Regular Symbolic Analysis of Dynamic Networks of Pushdown Systems

Ahmed Bouajjani[1], Markus Müller-Olm[2], and Tayssir Touili[1]

[1] LIAFA, University of Paris 7, 2 place Jussieu, 75251 Paris cedex 5, France
[2] Universität Dortmund, FB 4, LS 5, Baroper Str. 301, 44221 Dortmund, Germany

Abstract. We introduce two abstract models for multithreaded programs based on dynamic networks of pushdown systems. We address the problem of symbolic reachability analysis for these models. More precisely, we consider the problem of computing effective representations of their reachability sets using finite-state automata. We show that, while forward reachability sets are not regular in general, backward reachability sets starting from regular sets of configurations are always regular. We provide algorithms for computing backward reachability sets using word/tree automata, and show how these algorithms can be applied for flow analysis of multithreaded programs.

1 Introduction

Multithreaded programs are an important class of programs, in which parallelism is used routinely in practice. Parallel programming in general is known to be difficult and error prone, and multithreaded programs are no exception. Therefore, the design of methods and techniques for automatic analysis of such programs is an important and a quite challenging issue. For that, we need to define formal models which are adequate for modelling multithreaded programs, and for which it is possible to construct automatic analysis algorithms.

In recent related work, complete analysis algorithms for abstract classes of parallel programs have been studied by several researchers. Mayr [14] establishes a number of decidability and undecidability results for process classes in the so-called PRS (process rewrite system) hierarchy. PRS are able to model sequential as well as parallel phenomena. In fact, they can be seen as combinations of pushdown systems and Petri nets (defined in a term rewriting setting using prefix and multiset rewrite rules). Following the automata-based approach for the symbolic verification of pushdown systems [2, 11], Lugiez and Schnoebelen [13] show how to use tree automata for reachability analysis of PA processes [1], a particularly well-known class in the PRS hierarchy. Their paper has inspired further work that applies tree automata techniques to analysis of more expressive models [6, 7, 3, 4, 21]. Another line of research generalizes fixpoint-based techniques as common in flow analysis to analysis of similar models of parallel programs [20, 15, 16]. Both approaches can be used to solve bitvector problems, a certain type of simple but important data-flow-analysis problems, for flow graph systems with parallel calls of procedures, or, equivalently, parbegin/parend-blocks interprocedurally [9, 10, 20]. While [9, 10] reduce the problem to reachability analysis of PA-processes, [20] uses fixpoint-based techniques.

Unfortunately, these results do *not* cover interprocedural analysis of multithreaded programs because commands that start new threads cannot adequately be modelled by

M. Abadi and L. de Alfaro (Eds.):CONCUR'05, LNCS 3653, pp. 473–487, 2005.

parallel calls. In a multithreaded program such a command typically returns immediately (see, e.g., the JAVA or POSIX thread API). Therefore the father of a new thread can pursue its execution concurrently to its son and can even terminate or return to its caller while the son is still alive. In contrast, a parallel call returns only when and if all its component processes have terminated, which is a fundamentally different behavior. Indeed we show in Sect. 2 that in presence of procedures, multithreaded programs can have trace languages different from that of any program with parallel calls.

The goal of this paper is to adapt the automata-based approach mentioned above to interprocedural (reachability) analysis of multithreaded programs. For this purpose we propose two models of multithreaded programs, show how to perform reachability analysis for them with automata-theoretic constructions, and discuss their utility for modelling and analysing multithreaded and other classes of parallel programs.

In Sect. 2 we introduce *Dynamic Pushdown Networks (DPNs)* as a basic model of multithreaded programs. Intuitively, a DPN is a network of pushdown processes that run independently in parallel. Each process can create new members of the network as a side effect of a pushdown transition. DPNs thus model a network of threads each of which can perform basic actions, call (recursively) procedures, and *spawn* new processes. We show that while forward reachability of DPNs does not preserve regularity of configuration sets in general, it still preserves context-freeness (Sect. 4). Backward reachability in contrast preserves regularity and we show how to compute the backward reachability set of a regular set of configurations by means of a saturation algorithm in polynomial time (Sect. 4). We also show that DPN allow us to solve bitvector problems interprocedurally for multithreaded programs (Sect. 3), contrary to previously used models in the literature such as PA processes (Sect. 2).

We extend DPNs to *Constrained DPNs (CDPN)* in Sect. 5, a model that combines (indeed even extends) the modelling power of both DPNs and PA (and even the so-called PAD [14]). The new idea is that enabledness of a transition for a process can be made dependent on a *constraint* which is a regular pattern among the sequence of control states of its sons. We require constraints to be *stable* in the sense that further evolution of the sons cannot invalidate a constraint. We show that otherwise we lose the property that backward reachability preserves regularity. Transition rules with stable constraints increase the expressive power considerably over DPNs. In particular they allow us to model, in addition to thread creation and procedure calls, also parallel calls and various types of join commands among other things. It also allows us to return information back from procedures called in parallel to their caller which cannot be handled in PA and not even in PAD. Constrained DPNs inherit from DPNs that forward reachability does not preserve regularity. Therefore, we consider here backward reachability only. We show that the set of configurations that can reach a given regular set of configurations of a CDPN can again be computed by a saturation algorithm. As configurations of CDPNs are given by unbounded width trees rather than by words as in the DPN case—the tree structure captures the father-son relationship—we resort to hedge automata here [8]. The construction is nontrivial and its justification uses in a subtle manner the assumption about the stability of the constraints in the system definition. While the overall complexity of this procedure is exponential—we indeed prove a PSPACE lower bound—it is exponential only in the number of different constraints used in the rules of the given CDPN, and just polynomial in the other problem parameters. Therefore, if the number of different constraints is bounded, we obtain a polynomial-time analysis algorithm. This in particular holds if we just model (in addition to spawn operations), parallel calls, a fixed selection of join commands, or a combination of these. Due to lack of space, proofs are omitted. They can be found in [5].

2 Dynamic Pushdown Networks

A *Dynamic Pushdown Network* (DPN) is a tuple $M = (Act, P, \Gamma, \Delta)$, where Act is a finite set of visible *actions*, P is a finite set of *control states*, Γ is a finite set of *stack symbols* disjoint from P, and Δ is a finite set of transition rules of the following forms: either (a) $p\gamma \overset{a}{\hookrightarrow} p_1 w_1$, or (b) $p\gamma \overset{a}{\hookrightarrow} p_1 w_1 \rhd p_2 w_2$, where $p, p_1, p_2 \in P$, $a \in Act$, $\gamma \in \Gamma$, and $w_1, w_2 \in \Gamma^*$. A DPN can be seen as a collection of identical sequential processes running in parallel, each of them being able to (1) perform pushdown operations and to (2) create processes in the network. Synchronization is not allowed between processes.

A configuration of a DPN M (also called M-configuration) is a word over the alphabet $\Sigma = P \cup \Gamma$ starting with a symbol in P. An M-configuration can be seen as a sequence of (sub)words in $P\Gamma^*$ each of them corresponding to the configuration of one of the processes running in parallel in the network. Let $Conf_M$ be the set of all M-configurations.

