Tree Automata for Analyzing Dynamic Pushdown Networks

Peter Lammich

Institut für Informatik, Fachbereich Mathematik und Informatik
Westfälische Wilhelms-Universität Münster
peter.lammich@uni-muenster.de

Abstract. Dynamic Pushdown Networks (DPNs) are an abstract model for concurrent programs with recursive procedures and dynamic process creation. Usually, DPNs are described with an interleaving semantics, where an execution is a sequence of steps. Recently, we introduced a true-concurrency semantics for DPNs, where executions are trees.
The standard analysis methods for DPNs are based on a saturation algorithm, that, given a set of configurations, computes the set of all predecessor configurations. In this paper we present an alternative analysis algorithm that is based on tree automata. DPN executions as well as the properties to be analyzed are represented as tree automata, and the analysis is done by standard tree-automata algorithms for intersection and emptiness check.

1 Introduction

Many existing analysis techniques for concurrent programs use parallel pushdown automata (e.g. [5, 3, 4]) or parallel procedure calls (e.g. [10, 8]) as a model for concurrency. However, there are programming languages, like Java, that support dynamic thread creation. This cannot be mapped to parallel procedure calls [2], and it can only be mapped to parallel pushdown systems when bounding the maximum number of allowed threads. In contrast, dynamic pushdown networks (DPNs)[2] natively support dynamic thread creation.

DPNs extend pushdown systems by rules with the side-effect of creating a new pushdown process that is then executed concurrently. In [2], analysis of DPNs is done by automata based techniques. The key result for the analysis of DPNs is, that pre^*_M-computation preserves regularity. This can be used for, e.g., bitvector kill/gen analysis [2] or bounded model checking of DPNs with shared memory [1].

Originally, DPNs are given an interleaving semantics, where an execution is a totally ordered sequence of steps. More recently, a true concurrency semantics for DPNs has been introduced [7], where executions are modeled as trees. It is shown how to decide tree-regular properties of such executions, and thus generalize the results on DPNs of [2]. Moreover, this technique can be used to precisely compute pre^*_M-sets for DPNs with well-nested locks [7].

The techniques presented in [7] compute the cross-product of a DPN and a tree automaton for the regular property to be analyzed. This cross-product is, again, a DPN and can be analyzed using the standard pre^*_M-computation [2]. In this paper, we explore a different approach for the analysis of DPNs. We also use the true-concurrency semantics presented in [7], but instead of computing the cross-product DPN, we use tree-automata techniques: Let M be a DPN with the set of executions e_M, and Φ be a regular property. We want to check whether the DPN has an execution with property Φ, i.e. whether $e_M \cap \Phi \neq \emptyset$. Our analysis computes a tree-automaton A with language $L(A)$ from the DPN, such that $e_M = \alpha(L(A))$, where α maps a tree of $L(A)$, called *regular execution tree*, to an execution of the true-concurrency semantics, and $\alpha(L(A))$ is the element-wise application of α to the set $L(A)$. The problem now reduces to checking whether $L(A) \cap \alpha^{-1}(\Phi) \neq \emptyset$. If $\alpha^{-1}(\Phi)$ is also a regular set, this can be decided using standard tree automata algorithms.

In this paper, we manually derive the tree automata representation of $\alpha^{-1}(\Phi)$ for a specific reachability property Φ. It is left future work to automatically derive the tree automata representations of $\alpha^{-1}(\Phi)$ from a suitable representation of Φ, e.g. using theory related to macro tree-transducers.

The rest of this paper is organized as follows: In Section 2, we introduce DPNs along with their interleaving and true-concurrency semantics. In Section 3, we introduce regular execution trees, justify them w.r.t. the true-concurrency semantics and derive a specific reachability property for regular execution trees. Finally, in Section 4, we give a short conclusion and outlook to future work.

2 Dynamic Pushdown Networks

A DPN [2] is a tuple $M = (P, \Gamma, L, \Delta)$, where P is a finite set of control states, Γ is a finite stack alphabet with $P \cap \Gamma = \emptyset$, L is a finite set of rule labels, and $\Delta = \Delta_N \cup \Delta_S$ is a finite set of *non-spawning* (Δ_N) and *spawning* rules (Δ_S). A non-spawing rule $p\gamma \overset{l}{\hookrightarrow} p'w \in \Delta_N \subseteq P\Gamma \times L \times P\Gamma^*$ enables a transition on a single pushdown process with state p and top of stack γ to new state p' and new top of stack $w \in \Gamma^*$. A spawning rule $p\gamma \overset{l}{\hookrightarrow} p'w \rhd p_s w_s \in \Delta_S \subseteq P\Gamma \times L \times P\Gamma^* \times P\Gamma^*$ is a pushdown rule with the additional side-effect of creating a new process with initial state p_s and initial stack w_s. DPNs are able to model programs with potentially recursive procedures: A rule with $|w| = 0$ models a return from a procedure, a rule with $|w| = 1$ models a step inside a procedure, and a rule with $|w| = 2$ models a procedure call. Rules with $|w| > 2$ can be seen as generalized procedure calls where more than one return address is pushed onto the stack. In this paper, we restrict DPN rules to be of one of the following forms:

