05] for computing a finite-state automaton A_{pre^*} satisfying $L(A_{pre^*}) = pre^*_{\mathcal{N}}(L(A))$. The automaton is defined as $A_{pre^*} = (Q,\Sigma,\delta',q_0,F)$, where δ' is the smallest relation $\delta' \supseteq \delta$ satisfying the following two conditions.

- If $p\gamma \hookrightarrow p_1w_1 \in \Delta$ and $q \xrightarrow{p_1w_1}_{\delta'} q'$ for $q,q' \in Q$ then $(q_p,\gamma,q') \in \delta'$.
- If $p\gamma \hookrightarrow p_1w_1 \rhd p_2w_2 \in \Delta$ and $q \xrightarrow{p_2w_2p_1w_1}_{\delta'} q'$ for $q,q' \in Q$ then $(q_p,\gamma,q') \in \delta'$.

The construction of the automaton A_{pre^*} terminates since it corresponds to adding iteratively new transitions to the original automaton A without modifying the number of its states. The construction can be proved to be sound and complete [BMOT05].

It can be seen that this construction is polynomial but a naive implementation of it can be of a prohibitive cost, similarly to the basic algorithm of [BEM97] for pushdown systems with respect to its efficient implementation of [EHRS00]. Following the principles used in [EHRS00], we define an efficient algorithm implementing the saturation-based procedure above.

We assume without loss of generality that for every rule of the considered DPN which is of the form $p\gamma \hookrightarrow p_1w[\rhd p_2u]$, we always have $|w| \leq 2$ and $|u| = 1$.

Then, our algorithm is shown in Figure 4. Let us explain informally the intuition behind the algorithm and the role of each of the used data structures.

Input: A DPN $\mathcal{N} = (P, \Gamma, \Delta)$, and an $\mathcal{N}$-automaton $A = (Q, \Sigma, \delta, q_0, F)$.
Output: The set of transitions rel_1 of the automaton $A_{pre^*} = (Q, \Sigma, rel_1, q_0, F)$.

```
rel_1, rel_2, Δ_1, Δ_2 ← ∅;
W ← {w ∈ Γ ∪ Γ^2 | pγ ↪ p'w ▷ p''γ' ∈ Δ for some p, p', p'' ∈ P and γ, γ' ∈ Γ};
trans_1 ← δ;
trans_2 ← {(q, w, q') ∈ δ ∪ δ^2 | q ∈ Q_1 and w ∈ W};
for all s ∈ Q_0 and pγ ↪ p'ε ∈ Δ do
    trans_1 ← trans_1 ∪ {(s_p, γ, s_{p'})};
while trans_1 ∪ trans_2 ≠ ∅ do
    if trans_1 ≠ ∅ then
        pop t = (q, γ, q') from trans_1;
        if t ∉ rel_1 then
            rel_1 ← rel_1 ∪ {t};
            for all s_{p_1}γ_1 ↪ qγ ∈ Δ_1 do
                trans_1 ← trans_1 ∪ {(s_{p_1}, γ_1, q')};
            if q = s_p ∈ Q_1 then
                if γ ∈ W then
                    trans_2 ← trans_2 ∪ {t};
                for all (q', γ', q'') ∈ rel_1 such that γγ' ∈ W do
                    trans_2 ← trans_2 ∪ {(s_p, γγ', q'')};
                for all (s_{p'}, γ', q) ∈ rel_1 such that γ'γ ∈ W do
                    trans_2 ← trans_2 ∪ {(s_{p'}, γ'γ, q')};
                for all p_1γ_1 ↪ pγ ∈ Δ do
                    trans_1 ← trans_1 ∪ {(s_{p_1}, γ_1, q')};
                for all p_1γ_1 ↪ pγγ_2 ∈ Δ do
                    Δ_1 ← Δ_1 ∪ {s_{p_1}γ_1 ↪ q'γ_2};
                    for all (q', γ_2, q'') ∈ rel_1 do
                        trans_1 ← trans_1 ∪ {(s_{p_1}, γ_1, q'')};
                for all p_1γ_1 ↪ p_2w ▷ pγ ∈ Δ do
                    Δ_2 ← Δ_2 ∪ {s_{p_1}γ_1 ↪ (q', p_2w)};
                    for all (s'_{p_2}, w, q'') ∈ rel_2 such that (q', ε, s') ∈ δ do
                        trans_1 ← trans_1 ∪ {(s_{p_1}, γ_1, q'')};
    if trans_2 ≠ ∅ then
        pop t = (s'_p, w, q) from trans_2;
        if t ∉ rel_2 then
            rel_2 ← rel_2 ∪ {t};
            for all s_{p_1}γ_1 ↪ (q', pw) ∈ Δ_2 such that (q', ε, s') ∈ δ do
                trans_1 ← trans_1 ∪ {(s_{p_1}, γ_1, q)};
return rel_1
```