For every $a \in Act$, we define $\xrightarrow{a}_M$ to be the smallest relation in $Conf_M \times Conf_M$ s.t. $\forall u, v \in Conf_M$, $u \xrightarrow{a}_M v$ iff (1) there is a rule $p\gamma \overset{a}{\hookrightarrow} p_1 w_1$ in Δ s.t. $u = u_1 p\gamma u_2$ and $v = u_1 p_1 w_1 u_2$, or (2) there is a rule $p\gamma \overset{a}{\hookrightarrow} p_1 w_1 \rhd p_2 w_2$ in Δ s.t. $u = u_1 p\gamma u_2$ and $v = u_1 p_2 w_2 p_1 w_1 u_2$. We write $u \rightarrow_M v$ if there exists $a \in Act$ s.t. $u \xrightarrow{a}_M v$.

The semantics above says that rules of the form (a) correspond precisely to pushdown operations (manipulation of the top of the stack) which can be applied anywhere in the configuration (i.e., by any of the processes in the network): if a process is at control state p and has γ as topmost stack symbol, then it can move to control state p_1 and replace γ by w_1 at the top of its stack. Rules of the form (b) allow in addition the creation of new processes: a process with control state p and topmost stack symbol γ can (1) move to state p_1 and modify its stack by replacing γ with w_1, and moreover, (2) create (to its left) a process which starts its execution at the initial configuration $p_2 w_2$.

Given a configuration c, the set of immediate predecessors (resp. successors) of c is $\mathsf{pre}_M(c) = \{c' \in \mathcal{C} \,:\, c' \rightarrow_M c\}$ (resp. $\mathsf{post}_M(c) = \{c' \in \mathcal{C} \,:\, c \rightarrow_M c'\}$). These notations can be generalized straightforwardly to sets of configurations. Let pre^*_M (resp. post^*_M) denote the reflexive-transitive closure of pre_M (resp. post_M). We omit the subscript M when it is understood from the context. Given $\Delta' \subseteq \Delta$, we use $\mathsf{pre}_{\Delta'}$ (resp. $\mathsf{post}_{\Delta'}$) to denote immediate predecessors (resp. successors) using a rule in Δ'. Then, $\mathsf{pre}^*_{\Delta'}$ and $\mathsf{post}^*_{\Delta'}$ denote the corresponding reflexive-transitive closures. Furthermore, $\mathsf{Traces}_M(c) = \{w \in Act^* \,:\, \exists c'.\, c \xrightarrow{w}_M c'\}$ is the set of traces generated by c.

DPN vs. PA Processes: DPNs allow to model multithreaded programs where creation of threads is done using spawn commands (see Sect. 3). This is not the case for other formalisms used in the literature for modelling parallel programs like PA [1]:[3]

[3] PA corresponds to processes definable by a set of rewrite rules of the form $A \rightarrow t$ where A is a process variable, and t is a term built from process variables, sequential composition, and asynchronous parallel composition.

Theorem 1. *Let* $\mathcal{L} = \bigcup\{a^n(b^{n'} \otimes (c^m d^{m'})) : n \geq n' \geq 0,\ m \geq m' \geq 0\}$*, where* $\otimes$ *denotes the shuffle (or interleaving) operator defined as usual. Then:*

a) There is a DPN M and an M-configuration c such that $\mathsf{Traces}_M(c) = \mathcal{L}$*.*
b) There is no PA system Δ *and no process variable A such that* $\mathsf{Traces}_\Delta(A) = \mathcal{L}$*.*

Hence, PA processes are inadequate for capturing the behavior of multithreaded programs with spawn-like creation of threads. It also follows from the proof that trace sets of DPNs cannot be captured by the type of constraint systems used as semantic reference point in the constraint-based approach [20, 15, 16]. Therefore, the methods of [9, 10, 20, 16, 15] for interprocedural analysis of flow graphs with parallel calls do not carry over immediately to multithreaded programs. These inadequacy results are rather strong because any interesting process equivalence would imply equality of traces.

3 Program Analysis Based on DPN

We show hereafter how DPNs can be used to model multithreaded programs and how our results on symbolic reachability analysis can be used in flow analysis of these programs. This is inspired by Esparza et. al. [9, 10].

Flow Graph Systems: As common in program analysis we assume that the program is given by a flow graph system. Let **Proc** be a finite set of procedure names containing **Main**. We assume that the program operates on a set $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$ of global variables. We consider the following types of basic statements: assignment statements, $\mathbf{x}_i := e$, where $\mathbf{x}_i \in \mathbf{X}$ and e is some expression; call of a single procedure, $\mathsf{call}(\pi)$, where $\pi \in \mathbf{Proc}$; and spawn of a new thread, $\mathsf{spawn}(\pi)$, where $\pi \in \mathbf{Proc}$. The intuitive meaning of assignment statements and calls is obvious. The spawn command $\mathsf{spawn}(\pi)$ models creation of a new independent thread. Like the call $\mathsf{call}(\pi)$, $\mathsf{spawn}(\pi)$ starts an instance of procedure π. In contrast to a call, however, the spawn command returns immediately such that the newly created instance of π runs as a new thread concurrently to the statements that are executed after the spawn. Let **Stmt** be the set of basic statements.

The control flow of each procedure $\pi \in \mathbf{Proc}$ is described by a control flow graph $G_\pi = (N_\pi, E_\pi, e_\pi, x_\pi)$, where N_π is a finite set of program points of procedure π; $E_\pi \subseteq N_\pi \times \mathbf{Stmt} \times N_\pi$ is a finite set of edges annotated by basic statements; $e_\pi \in N_\pi$ is the entry point of π; and $x_\pi \in N_\pi$ is the exit point of π. We assume that the sets of program points of different procedures are disjoint, $N_\pi \cap N_{\pi'} = \emptyset$ if $\pi, \pi' \in \mathbf{Proc}$, $\pi \neq \pi'$, and agree that $N = \bigcup_{\pi \in \mathbf{Proc}} N_\pi$ and $E = \bigcup_{\pi \in \mathbf{Proc}} E_\pi$.

From Flow Graph Systems to DPN: From a given flow graph system as above we construct a DPN $M = (Act, P, \Gamma, \Delta)$ that captures its operational semantics:

- The actions are given by the assignments that appear in the flow graph system; a special symbol τ is used to signify steps in which no assignment is executed: $Act = \{\mathbf{x} := e \mid \exists u, v : (u, \mathbf{x} := e, v) \in E\} \cup \{\tau\}$;
- we have just one artificial control state #: $P = \{\#\}$;
- we work with a stack of program points; the topmost stack symbol is the current program point of the current procedure, the other stack symbols are the return points of its callers: $\Gamma = N$;

- the transition rules in Δ describe computation steps of the flow graph system:
  1. for every assignment edge $(u, x := e, v) \in E$ we put the rule $\#u \overset{x:=e}{\hookrightarrow} \#v$ to Δ;
  2. for every call edge $(u, \mathsf{call}(\pi), v) \in E$ we put the rule $\#u \overset{\tau}{\hookrightarrow} \#e_\pi v$ to Δ;
  3. for every spawn-edge $(u, \mathsf{spawn}(\pi), v) \in E$ we put the rule $\#u \overset{\tau}{\hookrightarrow} \#v \triangleright \#e_\pi$ to Δ,
  4. for each procedure $\pi \in \mathbf{Proc}$, we put the rule $\#x_\pi \overset{\tau}{\hookrightarrow} \#$ to Δ. This rule describes the return from procedure π.