Assumption 1 *Let $r \in \Delta$ be a DPN-rule. Then there exists $p, p', p_s \in P$, $\gamma, \gamma_1, \gamma_2, \gamma', \gamma_s \in \Gamma$ and $l \in L$ such that r has one of the following forms:*

basic step $r = p\gamma \overset{l}{\hookrightarrow} p'\gamma'$

procedure call $r = p\gamma \overset{l}{\hookrightarrow} p'\gamma_1\gamma_2$
procedure return $r = p\gamma \overset{l}{\hookrightarrow} p'$
process creation $r = p\gamma \overset{l}{\hookrightarrow} p'\gamma' \triangleright p_s\gamma_s$

Note that these rule types are sufficient to model interprocedural programs with DPNs. Moreover, by replacing generalized calls by sequences of procedure calls (adding new states and stack symbols as necessary), one can transform any DPN into a DPN that satisfies Assumption 1, while preserving the relevant properties for program analysis, e.g. reachability properties. For the rest of this paper, we assume that we have a fixed DPN $M = (P, \Gamma, L, \Delta)$, and a fixed initial configuration $p_0\gamma_0 \in P\Gamma$. Note that this restricts our analysis to programs with one initial process. If programs with more than one initial process are required, one has to create a process that sets up the program's initial configuration.

Interleaving Semantics. We briefly recall the interleaving semantics of a DPN as presented in [2]: Configurations $\mathsf{Conf} := (P\Gamma^*)^*$ are sequences of words from $P\Gamma^*$, each word containing the control state and stack of one of the processes running in parallel. The step relation $\rightarrow \subseteq \mathsf{Conf} \times L \times \mathsf{Conf}$ is the least solution of the following constraints:

$$\begin{array}{lll} \text{[nospawn]} & c_1(p\gamma r)c_2 \xrightarrow{l} c_1(p'wr)c_2 & \text{if } p\gamma \overset{l}{\hookrightarrow} p'w \in \Delta_N \\ \text{[spawn]} & c_1(p\gamma r)c_2 \xrightarrow{l} c_1(p_s w_s)(p'wr)c_2 & \text{if } p\gamma \overset{l}{\hookrightarrow} p'w \triangleright p_s w_s \in \Delta_S \end{array}$$

A [nospawn]-step corresponds precisely to a pushdown operation (manipulating the control state and the top of the stack), a [spawn]-step additionally creates a new process that is inserted to the left of the creating process. We define $\rightarrow^* \subseteq \mathsf{Conf} \times L^* \times \mathsf{Conf}$ to be the reflexive, transitive closure of $\rightarrow$. This semantics is an *interleaving semantics*, because steps of processes running in parallel are interleaved. It models the possible executions on a single processor, where preemption may occur after any step.

In order to simplify the presentation of the analysis, we assume that no configuration with an empty stack is reachable from the initial process $p_0\gamma_0$:

Assumption 2 $\forall l \in L,\ c' \in \mathsf{Conf}.\ p_0\gamma_0 \xrightarrow{l}{}^{*} c' \implies c' \in (P\Gamma^+)^*$

Again, any DPN can be transformed to satisfy this assumption while preserving the relevant properties. The transformation adds a new stack symbol $\gamma_\perp$ at the bottom of each stack.

Also note that a DPN contains no rules to remove an existing process from the configuration. Thus we have:

Lemma 3. $\forall pw \in P\Gamma^*,\ l \in L,\ c' \in \mathsf{Conf}.\ p\gamma \xrightarrow{l}{}^{*} c' \implies c' \in (P\Gamma^*)^+$

In DPNs, termination of a process can be modeled, e.g., by going into a special state from that there is no more progress.

Predecessor Computation. Given a set $C' \subseteq \mathsf{Conf}$ of configurations, the set $\mathsf{pre}_M(C') := \{c \mid \exists c' \in C',\ l \in L.\ c \xrightarrow{l} c'\}$ is the set of *immediate predecessors* of C', i.e. the set of configurations that can make a transition to a $c' \in C'$ in exactly one step. Similarly, $\mathsf{pre}^*_M(C') := \{c \mid \exists c' \in C',\ \bar{l} \in L^*.\ c \xrightarrow{\bar{l}}{}^* c'\}$ is the set of *predecessors* of C', i.e. the set of configurations that can make a transition to a $c' \in C'$ by executing an arbitrary number of steps.