Fig. 4. Algorithm for DPN backward reachability analysis.

Each time a transition is known to belong to A^*_{pre}, it is added to the set $trans_1$. Then, the algorithm examines each transition in $trans_1$ precisely once and put it in rel_1. The examination of a transition allows to (1) discover new transitions which must be added to $trans_1$, and (2) store informations which will be used later to speed up the discovery of further transitions to be added to $trans_1$. The basic idea to speed up this discovery is as follows. Consider the local rule $p_1\gamma_1 \hookrightarrow p\gamma\gamma_2$. If we see in the current set of transitions $trans_1$ a transition of the form $(s_p, \gamma q')$, then we know that for all transitions of the form (q', γ_2, q'') which have been stored in rel_1 so far, we can add the transition (s_{p_1}, γ_1, q'') to $trans_1$. But, for transitions (q', γ_2, q'') which will be discovered only later (which have not yet entered rel_1 or even $trans_1$), we should store some information allowing to apply the saturation rule when they will be examined. For that, we store in Δ_1 a rule of the form $s_{p_1}\gamma_1 \hookrightarrow q'\gamma_2$ meaning that we are waiting for a transition (q', γ_2, q'') for some arbitrary state q'', and when such a transition will be found (i.e., it will be popped from $trans_1$ for examination), we will generate a new transition (s_{p_1}, γ_1, q'') to $trans_1$.

Now, concerning the dynamic rules, we can adopt the same idea, but we must take care of some technical details. Consider a rule $p_1\gamma_1 \hookrightarrow p_2 w \rhd p\gamma \in \Delta$ where $w = \gamma_2\gamma_3$ (the case $w = \gamma_2$ is similar). Assume that we have a transition (s_p, γ, q') in the set $trans_1$. Then, we need to look for transition sequence $q' \xrightarrow{p_2\gamma_2\gamma_3} q''$, more precisely for a transition $(q', \varepsilon, s') \in \delta$ (note that our algorithm does not add any ε-transition) and for transitions $(s'_{p_2}, \gamma_2, t), (t, \gamma_3, q'') \in rel_1$. If such transitions exist, then we must add the transition (s_{p_1}, γ_1, q'') to $trans_1$. Like in the previous case of local rules, we also need to store informations which will be used later for saturation when other such transitions will be discovered. For that, we store in Δ_2 the rule $s_{p_1}\gamma_1 \hookrightarrow (q', p_2\gamma_2\gamma_3$ meaning that whenever a path $s'_{p_2} \xrightarrow{\gamma_2\gamma_3} q''$ will be added to the automaton for some $s' \in Q_0$ satisfying $(q', \varepsilon, s') \in \delta$ and some state q'', we must add a transition (s_{p_1}, γ_1, q'') to $trans_1$.