Note that it is possible to extend the semantics above in order to handle local procedure variables and return values from procedure calls. For that, we assume as usual that data values are mapped into a finite abstract domain using standard techniques such as predicate abstraction. Then, abstract values of local variables can be encoded in the stack alphabet and abstract return values can be encoded in the control states.

Solving Bitvector Problems: The operational semantics given above can be used for solving bitvector problems. In order to ease comparison with [10] we discuss detection of live (global) variables. Other bitvector problems can be solved in a similar fashion. Informally, a variable $\mathbf{x} \in \mathbf{X}$ is *live* at a program point $u \in N$ if there is an execution from u in which $\mathbf{x}$ is used before it is over-written. We restrict attention to *reachable* configurations and use a similar definition and notation as Esparza and Podelski [10]. Thus, we define: program variable $\mathbf{x}$ is *live* at a program point $u \in N$ if there is a transition sequence $\#e_{\mathbf{Main}} \xrightarrow{\sigma_1} c_1 \xrightarrow{\sigma_2} c_2 \xrightarrow{\mathbf{y}:=e} c_3$ such that: (1) u is *active* in configuration c_1, i.e., appears as the topmost stack symbol of one of the parallel pushdown processes in the network described by c_1; (2) σ_2 is a sequence of statements that do not modify $\mathbf{x}$ (i.e., do not write to $\mathbf{x}$); and (3) e is an expression in which $\mathbf{x}$ is used.

We denote the set of configurations c in which u is active by At_u, the set of assignments in the given program that modify $\mathbf{x}$ by $\mathsf{Mod}_\mathbf{x} \subseteq Act$, and the set of assignments in the program in which $\mathbf{x}$ is used by $\mathsf{Use}_\mathbf{x} \subseteq Act$. Moreover, we write Δ_A for the set of rules of Δ with an action in a subset $A \subseteq Act$: $\Delta_A = \{(p\gamma \overset{a}{\hookrightarrow} w) \in \Delta \mid a \in A\}$. Using this notation it is not hard to see that $\mathbf{x}$ is live at u if and only if

$$\#e_{\mathbf{Main}} \in \mathsf{pre}^*(\mathsf{At}_u \cap \mathsf{pre}^*_{\Delta_{Act \setminus \mathsf{Mod}_\mathbf{x}}}(\mathsf{pre}_{\Delta_{\mathsf{Use}_\mathbf{x}}}(Conf_M)))$$

Then, our results concerning backward reachability analysis of DPN given in the next section (see Theorem 3 and Note 1) can be used to decide this property.

4 Reachability Analysis for DPN

We consider the problem of computing representations of the post^* and pre^* images of given sets of configurations. We are interested in the case that sets of configurations are effectively given using automata-based representations.

Computing post^* Images: We show first that post^* does not preserve regularity in general. Consider indeed the DPN $M = (\{a\}, \{p\}, \{\gamma_1, \gamma_2\}, \{p\gamma_1 \overset{a}{\hookrightarrow} p\gamma_1\gamma_1 \triangleright p\gamma_2\})$. It is easy to see that $\mathsf{post}^*_M(\{p\gamma_1\}) = \{(p\gamma_2)^n p\gamma_1^{n+1} \ : \ n \geq 0\}$, which is clearly nonregular.

Proposition 1. *There is a DPN M, and a configuration c of M, such that* $\mathsf{post}^*(c)$ *is not a regular set of configurations.*

We prove, however, that post^* preserves context-freeness:

Theorem 2. *For every DPN M and any context-free set C of M-configurations, the set* $\mathsf{post}^*(C)$ *is context-free and effectively constructible in polynomial time.*

Computing pre^* Images: We show now that pre^* preserves regularity. Let M be a DPN and $\mathcal{A}$ be an automaton recognizing a set of M-configurations. We define a polynomial-time algorithm allowing to construct an automaton $\mathcal{A}_{\mathsf{pre}^*}$ s.t. $L(\mathcal{A}_{\mathsf{pre}^*}) = \mathsf{pre}^*_M(L(\mathcal{A}))$. For technical reasons, we require that $\mathcal{A}$ is in a special form we define below.

M-Automata: Let $M = (Act, P, \Gamma, \Delta)$ be a DPN. A finite automaton $\mathcal{A} = (S, \Sigma, \delta, s^0, F)$ is an M-automaton if the following conditions hold:

1. $\Sigma = P \cup \Gamma$ is the finite alphabet,
2. the set of states is partitioned into two sets, $S = S_\mathrm{c} \cup S_\mathrm{s}$, $S_\mathrm{c} \cap S_\mathrm{s} = \emptyset$,
3. for every $s \in S_\mathrm{c}$ and every $p \in P$, there is a (unique and distinguished) state $s_p \in S_\mathrm{s}$,
4. there is a relation $\delta' \subseteq S_\mathrm{s} \times \Gamma \times (S_\mathrm{s} \setminus \{s_p \,:\, s \in S_\mathrm{c}, p \in P\}) \cup S_\mathrm{s} \times \{\varepsilon\} \times S_\mathrm{c}$ such that $\delta = \delta' \cup \{(s, p, s_p) \,:\, s \in S_\mathrm{c}, p \in P\}$,
5. the initial state $s^0 \in S_\mathrm{c}$, and
6. $F \subseteq S$ is the set of final states.

For $\sigma \in \Sigma \cup \{\varepsilon\}$ and $s, s' \in S$, we write $s \xrightarrow{\sigma}_\delta s'$ in lieu of $(s, \sigma, s') \in \delta$. We extend this notation in the obvious manner to sequences of symbols: (1) $\forall s \in S.\ s \xrightarrow{\varepsilon}_\delta s$, and (2) $\forall s, s' \in S.\ \forall \sigma \in \Sigma \cup \{\varepsilon\}.\ \forall w \in \Sigma^*.\ s \xrightarrow{\sigma w}_\delta s'$ iff $\exists s'' \in S.\ s \xrightarrow{\sigma}_\delta s''$ and $s'' \xrightarrow{w}_\delta s'$.

Note that requirement (4) codes a number of conditions on δ: (1) each $s \in S_\mathrm{c}$ has s_p as its unique p-successor and has no Γ-transitions, (2) s is the only predecessor of s_p, (3) only ε-moves from states in S_s lead to states $s \in S_\mathrm{c}$, (4) states $s \in S_\mathrm{s}$ do not have p-successors, for any $p \in P$. So, every path in an M-automaton (starting from the initial state) is the concatenation of paths of the form $s \xrightarrow{p}_\delta s_p \xrightarrow{w}_\delta t \xrightarrow{\varepsilon}_\delta s'$ where $s, s' \in S_c$, $p \in P$, $w \in \Gamma^*$, and all states in the path $s_p \xrightarrow{w}_\delta t$ are in S_s. Note that for every finite automaton $\mathcal{A}$ over the alphabet $P \cup \Gamma$ such that $L(\mathcal{A}) \subseteq Conf_M$, it is possible to construct an M-automaton recognizing the same language.