An important result on DPNs is that pre_M and pre^*_M preserve regularity, i.e. if C' is a regular set then $\mathsf{pre}_M(C')$ and $\mathsf{pre}^*_M(C')$ are regular as well, and given an automaton accepting C', automata accepting $\mathsf{pre}_M(C')$ and $\mathsf{pre}^*_M(C')$, respectively, can be computed in polynomial time [2]. This result is the key to analysis of DPNs. For example, let $p\gamma \in P\Gamma$ be a state where a resource is write-accessed. Let $p'\gamma' \in P\Gamma$ be another state where the same resource is read-accessed. The regular set

$$C := (S^* p\gamma \Gamma^* S^* p'\gamma' \Gamma^* S^*) \mid (S^* p'\gamma' \Gamma^* S^* p\gamma \Gamma^* S^*)$$

with $S = P\Gamma^*$ contains exactly those configurations where one process is at state $p\gamma$ and another process is at state $p'\gamma'$. Thus, the query $p_0\gamma_0 \in \mathsf{pre}^*_M(C)$ decides whether a conflict situation between the two resource accesses at $p\gamma$ and $p'\gamma'$ is reachable[1].

Tree Semantics. We recently presented a tree-based semantics for DPNs [7]. An execution of the interleaving semantics is a sequence of steps, inducing a total ordering on the steps. Even steps of independent processes are ordered. In the tree-based semantics, an execution is a tree of steps, only inducing an ordering on steps of the same process and on steps of a created process to come after the step that created the process.

Formally, we model an execution starting at a single process as an *execution tree* of type $T ::= \mathsf{N}\ L\ T \mid \mathsf{S}\ L\ T\ T \mid \mathsf{L}\ P\Gamma^*$. A tree of the form $\mathsf{N}\ l\ t$ models an execution that performs the non-spawning step l first, followed by the execution described by t. A tree of the form $\mathsf{S}\ l\ t_s\ t$ models an execution that performs the spawning step l first, followed by the execution of the spawned process described by t_s and the remaining execution of the spawning process described by t. A node of the form $\mathsf{L}\ pw$ indicates that the process makes no more steps and that its final configuration is pw. The annotation of the reached configuration at the leafs of the execution tree increases expressiveness of regular sets of execution trees, e.g. one can characterize execution trees that reach certain control states. The distinction between spawned and spawning tree at S-nodes allows for keeping track of which steps belong to which process, e.g. when tracking the acquired locks of a process, as done in [7].

[1] Note that the regular sets are not required to only model valid configurations, thus we could also use the set $C = (S^* p\gamma S^* p'\gamma' S^*) \mid (S^* p'\gamma' S^* p\gamma S^*)$ with $S = P \cup \Gamma$. It has a simpler automata but does not satisfy $C \subseteq \mathsf{Conf}$.

The relation $\Longrightarrow \subseteq P\Gamma^* \times T \times \mathsf{Conf}$ characterizes the execution trees starting at a single process. It is defined as the least solution of the following constraints:

$$\begin{array}{lll}
\text{[leaf]} & qw \overset{\mathsf{L}\ qw}{\Longrightarrow} qw & \\
\text{[nospawn]} & q\gamma r \overset{\mathsf{N}\ l\ t}{\Longrightarrow} c' & \text{if } q\gamma \overset{l}{\hookrightarrow} q'w \in \Delta_N \ \wedge\ q'wr \overset{t}{\Longrightarrow} c' \\
\text{[spawn]} & q\gamma r \overset{\mathsf{S}\ l\ t_s\ t}{\Longrightarrow} c_s c' & \text{if } q\gamma \overset{l}{\hookrightarrow} q'w \rhd q_s w_s \in \Delta_S \\
 & & \quad \wedge\ q_s w_s \overset{t_s}{\Longrightarrow} c_s \ \wedge\ q'wr \overset{t}{\Longrightarrow} c'
\end{array}$$

In order to relate the tree semantics to the interleaving semantics, we define a scheduler that maps execution trees to compatible sequences of rules. From the ordering point of view, the scheduler maps the steps ordered by the execution tree to the set of its topological sorts. The scheduler is modeled as a labeled transition system over lists of execution trees. A step replaces the root node of a tree in the list by its successors. Formally, the scheduler $\rightsquigarrow \subseteq T^* \times L \times T^*$ is the least relation satisfying the following constraints:

$$\begin{array}{ll}
\text{[nospawn]} & h_1(\mathsf{N}\ l\ t)h_2 \overset{l}{\rightsquigarrow} h_1 t h_2 \\
\text{[spawn]} & h_1(\mathsf{S}\ l\ t_s\ t)h_2 \overset{l}{\rightsquigarrow} h_1 t_s t h_2
\end{array}$$