To make the saturation of dynamic rules effectively, we maintain a set $trans_2$ of transition sequences which starts from states in Q_1 and which are labelled by a word w appearing in the right-hand-side of the dynamic rules (the set W we define in the beginning of the algorithm corresponds precisely to the set of such words w). Note that these sequences consists of at most 2 transitions due to the considered restriction $|w| \leq 2$. These sequences are examined and transfered to the set rel_2. The examination of such a sequence allow to apply the saturation rule using informations stored in Δ_2 as explained above and in a similar way as for the case of local rules.

The correctness and complexity of the algorithm are proved along the lines of the algorithms of [EHRS00,ERS00] for the computation of pre^* in pushdown systems and in context-free grammars, respectively. In fact, the complexity is the same as that of the context-free grammar case. The reason is that in DPNs the move from a configuration to the next involves a rewriting step that can take place at any component of a configuration $p_1w_1p_2\ldots p_nw_n$. This is similar to the context-free case, where a production can be applied to any occurrence of the variable of its left-hand-side.

Theorem 4. *Given a DPN $\mathcal{N} = (P, \Gamma, \Delta)$ and an $\mathcal{N}$-automaton $A = (Q, \Sigma, \delta, q_0, F)$, it is possible to construct in $O(|Q|^3 \cdot |\Delta|)$ time and $O(|Q|^2 \cdot |\Delta|)$ space an automaton A_{pre^*} such that $L(A_{pre^*}) = pre^*(L(A))$.*

5.3 Computing pre_k^* images for ADPN

Let $\mathcal{N} = (G, P, \Gamma, \Delta_l, \Delta_g)$ be an ADPN, and let $k \geq 1$. Roughly speaking, the computation of a $pre^*_{k,\mathcal{N}}$ image is decomposed into k successive steps of $pre^*_{1,\mathcal{N}}$ image computation, each of them consisting basically in a pre^* image computation in a (suitably defined) DPN. To define in more details the construction, we need some notations and definitions. A *mass configuration* is a pair $M = (g, A)$. It represents the set of configurations (g, u) where $u \in L(A)$. Given a mass configuration $M = (g, A)$, let $local(M)$ denote the automaton A. We generalize this notation to finite collections of mass configurations by taking the union of their $\mathcal{N}$-automata.

Then, given a mass configuration (g, A), the computation of $pre^*_{k,\mathcal{N}}(g, A)$ is performed as follows: first we compute the set $pre^*_{1,\mathcal{N}}(g, A)$ corresponding to all predecessors of (g, A) without context switch. For every global state g', let (g', A') be the set of all configurations in $pre^*_{1,\mathcal{N}}(g, A)$ having g' as global state. Then, the second step constists in computing the $pre^*_{1,\mathcal{N}}$ images of all the pairs (g', A'), for all global states g', and so on. More precisely, given an $\mathcal{N}$-automaton A and a sequence of global states $\sigma \in G^+$, we define inductively the set $\textsc{Reach}_\sigma(A)$:

$$\begin{aligned} \textsc{Reach}_g(A) &= pre^*_{1,\mathcal{N}}(g, A) \\ \textsc{Reach}_{g_1 g_2 \sigma'}(A) &= \textsc{Reach}_{g_2 \sigma'}(local(\textsc{Reach}_{g_1}(A) \cap (g_2, (P\Gamma^*)^+))) \end{aligned}$$

where $g, g_1, g_2 \in G$ and $\sigma' \in G^*$. Then, the following fact holds.