Constructing the Automaton $\mathcal{A}_{\mathsf{pre}^*}$: Let M be a DPN and $\mathcal{A} = (S, \Sigma, \delta, s^0, F)$ be an M-automaton. The construction of $\mathcal{A}_{\mathsf{pre}^*}$ is in the same spirit as the ones for single pushdown systems (see [2]). It consists in adding iteratively new transitions to the automaton $\mathcal{A}$ according to *saturation* rules (reflecting the backward application of the transition rules in the system), while the set of states remains unchanged. Therefore, we define $\mathcal{A}_{\mathsf{pre}^*}$ to be the finite-state automaton $(S, \Sigma, \delta', s^0, F)$, where δ' is the smallest relation which contains δ (i.e., $\delta \subseteq \delta'$) and satisfies the following conditions:

$R1$: If $(p\gamma \overset{a}{\hookrightarrow} p_1 w_1) \in \Delta$ and $s \xrightarrow{p_1 w_1}_{\delta'} s'$, for $s, s' \in S$, then $(s_p, \gamma, s') \in \delta'$.
$R2$: If $(p\gamma \overset{a}{\hookrightarrow} p_1 w_1 \rhd p_2 w_2) \in \Delta$ and $s \xrightarrow{p_2 w_2 p_1 w_1}_{\delta'} s'$, for $s, s' \in S$, then $(s_p, \gamma, s') \in \delta'$.

The relation δ' can be computed as the limit of an increasing sequence of relations obtained by adding transitions to δ that are required by one of the implications above. This procedure terminates after a polynomial number of steps since only a polynomial number of transitions can potentially be added.

Let us explain intuitively the role of the saturation rule (R_1). Consider a path in the automaton of the form $s \xrightarrow{p_1 w_1} s'$. This means, by definition of M-automata, that s is necessarily in S_c and that we have $s \xrightarrow{p_1} s_{p_1} \xrightarrow{w_1} s'$. Then, the rule consists in adding to the automaton the transition $s_p \xrightarrow{\gamma} s'$. Since by definition of M-automata we have $s \xrightarrow{p} s_p$, we obtain a path $s \xrightarrow{p\gamma} s'$ in the automaton. Therefore, if a configuration $u_1 p_1 w_1 u_2$ is recognized by a run $s^0 \xrightarrow{u_1} s \xrightarrow{p_1 w_1} s' \xrightarrow{u_2} s_F$, then its predecessor $u_1 p \gamma u_2$ is also recognized due to the new transition by the run $s^0 \xrightarrow{u_1} s \xrightarrow{p\gamma} s' \xrightarrow{u_2} s_F$. The role of ($R_2$) is similar.

Theorem 3. $L(\mathcal{A}_{\mathsf{pre}^*}) = \mathsf{pre}^*_M\big(L(\mathcal{A})\big)$.

Note 1. For the sake of completeness, we mention that for every DPN M, and every M-automaton $\mathcal{A}$, the sets $\mathsf{pre}_M(\mathcal{A})$ and $\mathsf{post}_M(\mathcal{A})$ are regular and effectively constructible. The constructions are quite straightforward. For pre_M we take two copies of $\mathcal{A}$. The first copy provides the initial state and the second copy the final states. We then apply the saturation rules to the first copy of the automaton, but let all new transitions lead from states of the first copy to states of the second copy. The post_M construction is similar (it needs adding a finite number of intermediary states).

5 Constrained DPN

We consider in this section an extension of the DPN model introduced in Section 2. In addition to the ability of performing spawn operation as previously, processes are now allowed to observe the control states of their children (processes they have created in the past). This is relevant in particular for handling return values and some kinds of *join* statements between parallel processes. To achieve that, we define a model where the application of a transition rule by some process is conditioned by a (regular language) constraint on the sequence of control states of its children. We need however to impose a *stability* condition (defined below) on the constraints in order to have a model which can be analysed by means of finite-state automata representations. We show later that we lose regularity of the reachability sets if we relax the stability condition.

Stable Regular Languages: Let Σ be a finite alphabet and let $\rho \subseteq \Sigma \times \Sigma$ be a binary relation over Σ. Then, a set of symbols $S \subseteq \Sigma$ is ρ-*stable* iff $\forall s \in S.\ \forall t \in \Sigma.\ (s,t) \in \rho \Rightarrow t \in S$. A ρ-stable regular language over Σ is a subset of Σ^* which is definable by a regular expression of the form:

$$e ::= S,\ \text{a } \rho\text{-stable set} \mid e + e \mid e \cdot e \mid e^*$$

We can prove straightforwardly by induction on the structure of regular expressions:

Lemma 1. *Let $\phi \subseteq \Sigma^*$ be a ρ-stable regular language, let $u, v \in \Sigma^*$, and let $a \in \Sigma$ such that $uav \in \phi$. Then, for every $b \in \Sigma$, $(a,b) \in \rho$ implies that $ubv \in \phi$.*

Definition of the Models: A *Constrained Dynamic Pushdown Network* (CDPN) is a tuple $M = (Act, P, \Gamma, \Delta)$, where Act is a finite set of visible *actions*, P is a finite set of *control states*, Γ is a finite set of *stack symbols* disjoint from P, and Δ is a finite set of transition rules of the following forms: either (a) $\phi : p\gamma \overset{a}{\hookrightarrow} p_1 w_1$, or (b) $\phi : p\gamma \overset{a}{\hookrightarrow} p_1 w_1 \rhd p_2 w_2$, where $p, p_1, p_2 \in P$, $a \in Act$, $\gamma \in \Gamma$, $w_1, w_2 \in \Gamma^*$, and ϕ is a ρ_Δ-stable regular language over P, with $\rho_\Delta = \{(p, p') \in P \times P$: there is a rule $\psi : p\delta \overset{a}{\hookrightarrow} p'u$ or $\psi : p\delta \overset{a}{\hookrightarrow} p'u \rhd p''v$ in $\Delta\}$.

A CDPN consists of a collection of identical sequential processes running in parallel, each of them being modeled as a pushdown system which is able to (1) manipulate its own stack using pushdown rules of the form (a), (2) create a new process (which becomes its youngest son) using rules of the form (b), and (3) observe, under some conditions, the states of its children (processes it created in the past): each transition rule is constrained by the fact that the sequence of control states of the children (given in the decreasing order of their age) must belong to the specified language ϕ.

Since we need to refer to the children of each process, a configuration of a CDPN can be naturally seen as a tree where each vertex is annotated with the configuration of some sequential process (pushdown system), and where the structure corresponds to the relation father-son. Notice that such a tree may have an arbitrary width. We define hereafter a class of terms describing such configurations and we define a transition relation between such terms.