We call $\mathsf{sched}(t) := \{\bar{l} \in L^* \mid \exists h' \in (\mathsf{L}\ P\Gamma^*)^*.\ [t] \overset{\bar{l}}{\rightsquigarrow}{}^* h'\}$ the set of *schedules* of a tree $t \in T$, where $(\mathsf{L}\ P\Gamma^*)^*$ is the set of all lists of L-nodes, and $\rightsquigarrow^*$ is the reflexive, transitive closure of the scheduler $\rightsquigarrow$. Notice that this definition of the scheduler corresponds to the well-known topological sorting algorithm that iteratively removes minimal elements (in this case root nodes of trees in the list) until there are no more minimal elements left. It can be shown by induction[2] that every execution of the interleaving semantics is a schedule of an execution of the tree semantics and vice versa:

Theorem 4. *Let $p \in P$, $w \in \Gamma^*$, $c' \in \mathsf{Conf}$ and $\bar{l} \in L^*$, then $pw \overset{\bar{l}}{\rightarrow}{}^* c'$ if and only if there is an execution tree $t \in T$ with $pw \overset{t}{\Longrightarrow} c'$ and $\bar{l} \in \mathsf{sched}(t)$.*

Moreover, as trees are acyclic and thus any tree has at least one topological sort, any execution tree has at least one schedule:

Lemma 5. $\forall t \in T.\ \mathsf{sched}(t) \neq \emptyset$

Reached Configuration. The execution tree already determines the configuration that is reached by the execution, as the reached configuration is annotated at the leafs of the execution tree. We define the function $\mathsf{conf} : T \rightarrow \mathsf{Conf}$ that returns the configuration reached by an execution tree:

$$\begin{array}{l}
\mathsf{conf}(\mathsf{L}\ pw) = pw \\
\mathsf{conf}(\mathsf{N}\ l\ t) = \mathsf{conf}(t) \\
\mathsf{conf}(\mathsf{S}\ l\ t_s\ t) = \mathsf{conf}(t_s)\mathsf{conf}(t)
\end{array}$$

[2] To get the induction through, one has to generalize the $\overset{\cdot}{\Longrightarrow}$-relation to more than one initial process and lists of execution trees, as done in [7, 6].

It is straightforward to show that $\mathsf{conf}(t)$ really returns the configuration reached by any execution of t:

Lemma 6. $\forall pw \in P\Gamma^*,\ t \in T, c' \in \mathsf{Conf}.\ pw \overset{t}{\Longrightarrow} c' \quad \Longrightarrow \quad c' = \mathsf{conf}(t)$

3 Regular Execution Trees

An execution tree makes the structure of process creation explicitely visible. However, the structure of procedure calls and matching returns is not explicitely visible in the structure of the execution tree. The information whether a procedure call returns and where the matching return node is, can only be obtained by inspecting the execution of the process and searching for a matching return node. In this section, we develop another tree-based representation of executions, so called *regular execution trees*. The structure of those execution trees makes visible both, process creation and procedure calls. For the remainder of this paper, we will use the term *standard execution tree* to refer to the execution trees introduced above. Regular execution trees have the advantage that the set of executions starting at a configuration of the form $p\gamma$ (in particular $p_0\gamma_0$) can be described as a regular set, and it is straightforward to construct a tree-automaton for this set from the rules of the DPN. Thus, we can use standard tree-automata operations like intersection and emptiness check for the analysis. We distinguish between the set R_r of returning execution trees and the set R_n of non-returning execution trees. A returning execution tree returns from the current procedure, while a non-returning execution tree does not. Formally, regular execution trees have the type:

$$R_r ::= \mathsf{ret}\ L\ P \mid \mathsf{base}\ L\ R_r \mid \mathsf{callr}\ L\ R_r\ R_r \mid \mathsf{spawn}\ L\ R_n\ R_r$$
$$R_n ::= \mathsf{nil}\ P\Gamma \mid \mathsf{base}\ L\ R_n \mid \mathsf{callr}\ L\ R_r\ R_n \mid \mathsf{calln}\ L\ R_n\ \Gamma \mid \mathsf{spawn}\ L\ R_n\ R_n$$

Moreover, we define the set R of all regular execution trees by $R := R_r \cup R_n$. Intuitively, a nil $p\gamma$-node represents the end of a process's execution. The process ends in state p with top-of-stack γ. A ret l p-node represents a procedure return. The return step is labeled with l, and the return state is p. A base l τ-node represents a basic-step with label l. After the basic step, the execution continues with τ. A callr l τ_c τ-node represents a returning procedure call labeled with l. The execution of the procedure is described by τ_c and the execution after the procedure has returned is described by τ. A calln l τ_c γ-node represents a procedure call that does not return. τ_c represents the execution of the procedure, and γ is the return address of the procedure, that will – as the procedure does not return – remain on the stack for the rest of the execution. A spawn l τ_s τ-node represents a process creation step, where τ_s is the execution of the created process and τ is the remaining execution of the creating process. Note that, due to Assumption 2, a (reachable) spawned process does not return from its initial procedure, and thus we have $\tau_s \in R_n$.