Lemma 2. *Given an ADPN $\mathcal{N}$, a global state g, an $\mathcal{N}$-automaton A, and an integer $k \geq 1$, we have $pre^*_{k,\mathcal{N}}(g, A) = \bigcup_{g_1,\ldots,g_{k-1} \in G^{k-1}} \textsc{Reach}_{g g_1 \cdots g_{k-1}}(A)$.*

Therefore, we only have to show how to construct $pre^*_{1,\mathcal{N}}$ images. For that, we can actually use our algorithm of Theorem 4 which allows to perform backward analysis for DPN. Given an $\mathcal{N}$-automaton A and a global state g, we proceed as follows:

- we construct an automaton $\widehat{A}$ such that for every word u of component configurations which is accepted by A, the automaton $\widehat{A}$ accepts all words arising from u by embedding the global state g into a local state of one of the components. More precisely, $\widehat{A}$ accepts a word w if and only if there is a word $u_1 p u_2 \in L(A)$ such that $u_1 \in (P\Gamma^*)^*$, $p \in P$, $u_2 \in \Gamma^*(P\Gamma^*)^*$, and $w = u_1(g, p)u_2$.
- we transform the sets Δ_l and Δ_g into a set of *local rules* Δ which are applicable to local states (with an embedded global state). The set of obtained rules has a size $O(|G| \cdot |\Delta_l| + |\Delta_g|)$.
- we use the algorithm for DPN of Theorem 4 to build an automaton $\widehat{A}_{pre^*}$.
- then,

$$pre^*_{1,\mathcal{N}}(g, A) = \bigcup_{g' \in G} (g', \{w \in (P\Gamma^*)^+ \ : \ w = upu' \text{ and } \exists u(g', p)u' \in L(\widehat{A}_{pre^*})\}).$$

An automata-based representation for this set can be straightforwardly obtained from $\widehat{A}_{pre^*}$ using intersection and projection. Then, we have the following result.

Theorem 5. *Given an ADPN $\mathcal{N} = (G, P, \Gamma, \Delta_l, \Delta_g)$, $k \geq 1$, $g \in G$, and an $\mathcal{N}$-automaton $A = (Q, \Sigma, \delta, q_0, F)$, it is possible to construct a finite-state automata-based representation of the set $pre^*_{k,\mathcal{N}}(g,A)$ in $O(k^4 \cdot |Q|^3 \cdot (|G|^k \cdot |\Delta_l| + |G|^{k-1} \cdot |\Delta_g|))$ time.*

Proof: *(Sketch)* The justification of the complexity is as follows. For every given sequence $\sigma = g_1 \cdots g_k \in G^{k-1}$, by Lemma 2, our algorithm consists in running k times the algorithm of Figure 4. The latter does not increase the size of its input automaton. However, the automata have to be transformed in special form after each step, which has the effect of adding at most a new copy of the states at each step. Therefore, by Theorem 4, the time complexity for computing $pre^*_{k,\mathcal{N}}(g,A)$ is in $O(G^{k-1} \cdot \sum_{i=1}^{k} (i \cdot |Q|)^3 \cdot (|G| \cdot |\Delta_l| + |\Delta_g|))$, which implies that it is indeed in $O(k^4 \cdot |Q|^3 \cdot (G^k \cdot |\Delta_l| + G^{k-1} \cdot |\Delta_g|))$. $\square$

References

[BEM97] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *Proceedings of CONCUR'97*, LNCS 1243, pages 135–150, 1997.

[BET03] A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *Proceedings of POPL'2003*, pages 62–73. ACM Press, 2003.

[BET04] A. Bouajjani, J. Esparza, and T. Touili. Reachability analysis of synchronized PA-systems. In *Proceedings of Infinity 2004*, 2004. To appear.

[BMOT05] A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown processes. In *Proceedings of CONCUR 2005*, LNCS 3653, pages 473–487, 2005.

[EHRS00] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Proceedings of CAV'2000*, LNCS 1855, pages 232–247, 2000.

[ERS00] J. Esparza, P. Rossmanith, and S. Schwoon. A uniform framework for problems on context-free grammars. *EATCS Bulletin*, 72:169–177, October 2000.

[QR05] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *Proceedings of TACAS'2005*, LNCS 3440, pages 93–107, 2005.

[Ram00] G. Ramalingam. Context-sensitive synchronisation-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems*, 22:416–430, 2000.