M-Terms: Let $X = \{x_1, \ldots, x_n\}$ be a set of variables. We define the set $\mathcal{T}[X]$ of M-terms over $P \cup \Gamma \cup X$ inductively as follows:

- $X \subseteq \mathcal{T}[X]$,
- If $t \in \mathcal{T}[X]$ and $\gamma \in \Gamma$, then $\gamma(t) \in \mathcal{T}[X]$,
- If $t_1, \ldots, t_n \in \mathcal{T}[X]$ and $p \in P$, then $p(t_1, \ldots, t_n) \in \mathcal{T}[X]$, for $n \geq 0$.

Note that in the last item of this definition, n can be 0 (i.e., p is on a leaf). In that case, we write $p()$ or simply p to represent the corresponding term.

Terms in $\mathcal{T}[\emptyset]$ are called *ground terms*, and will also be denoted by $\mathcal{T}$. A term in $\mathcal{T}[X]$ is linear if each variable occurs at most once. A *context* C is a linear term. Let $t_1, \ldots, t_n$ be n ground terms. Then $C[t_1, \ldots, t_n]$ is the ground term obtained by substituting in C the occurrence of the variable x_i with the term t_i, for $1 \leq i \leq n$.

A term in $\mathcal{T}[X]$ can be seen as a rooted labeled tree of arbitrary width, where (1) an internal node is either of arity 1 (has one successor) if it is labeled with a stack symbol $\gamma \in \Gamma$, or it has an arbitrary arity if it is labeled with a state $p \in P$, and (2) where the leaves are labeled with either variables $x \in X$, or with states $p \in P$.

M-Configurations: We define M-configurations to be the ground M-terms (terms in $\mathcal{T}[X]$ without variables). Given n ground terms $t_1, \ldots, t_n$, the term $\gamma_m \cdots \gamma_1 p(t_1, \ldots, t_n)$ represents a configuration where (1) the common ancestor to all processes is at local control state p and has $\gamma_1 \cdots \gamma_m$ as stack content, where γ_1 is the topmost stack symbol, and (2) this process has n children, the i^{th} of which is described, together with all of its descendants, by the term t_i, for $i = 1, \ldots, n$. A ground term of the form $\gamma_m \cdots \gamma_1 p$ corresponds to the case of one single process without children.

Transition Relation: Given a CDPN M, we define a transition relation $\to_M$ between M-configurations. We introduce first a notation. Given a configuration t of one of the forms $\gamma_m \cdots \gamma_1 p(t_1, \ldots, t_n)$ or $\gamma_m \cdots \gamma_1 p$, we define $\mathcal{S}(t)$ to be the control state p, i.e., $\mathcal{S}(t)$ is the local control state of the topmost process represented in t. Then, $\to_M$ is the smallest relation between M-configurations such that:

- If $(\phi : p\gamma \overset{a}{\hookrightarrow} p_1 w_1) \in \Delta$ and $\mathcal{S}(t_1) \cdots \mathcal{S}(t_n) \in \phi$, then

$$C\big[\gamma p(t_1, \ldots, t_n)\big] \to_M C\big[w_1^R p_1(t_1, \ldots, t_n)\big]$$

- If $(\phi : p\gamma \overset{a}{\hookrightarrow} p_1 w_1 \rhd p_2 w_2) \in \Delta$ and $\mathcal{S}(t_1) \cdots \mathcal{S}(t_n) \in \phi$, then

$$C\big[\gamma p(t_1, \ldots, t_n)\big] \to_M C\big[w_1^R p_1(t_1, \ldots, t_n, w_2^R p_2)\big]$$

where w^R denotes the reverse word (mirror image) of w. The notions of post, pre, post^*, and pre^* are defined as usual.

Modelling Power: Since CDPN generalize DPN, the modelling of programs with spawn operations given in Section 3 is still valid for CDPN. Moreover, stable constraints as preconditions of transition rules increase tremendously the modelling power of our formalism. We discuss some applications in this section.

Parallel Calls: In the data-flow analysis scenario, we can use constraints, e.g., in order to accommodate parallel call commands as another basic primitive for creation of parallelism in addition to spawn commands. A parallel call, $\mathsf{pcall}(\pi, \pi')$ with $\pi, \pi' \in \mathbf{Proc}$ starts an instance of procedure π and an instance of π' and runs them in parallel. It terminates if and when both these instances terminate.

Assume that we extend the flow-graph model of Section 3 by allowing parallel calls as another type of basic statement. In the CDPN model we capture the operational semantics of an edge $(u, \mathsf{pcall}(\pi, \pi'), v)$ as follows: we start two new threads for π and π' and ensure by a transition rule with an appropriate constraint that we can move to v only after both these threads have terminated. For that, both threads indicate termination by moving to a special new "terminated" control state $\natural$ when they see a special new stack symbol \$ that we put at the bottom of their stack upon thread creation. Thus, we have the following rules for modelling $(u, \mathsf{pcall}(\pi, \pi'), v)$:

$$P^* : \#u \overset{\tau}{\hookrightarrow} \#\gamma_1 \rhd \#e_\pi\$ \qquad P^* : \#\gamma_1 \overset{\tau}{\hookrightarrow} \#\gamma_2 \rhd \#e_{\pi'}\$ \qquad P^*\natural^2 : \#\gamma_2 \overset{\tau}{\hookrightarrow} \#v$$

where γ_1, γ_2 are two auxiliary stack symbols chosen fresh for each parallel call. Moreover, the rule $P^* : \#\$ \overset{\tau}{\hookrightarrow} \natural$ allows a thread to move to the state $\natural$ once it has terminated.

Join Statements: Besides parallel calls we can also model different types of join-commands. We use the same technique as above for making termination visible to the father of threads: we now use the rule $\#u \overset{\tau}{\hookrightarrow} \#v \rhd \#e_p\$$ to describe the behavior of a spawn edge $(u, \mathsf{spawn}(p), v) \in E$. Thus, we mark the bottom of the stack with the special symbol \$. We also use the rule $P^* : \#\$ \overset{\tau}{\hookrightarrow} \natural$ from above to make termination visible in the control state. This allows us to describe the operational semantics of different types of join-command such as for instance (1) $\mathsf{join}_\forall$: proceed if all threads directly created by the current thread have terminated, and (2) $\mathsf{join}_{\exists k}$: proceed if at least k among the threads directly created by the current thread have terminated.

The behavior of an edge (u, j, v) where j is one of the join commands from above is modelled by the rule $\phi : \#u \xhookrightarrow{\tau} \#v$ where $\phi = \natural^*$ for $j = \mathsf{join}_\forall$, and $\phi = (P^*\natural)^k P^*$ for $j =\mathsf{join}_{\exists k}$. Obviously, these constraints are stable.

Return Values: We can distinguish between different termination conditions by using more than one terminated control state and use regular patterns of such control states in constraints in the father process. This allows us, for instance, to return information back to the caller from procedures called in parallel. Therefore, the modelling power of CDPNs exceeds that of PA and even that of PAD [4] [14]: While in a PAD process (like in a DPN process) we can use control states to return information back to a caller in a normal procedure call, there is no such mechanism for parallel calls. The modelling power for calls and parallel calls is thus more symmetric for CDPNs than for PAD.