In order to map from a regular execution tree to a standard execution tree, we need to define a concatenation operation that glues together two standard execution trees, by replacing the rightmost leaf of the first tree by the second tree. We define the operation $\cdot;\cdot : T \times T \to T$ recursively over the structure of the first tree:

$$\begin{aligned}
&(\mathsf{L}\ pw);t' = t'\\
&(\mathsf{N}\ l\ t);t' = \mathsf{N}\ l\ (t;t')\\
&(\mathsf{S}\ l\ t_s\ t);t' = \mathsf{S}\ l\ t_s\ (t;t')
\end{aligned}$$

In order to define the function $\alpha : R \to T$ that maps from regular execution trees to standard execution trees, we first define the auxiliary function $\alpha' : R \times \Gamma^* \to T$ that maps a regular execution tree and some stack to a standard execution tree. Intuitively, the stack contains the symbols that will not be popped during the rest of the execution.

$$\begin{aligned}
&\alpha'(\mathsf{nil}\ p\gamma, s) = \mathsf{L}p(\gamma s)\\
&\alpha'(\mathsf{ret}\ l\ p, s) = \mathsf{N}\ l\ (\mathsf{L}\ ps)\\
&\alpha'(\mathsf{base}\ l\ \tau, s) = \mathsf{N}\ l\ (\alpha'(\tau), s)\\
&\alpha'(\mathsf{callr}\ l\ \tau_c\ \tau, s) = \mathsf{N}\ l\ (\alpha'(\tau_c, \varepsilon); \alpha'(\tau, s))\\
&\alpha'(\mathsf{calln}\ l\ \tau_c\gamma, s) = \mathsf{N}\ l\ \alpha'(\tau_c, \gamma s)\\
&\alpha'(\mathsf{spawn}\ l\ \tau_s\ \tau, s) = \mathsf{S}\ l\ \alpha'(\tau_s, \varepsilon)\ \alpha'(\tau, s)
\end{aligned}$$

The function α is then defined as $\alpha(\tau) := \alpha'(\tau, \varepsilon)$. We overload $\alpha : 2^R \to 2^T$ for sets of trees by element-wise function application, i.e. $\alpha(X) = \{\alpha(\tau) \mid \tau \in X\}$.

We now characterize the regular execution trees of a DPN by the least fixed point of a constraint system. The constraint system contains variables of the form $N[p,\gamma] \subseteq R_n$ and $R[p,\gamma,q] \subseteq R_r$ for $p, q \in P$ and $\gamma \in \Gamma$. Intuitively, $N[p,\gamma]$ contains the set of non-returning execution trees starting at configuration $p\gamma$ and $R[p,\gamma,q]$ contains the set of execution trees starting at configuration $p\gamma$ and returning with state q:

$$\begin{array}{lll}
\quad[\text{n-nil}] & \mathsf{nil}\ p\gamma \in N[p,\gamma] & \\
\multicolumn{3}{l}{\text{for } p\gamma \overset{l}{\hookrightarrow} p' \in \Delta :}\\
\quad[\text{r-ret}] & \mathsf{ret}\ l\ p' \in R[p,\gamma,p'] & \\
\multicolumn{3}{l}{\text{for } p\gamma \overset{l}{\hookrightarrow} p'\gamma' \in \Delta,\ \tilde{p} \in P :}\\
\quad[\text{n-base}] & \mathsf{base}\ l\ \tau \in N[p,\gamma] & \Leftarrow \tau \in N[p',\gamma']\\
\quad[\text{r-base}] & \mathsf{base}\ l\ \tau \in R[p,\gamma,\tilde{p}] & \Leftarrow \tau \in R[p',\gamma',\tilde{p}]\\
\multicolumn{3}{l}{\text{for } p\gamma \overset{l}{\hookrightarrow} p'\gamma_1\gamma_2 \in \Delta,\ \tilde{p},\hat{p} \in P :}\\
\quad[\text{n-calln}] & \mathsf{calln}\ l\ \tau\ \gamma_2 \in N[p,\gamma] & \Leftarrow \tau \in N[p',\gamma_1]\\
\quad[\text{n-callr}] & \mathsf{callr}\ l\ \tau_c\ \tau \in N[p,\gamma] & \Leftarrow \tau_c \in R[p',\gamma_1,\hat{p}] \wedge \tau \in N[\hat{p},\gamma_2]\\
\quad[\text{r-callr}] & \mathsf{callr}\ l\ \tau_c\ \tau \in R[p,\gamma,\tilde{p}] & \Leftarrow \tau_c \in R[p',\gamma_1,\hat{p}] \wedge \tau \in R[\hat{p},\gamma_2,\tilde{p}]\\
\multicolumn{3}{l}{\text{for } p\gamma \overset{l}{\hookrightarrow} p'\gamma' \rhd p_s\gamma_s \in \Delta,\ \tilde{p} \in P :}\\
\quad[\text{n-spawn}] & \mathsf{spawn}\ l\ \tau_s\ \tau \in N[p,\gamma] & \Leftarrow \tau_s \in N[p_s,\gamma_s] \wedge \tau \in N[p',\gamma']\\
\quad[\text{r-spawn}] & \mathsf{spawn}\ l\ \tau_s\ \tau \in R[p,\gamma,\tilde{p}] & \Leftarrow \tau_s \in N[p_s,\gamma_s] \wedge \tau \in R[p',\gamma',\tilde{p}]
\end{array}$$

In the remainder, we use $N[p,\gamma]$ and $R[p,\gamma,q]$ to refer to the least solution of this constraint system. Note that these constraints also define rules of a tree automaton over states $\{N[p,\gamma] \mid p \in P, \gamma \in \Gamma\} \cup \{R[p,\gamma,q] \mid p,q \in P, \gamma \in \Gamma\}$.

Thus, the sets $N[p,\gamma]$ and $R[p,\gamma,q]$ are tree regular sets. We will use this tree-automata view and the closedness properties of regular tree languages in order to decide whether a DPN has some execution that satisfies a given tree-regular property $\Phi \subseteq R$, as this is equivalent to $N[p_0,\gamma_0] \cap \Phi \neq \emptyset$ which can be decided by standard tree automata operations.

First, we justify the regular execution tree semantics defined by the constraint system w.r.t. the standard execution tree semantics:

Theorem 7. *Let $p, p' \in P$ and $\gamma \in \Gamma$. Then we have:*

$$\begin{array}{rcl} a) \quad \{t \mid \exists c' \in (P\Gamma^+)^+.\ p\gamma \overset{t}{\Longrightarrow} c'\} & = & \alpha(N[p,\gamma]) \\ b)\ \{t \mid \exists c_s \in (P\Gamma^+)^*.\ p\gamma \overset{t}{\Longrightarrow} c_s p'\} & = & \alpha(R[p,\gamma,p']) \end{array}$$

Intuitively, Theorem 7a states that there is a standard execution t, starting at configuration $p\gamma$ and ending in a configuration with no empty stack ($c' \in (P\Gamma^+)^+$), if and only if there is a regular execution tree $\tau \in N[p,\gamma]$ that matches the standard execution tree t, i.e. ($\alpha(\tau) = t$). Theorem 7b makes the analog statement for returning executions.

Reached Configuration. We now extract the configuration reached by a regular execution tree: We overload the function $\mathsf{conf} : R \to \mathsf{Conf}$:

$$\begin{array}{l} \mathsf{conf}(\mathsf{nil}\ p\ \gamma) = p\gamma \\ \mathsf{conf}(\mathsf{ret}\ l\ p) = \varepsilon \\ \mathsf{conf}(\mathsf{base}\ l\ \tau) = \mathsf{conf}(\tau) \\ \mathsf{conf}(\mathsf{callr}\ l\ \tau_c\ \tau) = \mathsf{conf}(\tau_c)\mathsf{conf}(\tau) \\ \mathsf{conf}(\mathsf{calln}\ l\ \tau_c\ \gamma) = \mathsf{conf}(\tau_c)\gamma \\ \mathsf{conf}(\mathsf{spawn}\ l\ \tau_s\ \tau) = \mathsf{conf}(\tau_s)\mathsf{conf}(\tau) \end{array}$$

It is straightforward to show that, for non-returning execution trees, the configuration computed by conf matches the configuration reached by the corresponding standard execution tree:

Lemma 8. $\forall \tau \in R_n.\ \mathsf{conf}(\tau) = \mathsf{conf}(\alpha(\tau))$

In the rest of this paper, we want to characterize the set of regular execution trees that reach a configuration that is in a given regular set of configurations. For this purpose, we regard a finite state machine (FSM) $F = (Q, q_0, Q_F, \delta)$ with states Q, initial state $q_0 \in Q$, final states $Q_F \subseteq Q$ and transition relation $\delta \subseteq Q \times (P \cup \Gamma) \times Q$. Let δ^* be the reflexive, transitive closure of δ. The language of F is defined by $L(F) := \{c \in (P \cup \Gamma)^* \mid \exists q' \in Q_F.\ (q_0, c, q') \in \delta^*\}$. We regard a constraint system over variables $T[q,q'] \subseteq R$ for $q, q' \in Q$. Intuitively, $T[q,q']$ contains all regular execution trees whose reached configuration drives the FSM from state q to state q':

$$\begin{array}{lll} [\text{t-nil}] & \mathsf{nil}\ p\gamma \in T[q,q'] & \Leftarrow (q, p\gamma, q') \in \delta^* \\ [\text{t-ret}] & \mathsf{ret}\ l\ p \in T[q,q] & \Leftarrow q \in Q \\ [\text{t-base}] & \mathsf{base}\ l\ \tau \in T[q,q'] & \Leftarrow \tau \in T[q,q'] \\ [\text{t-callr}] & \mathsf{callr}\ l\ \tau_c\ \tau \in T[q,q'] & \Leftarrow \exists \hat{q}.\ \tau_c \in T[q,\hat{q}] \wedge \tau \in T[\hat{q},q'] \\ [\text{t-calln}] & \mathsf{calln}\ l\ \tau_c\ \gamma \in T[q,q'] & \Leftarrow \exists \hat{q}.\ \tau_c \in T[q,\hat{q}] \wedge (\hat{q},\gamma,q') \in \delta \\ [\text{t-spawn}] & \mathsf{spawn}\ l\ \tau_s\ \tau \in T[q,q'] & \Leftarrow \exists \hat{q}.\ \tau_s \in T[q,\hat{q}] \wedge \tau \in T[\hat{q},q'] \end{array}$$

In the rest of this paper we use $T[q, q']$ to refer to the least solution of this constraint system. Note that this constraint system also defines the rules of a tree automaton, and thus the sets $T[q, q']$ are regular. It is straightforward to show that $T[q, q']$ contains exactly those regular execution trees whose configuration transfers the FSM from state q to state q':

Theorem 9. $\forall \tau \in R,\ q, q' \in Q.\ \tau \in T[q, q'] \ \Leftrightarrow\ (q, \mathsf{conf}(\tau), q') \in \delta^*$

By combining the previous results of this paper, we can now decide whether some configuration from the regular set of configurations $L(F)$ is reachable from the initial configuration $p_0\gamma_0$:

Theorem 10 (Main Result).

$$(\exists \bar{l} \in L^*,\ c' \in L(F).\ p_0\gamma_0 \xrightarrow{\bar{l}}{}^* c') \ \Leftrightarrow\ N[p_0, \gamma_0] \cap \bigcup\{T[q_0, q'] \mid q' \in Q_F\} \neq \emptyset$$

Note that both $N[p_0, \gamma_0]$ and $\bigcup\{T[q_0, q'] \mid q' \in Q_F\}$ are regular tree languages, and the rules of the corresponding tree automata can be constructed from the scheme given by the constraint systems for N, R and T. Hence, the right hand side of the equivalence can be decided using standard tree automata algorithms for intersection and emptiness check.

Proof. For the $\Rightarrow$-direction assume there is an sequential execution $\bar{l} \in L^*$ and a configuration $c' \in L(F)$ with $p_0\gamma_0 \xrightarrow{\bar{l}}{}^* c'$. With Theorem 4, we obtain an standard execution tree t with $p_0\gamma_0 \overset{t}{\Longrightarrow} c'$. From Assumption 2 and Lemma 3 we have $c' \in (P\Gamma^+)^+$, hence we can use Theorem 7a to obtain a regular execution tree $\tau \in N[p_0, \gamma_0]$ with $\alpha(\tau) = t$.

Moreover, with Lemma 6, we have $\mathsf{conf}(t) = c'$, and with Lemma 8 we get $\mathsf{conf}(\tau) = c'$. With $c' \in L(F)$ we obtain a state $q' \in Q_F$ with $(q_0, \mathsf{conf}(\tau), q') \in \delta^*$. With Theorem 9, we have $\tau \in T[q_0, q']$ and hence $\tau \in \bigcup\{T[q_0, q'] \mid q' \in Q_F\}$. Ultimately, we get $N[p_0, \gamma_0] \cap \bigcup\{T[q_0, q'] \mid q' \in Q_F\} \neq \emptyset$.