Observing Execution Phases: Finally, as we allow *stable* constraints, a creator of a thread can react on situations in which the created thread has achieved some progress already but is not necessarily terminated yet. As an example, let us assume that a process F (the father) creates a number of worker threads that sequentially go through a number of phases, say phases $1, \ldots, n$, before termination. For modelling the worker threads we use new control states from a hierarchy $P_0 \supset P_1 \supset \ldots \supset P_n = \emptyset$ of control states such that a worker thread is in phase i if and only if its control state is in $P_{i-1} \setminus P_i$. This means a worker thread has finished phase i if and only if its control state belongs to P_i. Then, the sets P_i are stable and can be used as building blocks for constraints in transitions of F. Hence, process F can react on situations like "all worker threads have finished phase i" by using the constraint P_i^*, "there is a worker thread that has finished phase i and all other worker threads have finished phase j" by the constraint $P_j^* P_i P_j^*$, etc.

6 Backward Reachability Analysis of CDPN

Symbolic Representations: We use hedge automata (unbounded width tree automata) [8] to represent infinite sets of CDPN configurations. Let $M = (Act, P, \Gamma, \Delta)$ be a CDPN. An *M-tree automaton* is a tuple $\mathcal{A} = (Q, \delta, F)$, where Q is a set of states, F is the set of final states, and δ is a set of rules of either the form (1) $\gamma(q) \to q'$, where $\gamma \in \Gamma$, and $q, q' \in Q$, or (2) $p(L) \to q$, where L is a regular language over Q, $p \in P$, and $q \in Q$.

In order to define the language recognized by $\mathcal{A}$, we define a *move relation* $\to_\delta$ between terms over $P \cup \Gamma \cup Q$: for every two terms t and t', we have $t \to_\delta t'$ iff there exist a context C and a rule $r \in \delta$ such that $t = C[s]$, $t' = C[s']$, and (1) either $r = \gamma(q) \to q'$, $s = \gamma(q)$, and $s' = q'$, or (2) $r = p(L) \to q$, $s = p(q_1, \ldots, q_n)$, $q_1 \cdots q_n \in L$, and $s' = q$.

Let $\xrightarrow{*}_\delta$ denote the reflexive-transitive closure of $\to_\delta$. A term $t \in \mathcal{T}$ is accepted by $q \in Q$ if $t \xrightarrow{*}_\delta q$. Let $L_q^\delta = \{t \in \mathcal{T} \,:\, t \xrightarrow{*}_\delta q\}$. A term t is accepted by $\mathcal{A}$ if there exists a state $q \in F$ such that $t \xrightarrow{*}_\delta q$. Let $L(\mathcal{A})$ be the set of all terms accepted by $\mathcal{A}$.

[4] PAD extends PA by allowing rewrite rules of the form $A \cdot B \to t$

A straightforward adaptation of the proofs in [8] allows to show that:

Theorem 4. *The class of M-tree automata is closed under boolean operations. Moreover, the emptiness problem of M-tree automata is decidable.*

Computing pre^* Images: Let $M = (Act, P, \Gamma, \Delta)$ be a CDPN and let $\mathcal{A} = (Q, \delta, F)$ be an M-tree automaton. We present hereafter an algorithm that allows us to construct an M-tree automaton $\mathcal{A}_{\mathsf{pre}^*}$ recognizing the pre^*-image of $L(\mathcal{A})$. The construction proceeds (similarly to Section 4) by adding new transitions to the original automaton $\mathcal{A}$ corresponding to the backward application of transition rules. In order to deal with the constraints in the transition rules, we need to extend the original automaton.

Propagating Control States: Remember that, by definition of CDPN terms, the configuration of each process is encoded bottom-up in the tree (reading first the control state, and then the stack contents starting from its topmost symbol). Since constraints in CDPN transition rules refer to control states of the children processes, and since hedge automata can check only constraints on immediate successors in trees (which correspond in our case to the bottom symbols in the stacks of the children processes), we need to propagate upward the informations about the control states through the stacks. Therefore, the first step of our construction consists in defining a new automaton $\mathcal{A}_P = (Q_P, \delta_P, F_P)$ such that $L(\mathcal{A}_P) = L(\mathcal{A})$, and where states of Q are labelled by control states $p \in P$. This automaton is given by: $Q_P = Q \times P$, $F_P = F \times P$, and δ_P is the smallest set of rules such that:

- if $p(L) \to s \in \delta$, then $p(L') \to (s, p) \in \delta_P$, where L' is obtained by substituting in the words of L every occurrence of a state $s \in Q$ by $\{(s, p) \mid p \in P\}$;
- if $\gamma(s) \to s' \in \delta$, then for every $p \in P$, $\gamma\big((s, p)\big) \to (s', p) \in \delta_P$.

Lemma 2. *$L(\mathcal{A}_P) = L(\mathcal{A})$, and for every $t \in \mathcal{T}$, $t \xrightarrow{*}_{\delta_P} (s, p)$ iff $t \xrightarrow{*}_{\delta} s$ and $\mathcal{S}(t) = p$.*

Note 2. To avoid confusion, we use in the sequel $p, p', p_1, p_2, \ldots$ to denote elements of P, $s, s', s_1, s_2, \ldots,$ to denote states of $\mathcal{A}$, and $q, q', q_1, q_2, \ldots$ to denote states of $\mathcal{A}_P$.

From Constraints over P to Constraints over Q_P: Given a constraint ϕ and n terms $t_1, \ldots, t_n$ such that $t_i \xrightarrow{*}_{\delta_P} q_i$ for $1 \le i \le n$, we need also to be able to get the information whether $\mathcal{S}(t_1) \cdots \mathcal{S}(t_n) \in \phi$ from the states $q_1, \ldots, q_n$. For that, we associate with each constraint ϕ over P a constraint $\langle \phi \rangle$ over Q_P such that $\mathcal{S}(t_1) \cdots \mathcal{S}(t_n) \in \phi$ if and only if $q_1 \cdots q_n \in \langle \phi \rangle$. The definition of $\langle \phi \rangle$ is straightforward by induction on the structure of regular expressions for stable languages: (1) $\langle S \rangle = \{(s, p) : s \in Q, p \in S\}$, (2) $\langle \phi_1 \cdot \phi_2 \rangle = \langle \phi_1 \rangle \cdot \langle \phi_2 \rangle$, (3) $\langle \phi_1 + \phi_2 \rangle = \langle \phi_1 \rangle + \langle \phi_2 \rangle$, and (4) $\langle \phi^* \rangle = \langle \phi \rangle^*$.