The proof of the $\Leftarrow$-direction is analog. Only in order to apply Theorem 4, we need the additional fact that every standard execution tree has at least one schedule (Lemma 5). $\square$

4 Conclusion

We presented a tree-based semantics for DPNs. The set of executions of the DPN are a regular set, and a tree automata for this set can efficiently be derived from the DPN rules. The semantics is justified w.r.t. the true-concurrency semantics presented in [7] and thus, indirectly, w.r.t the original interleaving semantics of DPNs [2]. By using standard tree automata techniques, we are able to decide tree regular properties of DPN executions. To show the usefulness of tree-regular properties, we showed that reaching a configuration from a given regular set is a tree-regular property.

All the results presented in this paper have been formalized in the interactive theorem prover Isabelle/HOL [9]. The proofs related to DPNs and standard execution trees have been published as a technical report [6]. The proofs related to regular execution trees are still unpublished.

Future Work Currently, we have only modeled one specific property – execution trees reaching a configuration from a regular set – as a tree-automaton and justified it by hand. However, there is strong indication that the function α, that maps from regular execution trees to standard execution trees, can be written as a macro tree transducer. Then, for every regular set $S \subseteq T$ of standard execution trees, the set $\alpha^{-1}(T) \subseteq R$ of corresponding regular execution trees is also regular, and an automaton for $\alpha^{-1}(T) \subseteq R$ can be constructed automatically. This would allow us to automatically transfer other useful properties of standard execution trees to our new analysis technique, e.g. the analysis of reachability w.r.t. nested locks presented in [7].

By iterating pre^*_M-computations, one can check for executions that reach a sequence of intermediate configurations from given regular sets. This can be used for, e.g., bounded model checking of DPNs with shared memory [1]. In order to achieve the same with our tree-automata based techniques, we have to specify tree automata for execution trees that reach a sequence of intermediate configurations, each from a given regular set. Ideally, we would develop a method to automatically derive these tree automata from regular properties that are separately specified for each execution between two intermediate configurations.

Another direction of future work is to extend the power of the analyzed model. While, in this paper, we only presented an analysis for DPNs, we are pretty sure that our results carry over to the strictly more powerful CDPNs [2], where rules may be constrained by properties of the state of the processes that have been spawned by the process that executes the rule. This allows, e.g., to model parallel procedure calls. Moreover, we could try to use other techniques, in particular Horn-Clause solvers and symbolic representations of the tree automata rules, to improve the performance of the analysis, in particular for large DPNs or properties where the automaton has many states but a possible short symbolic description.

Moreover, we have to compare our analysis techniques with the existing techniques for pre^*_M-computations. One aspect of this comparison is to compare the theoretical worst case complexities of the two methods. Another aspect would be an experimental evaluation of the runtimes for typical problems. We currently have prototype implementations for both techniques, but no experimental results yet, nor any collection of ,,typical problems".

Acknowledgment. We thank Helmut Seidl for helpful discussion about tree automata and Markus Müller-Olm and Alexander Wenner for inspiring discussion about DPNs. Actually, the idea of regular execution trees evolved during a meeting with Helmut, Markus and Alexander.

References

1. A. Bouajjani, J. Esparza, S. Schwoon, and J. Strejcek. Reachability analysis of multithreaded software with asynchronous communication. In *Proc. of FSTTCS'05*, volume 3821 of *LNCS*, pages 348–359. Springer, 2005.

2. A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *Proc. of CONCUR'05*, volume 3653 of *LNCS*. Springer, 2005.
3. V. Kahlon and A. Gupta. An automata-theoretic approach for model checking threads for LTL properties. In *Proc. of LICS 2006*, pages 101–110. IEEE Computer Society, 2006.
4. V. Kahlon and A. Gupta. On the analysis of interacting pushdown systems. In *POPL*, pages 303–314, 2007.
5. V. Kahlon, F. Ivancic, and A. Gupta. Reasoning about threads communicating via locks. In *Proc. of CAV 2005*, volume 3576 of *LNCS*. Springer, 2005.
6. P. Lammich. Isabelle formalization of hedge-constrained pre* and DPNs with locks. Available from `http://cs.uni-muenster.de/sev/publications/`. Technical Report.
7. P. Lammich, M. Müller-Olm, and A. Wenner. Predecessor sets of dynamic pushdown networks with tree-regular constraints. In *Computer Aided Verification*, volume 5643 of *LNCS*, 2009.
8. M. Müller-Olm. Precise interprocedural dependence analysis of parallel programs. *Theor. Comput. Sci.*, 311(1-3):325–388, 2004.
9. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
10. H. Seidl and B. Steffen. Constraint-based inter-procedural analysis of parallel programs. *Nordic Journal of Computing (NJC)*, 7(4):375–400, 2000.