Closed Set of Constraints: During the construction of the automaton, new transition rules of the form $p(L') \to q$ are added where L' are languages which are built from languages L appearing in the rules of the original automaton $\mathcal{A}$, and constraints ϕ appearing in the transition rules of the CDPN M, using intersection and right-quotient operations. Intersections $L \cap \langle \phi \rangle$ allow us to check that the guarding constraint for the application of a transition rule is satisfied at the considered position in the tree. Right-quotients $Lq^{-1} = \{w : wq \in L\}$ allow us to get immediate predecessors by a spawn operation of trees where the children of the spawning process are recognized by a sequence of states in L, and the youngest son among these children (i.e., the one created by the spawn operation and which is the right-most one in the list of children) is recognized by the state q. Then, let us define Λ to be the smallest family of languages over Q_P such that:

- If $(p(L) \to q) \in \delta_P$, then $L \in \Lambda$.
- If $L \in \Lambda$, and $(\phi : p\gamma \overset{a}{\hookrightarrow} p_1 w_1 \rhd p_2 w_2) \in \Delta$, then $L \cap \langle \phi \rangle \in \Lambda$.
- If $L \in \Lambda$ and $q \in Q_P$, then $Lq^{-1} \in \Lambda$.

Lemma 3. *The family Λ is finite. Assuming that all languages and constraints appearing in rules δ_P and Δ are given by backward-deterministic finite-state automata of size at most K, the number of elements of Λ is in $O(K^{n+1})$ where n is the number of different constraints appearing in the rules of Δ.*

Constructing $\mathcal{A}_{\mathsf{pre}^*}$: We define $\mathcal{A}_{\mathsf{pre}^*}$ to be the M-tree automaton (Q', δ', F') such that (1) $Q' = Q_P \cup \{q_p^L \; : \; p \in P, \, L \in \Lambda\}$, (2) $F' = F_P$, and (3) δ' is the smallest set of rules such that $\delta'_0 = \delta_P \cup \{p(L) \to q_p^L \; : \; p \in P, \, L \in \Lambda\} \subseteq \delta'$ and:

R_1: If $(\phi : p\gamma \overset{a}{\hookrightarrow} p'w) \in \Delta$, $p'(L) \to q \in \delta'_0$, and $w^R(q) \xrightarrow{*}_{\delta'} q'$, then $\big(\gamma(q_p^{L \cap \langle \phi \rangle}) \to q'\big) \in \delta'$.

R_2: If $(\phi : p\gamma \overset{a}{\hookrightarrow} p'w_1 \rhd p''w_2) \in \Delta$, $p'(L) \to q'' \in \delta'_0$, $w_1^R(q'') \xrightarrow{*}_{\delta'} q'$, and $w_2^R(p'') \xrightarrow{*}_{\delta'} q$,

then $\big(\gamma(q_p^{Lq^{-1} \cap \langle \phi \rangle}) \to q'\big) \in \delta'$.

Note that the states q_p^L, for $p \in P$, and $L \in \Lambda$, are added to the automaton in order to recognize precisely all the terms having p at the root and such that the sequence of children of the root is recognized by a sequence of states in the language L. Note also that all the transitions added by the construction are Γ-transitions, and therefore they do not add P-transitions to the automaton.

The set of rules δ' can be computed iteratively as the limit of an increasing sequence $\delta'_0 \subseteq \delta'_1 \cdots$ such that δ'_{i+1} contains at most one transition more than δ'_i added by applying either (R_1) or (R_2). Note that δ' is necessarily finite since (by Lemma 3) the number of triples (γ, q_p^L, q), for $\gamma \in \Gamma$, $p \in P$, $L \in \Lambda$, and $q \in Q'$ is finite.

Lemma 4. *For every $q \in Q_P$, $L_q^{\delta'} = \mathsf{pre}^*(L_q^{\delta_P})$.*

The lemma above says that the construction ensures that every state recognizes the set of all predecessors of its original language (i.e., in the automaton before saturation). Let us give some intuitive explanations about the role of the saturation rules, and let us consider the rule (R_1) (since the role of (R_2) is similar). Consider a term $w^R p'(t_1, \ldots, t_n)$ such that $t_i \xrightarrow{*}_{\delta'} q_i$, for $i \in \{1, \ldots, n\}$. Assume that $p'(L) \to q$ is a rule of the automaton. This means that after recognizing each of the terms t_i and labelling their roots by the states q_i, the automaton can label the term $p'(t_1, \ldots, t_n)$ by q if the sequence $q_1 \cdots q_n$ is in L. Assume furthermore that $w^R(q) \xrightarrow{*}_{\delta'} q'$. This means that the automaton can proceed by reading upward the word w and label the term $w^R p'(t_1, \ldots, t_n)$ by q'. Therefore, if $(\phi : p\gamma \overset{a}{\hookrightarrow} p'w)$ is a transition rule of the system, and if the sequence of control states $\mathcal{S}(t_1)\cdots\mathcal{S}(t_n)$ is in ϕ, then we must add the term $\gamma p(t_1,\ldots,t_n)$ (which is the immediate predecessor of $w^R p'(t_1,\ldots,t_n)$ by the transition rule) to the language of q' (to say that this term is a predecessor of some term which was recognized by q' in the original automaton). This is achieved by applying the saturation rule which adds to the automaton the transition $(\gamma(q_p^{L\cap\langle\phi\rangle}) \to q')$. The justification of this is in fact subtle. First, if $\mathcal{S}(t_1)\cdots\mathcal{S}(t_n) \in \phi$, we must have $q_1\cdots q_n \in \langle\phi\rangle$. Since states recognize predecessors of terms in their original language, each state q_i is a pair (s_i, p'_i) such that $p'_i = \mathcal{S}(t'_i)$ for some t'_i such that $t_i \in \mathsf{pre}^*(t'_i)$. Now, here is the point where the stability property of ϕ plays a crucial role: it ensures that backward transitions cannot make a term satisfy new constraints (or equivalently, that forward transitions cannot falsify a constraint). Therefore, since $\mathcal{S}(t_1)\cdots\mathcal{S}(t_n) \in \phi$, we must have also $\mathcal{S}(t'_1)\cdots\mathcal{S}(t'_n) \in \phi$, which implies that $q_1\cdots q_n \in \langle\phi\rangle$. On the other hand, assume that $\mathcal{S}(t_1)\cdots\mathcal{S}(t_n) \notin \phi$ but $q_1\cdots q_n \in \langle\phi\rangle$ because $\mathcal{S}(t'_1)\cdots\mathcal{S}(t'_n) \in \phi$. We can show that $\gamma p(t_1,\ldots,t_n)$ is actually in the pre^* image of the original language. Indeed, it is possible in this case to start by rewriting each term t_i to its successor t'_i, which makes the transition rule $(\phi : p\gamma \overset{a}{\hookrightarrow} p'w)$ applicable.

Theorem 5. *For every CDPN M, and for every M-tree automaton $\mathcal{A}$, we can construct an M-tree automaton $\mathcal{A}_{\mathsf{pre}^*}$ such that $L(\mathcal{A}_{\mathsf{pre}^*}) = \mathsf{pre}^*\big(L(\mathcal{A})\big)$.*

Note 3. It is easy to show that, given an M-tree automaton $\mathcal{A}$, the set $\mathsf{pre}_M(\mathcal{A})$ (and in fact also the set $\mathsf{post}_M(\mathcal{A})$) is an effectively M-tree automata definable set.

Then, based on the modelling described in Sections 3 and 5, we can apply Theorems 5 and 4 to check reachability properties and solve flow analysis problems (such as bitvector problems) for multithreaded programs.

Complexity Issues: By Lemma 3, we know that the size of the automaton $\mathcal{A}_{\mathsf{pre}^*}$ is at most exponential in the number of constraints appearing in the given CDPN. In fact, we can prove the following PSPACE lower bound by a reduction of the satisfiability problem for quantified Boolean formulas (QBF).

Theorem 6. *It is at least PSPACE-hard to decide for a given CDPN M, a regular set of M-configurations R and an M-configuration c, whether $c \in \mathsf{pre}^*(R)$ or not.*

Despite the hardness result above, in many interesting cases, we only need a *fixed* number of constraints, which leads to polynomial analysis algorithms. For instance, this is the case when only trivial constraints (i.e., of the form P^*) are used, which corresponds to the case of DPN models. Also, to model parallel calls only one additional constraint is needed, namely $P^*\natural^2$, as we have seen in Section 5. Similarly, we only need one additional constraint for each type of join statement such as $\mathsf{join}_\forall$ or $\mathsf{join}_{\exists k}$. Note that the automata for these constraints can easily be defined by backward deterministic automata of very small sizes. Also for typical properties such as bitvector problems (see Section 3), the initial automaton is always the one recognizing the set of all configurations. Therefore, for an important fragment of CDPN which subsumes (in modelling power) existing formalisms such as PA and PAD, and allows us in addition to model spawn operations, our construction leads to a polynomial analysis algorithm.

However, when return values from parallel processes are taken into account, our construction becomes exponential in the number of used abstract data values. This price is unavoidable since dealing with an unfixed domain of return values is precisely the feature which makes our model complex (see the proof of Theorem 6). Such complexity does not appear for weaker models such as PA or PAD (which have polynomial analysis algorithms [13, 10, 6]) since they cannot handle return values from parallel processes.

Relaxing Stability: We end this section by mentioning the fact that relaxing the stability condition on the constraints appearing in the transition rules of CDPN leads to a model for which pre^* images are not regular in general.

Theorem 7. *There exists a CDPN M with nonstable constraints, and a regular set T of M-configurations such that $\mathsf{pre}^*_M(T)$ is not definable by an M-tree automaton.*

Actually, we can define M s.t. all its transition rules are of the form $\phi : p\gamma \hookrightarrow p'\gamma$ (i.e., without stack manipulation and dynamic creation of processes), and where ϕ is of the simple form pP^*, for $p \in P$. This shows that it is hard to relax the stability condition in the definition of CDPN without losing the property that pre^* preserves regularity.

7 Conclusion

We have defined new formalisms (DPN and CDPN), based on word/term rewrite systems, allowing to model adequately spawn-like commands in multithreaded programs. We have shown that (1) they are more suitable for modelling these commands than previously proposed formalisms (such as PA and PAD), and that (2) they subsume in fact in modelling power these models (concerning CDPN), and allow to handle features these models cannot handle such as return values from parallel processes, various join commands, etc.

We have defined automata-based techniques for computing backward reachability sets of our models. In the case of the basic model of DPN, word automata can be used for this purpose and the construction is simple. In the case of CDPN where constraints on the children are used, the problem of reachability analysis becomes much more delicate. The condition of stability we impose in CDPN on the constraints (guards) appearing in the transition rules seems to be necessary in order to have regular backward reachability sets. Concerning complexity, our construction is exponential in the number of different constraints used in the model, but significant classes of parallel programs can be modelled using a fixed number of constraints (often representable using small automata), and therefore they can be analysed in polynomial time.

Future work includes the extension of our models and our approach to handle synchronisation between parallel processes. Of course, the reachability analysis becomes undecidable in general, but reasonable classes of programs with particular synchronisation policies can be considered (see e.g., [18]), and generic frameworks for defining abstractions (and refining them) can be developed based on our models and our techniques, e.g., following the approaches of [3, 4, 15]. We think also that our techniques could be used to handle models which extend those considered in this paper by allowing a bounded number of context switches, in the spirit of the approach of [19].

References

1. J. Baeten and W. Weijland. Process Algebra. In *Cambridge Tracts in Theoretical Computer Science*, volume 18, 1990.
2. A. Bouajjani, J. Esparza, and O. Maler. Reachability Analysis of Pushdown Automata: Application to Model Checking. In *CONCUR'97*. LNCS 1243, 1997.
3. A. Bouajjani, J. Esparza, and T. Touili. A Generic Approach to the Static Analysis of Concurrent Programs with Procedures. In *POPL'03*. ACM, 2003.
4. A. Bouajjani, J. Esparza, and T. Touili. Reachability Analysis of Synchronised PA systems. In *INFINITY'04*. to appear in ENTCS, 2004.
5. A. Bouajjani, M. Müller-Olm, and T. Touili. Regular Symbolic Analysis of Dynamic Networks of Pushdown Processes. Technical report, LIAFA lab No 2005-05, and University of Dortmund No 798, June 2005.
6. A. Bouajjani and T. Touili. Reachability Analysis of Process Rewrite Systems. In *FSTTCS'03*. LNCS 2914, 2003.
7. A. Bouajjani and T. Touili. On Computing Reachability Sets of Process Rewrite Systems. In *RTA'05*. LNCS, 2005.
8. A. Bruggemann-Klein, M. Murata, and D. Wood. Regular Tree and Regular Hedge Languages over Unranked Alphabets. Research report, 2001.
9. J. Esparza and J. Knoop. An Automata-Theoretic Approach to Interprocedural Data-Flow Analysis. In *FoSSaCS'99*, volume 1578 of *LNCS*, 1999.
10. J. Esparza and A. Podelski. Efficient Algorithms for pre^* and post^* on Interprocedural Parallel Flow Graphs. In *POPL'00*. ACM, 2000.
11. A. Finkel, B. Willems, and P. Wolper. A Direct Symbolic Approach to Model Checking Pushdown Systems. In *Infinity'97, ENTCS 9*. Elsevier Sci. Pub., 1997.
12. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.
13. D. Lugiez and P. Schnoebelen. The Regular Viewpoint on PA-Processes. *Theoretical Computer Science*, 274(1-2):89–115, 2002.
14. R. Mayr. Decidability and Complexity of Model Checking Problems for Infinite-State Systems. Phd. thesis, Technical University Munich, 1998.
15. M. Müller-Olm. Variations on Constants. Habilitationsschrift, Fachbereich Informatik, Universität Dortmund, 2002.
16. M. Müller-Olm. Precise Interprocedural Dependence Analysis of Parallel Programs. *Theoretical Computer Science*, 311:325–388, 2004.
17. C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
18. S. Qadeer, S. Rajamani, and J. Rehof. Procedure Summaries for Model Checking Multithreaded Software. In *POPL'04*, 2004.
19. S. Qadeer and J. Rehof. Context-Bounded Model-Checking of Concurrent Software. In *TACAS'05*. LNCS 3440, 2005.
20. H. Seidl and B. Steffen. Constraint-based Interprocedural Analysis of Parallel Programs. In *ESOP'2000*. LNCS 1782, 2000.
21. T. Touili. Dealing with Communication for Dynamic Multithreaded Recursive Programs. In *1st VISSAS workshop*, March 2005. Invited Paper.