Join-Lock-Sensitive Forward Reachability Analysis for Concurrent Programs with Dynamic Process Creation *

Thomas Martin Gawlitza[1], Peter Lammich[2], Markus Müller-Olm[2], Helmut Seidl[3], and Alexander Wenner[2]

[1] CNRS/VERIMAG, France, Thomas.Gawlitza@imag.fr
[2] Institut für Informatik, Westfälische Wilhelms-Universität Münster, Germany
{peter.lammich,markus.mueller-olm,alexander.wenner}@wwu.de
[3] Technische Universität München, Germany, seidl@in.tum.de

Abstract. Dynamic Pushdown Networks (DPNs) are a model for parallel programs with (recursive) procedures and dynamic process creation. *Constraints* on the sequences of spawned processes allow to extend the basic model with joining of created processes [2]. Orthogonally DPNs can be extended with nested locking [9]. Reachability of a regular set R of configurations in presence of stable constraints as well as reachability without constraints but with nested locking are based on computing the set of predecessors $\mathsf{pre}^*(R)$. In the present paper, we present a *forward*-propagating algorithm for deciding reachability for DPNs. We represent sets of executions by sets of *execution trees* and show that the set of all execution trees resulting in configurations from R which either allow a lock-sensitive execution or a join-sensitive execution, is *regular*. Here, we rely on basic results about *macro tree transducers*. As a second contribution, we show that reachability is decidable also for DPNs with both nested locking and joins.

1 Introduction

Bouajjani et al. [2] introduced (Constrained) Dynamic Pushdown Networks ((C)DPNs) for modeling parallel programs with (potentially recursive) procedures and dynamic process creation. They propose an algorithm inspired by the pre^*-algorithm for Pushdown Systems that, under the mild assumption that all constraints are *stable*, computes the set of all predecessors of a given regular set of configurations. Stable constraints are, for instance, capable of modeling *fork-join* parallelism as considered by Esparza and Podelski [4], Müller-Olm and Seidl [10], Seidl and Steffen [11].

Only weak forms of synchronization can be expressed in CDPNs. In order to deal with *nested* locking, Lammich et al. [9] introduced another mechanism for restricting executions. For unconstrained DPNs, they introduce *action trees*[4] to represent sets of related executions. Generalizing the acquisition history techniques of Kahlon et al. [6, 7], they construct a finite tree automaton to check whether an action tree represents a *lock-sensitive* execution. This allows them to perform a lock-sensitive backward reachability analysis for unconstrained DPNs with nested locking.

* This work was partially funded by the DFG project OpIAT (MU 1508/1-1 and SE 551/13-1) and the ANR project ASOPT.

[4] Action trees are called execution trees in Lammich et al. [9].

The techniques of Lammich et al. [9] cannot directly be adapted to perform a lock-sensitive *forward* reachability analysis, since the set of all possible executions cannot always be represented by a regular set of action trees. In this paper, we therefore propose an augmented version of action trees, which we call *execution trees* throughout the present paper. Our construction should be seen in analogy to contextfree grammars where the linear derivation process can be represented in treelike form by representing subderivations referring to distinct sections of sentential forms in distinct subtrees. We show that the set of executions can be represented by a *regular* set of execution trees. This enables us to use tree automata techniques for forward reachability analysis. Moreover, by applying known results for *macro tree transducers*, we show that every tree regular property of action trees can be translated into a tree regular property of execution trees. This allows us to re-use the results of Lammich et al. [9] to perform a *forward* reachability analysis for DPNs with joins as well as a lock-sensitive forward reachability analysis for DPNs with nested locking. The algorithms we obtain in this way are both natural and not less efficient than the corresponding algorithms for backward reachability analysis of Bouajjani et al. [2] and Lammich et al. [9], respectively. These algorithms for *forward* reachability are our first contribution.

In programming languages which support multithreaded programming such as JAVA and C with POSIX threads, joins and locks may occur simultaneously in programs. Therefore, as our second contribution, we provide an algorithm for performing a *forward* reachability analysis that is simultaneously join- and lock-sensitive. Following the approach of Lammich et al. [9], we construct a finite tree automaton which accepts all action trees for which a schedule exists that is simultaneously lock-sensitive and join-sensitive. This tree automaton can be used with both, the forward analysis techniques of this paper, and the backward analysis techniques of Lammich et al. [9]. Note that such an automaton cannot be constructed by simply intersecting a tree automaton for lock-sensitive action trees with a tree automaton for join-sensitive action trees, since action trees may represent multiple executions. Hence, an action tree may represent a lock-sensitive execution and a different join-sensitive execution, but no execution that is simultaneously join- and lock-sensitive.

Before we proceed with the technical constructions, we consider an introductory example written in a real-world programming language:

Example 1. We consider the following C program that uses POSIX threads:

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
void* f(void* x) {
  pthread_mutex_lock(&m); printf("A"); printf("B\n");
  pthread_mutex_unlock(&m); }
int main() { pthread_t t[10]; int i;
  for (i = 0; i < 10; i++) pthread_create(&t[i],NULL,f,NULL);
  for (i = 0; i < 10; i++) pthread_join(t[i],NULL);
  printf("End!\n"); return 0; }
```

Here, a main process creates 10 child-processes, waits until all these processes terminate, prints *End!*, and terminates. Each child-process prints *AB*. The lock m guarantees that the critical section of f is not used by two different processes at the same time. The main process does not need to acquire the lock m, since it only may access the

printer when no child-process is alive. A lock-sensitive reachability analysis which does not take joining into account, may flag a spurious warning that exclusive access to the printer might be violated. Likewise, a join-sensitive reachability analysis which ignores the locking cannot exclude potential violations. In order to exclude such false alarms, a reachability analysis is required which is simultaneously join- and lock-sensitive. □

2 Dynamic Pushdown Networks

Let us first introduce some notation: The Boolean lattice $\{\bot, \top\}$ is denoted by $\mathbb{B}$. The natural numbers are denoted by $\mathbb{N}$. We assume $0 \in \mathbb{N}$ and set $\mathbb{N}_+ := \mathbb{N} \setminus \{0\}$. The domain of a partial function $f : A \rightsquigarrow B$ is denoted by $\mathrm{dom}(f)$. $\oplus$ denotes the update operator for partial functions, i.e., for $f, g : A \rightsquigarrow B$ and $x \in A$, $(f \oplus g)(x)$ equals $g(x)$ if $x \in \mathrm{dom}(g)$ and $f(x)$ otherwise. We write $\{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$ for the partial function that maps x_i to y_i for all $i \in \{1, \ldots, n\}$ and is undefined otherwise.

Let Act be a non-empty set of *actions*. The specific action $\mathsf{sp} \notin \mathsf{Act}$ represents the spawning of a new process. We set $\overline{\mathsf{Act}} := \mathsf{Act} \cup \{\mathsf{sp}\}$. A *Dynamic Pushdown Network* (DPN) M is a tuple $(\mathsf{Act}, P, \Gamma, \Delta)$ where P is a finite set of *control states*, Γ is a finite set of *stack symbols*, and Δ is a finite set of *transition rules* of the following forms:

$$\text{(Base) } p\gamma \overset{a}{\hookrightarrow} p'\gamma' \qquad \text{(Push) } p\gamma \overset{a}{\hookrightarrow} p'\gamma_1\gamma_2$$
$$\text{(Pop) } p\gamma \overset{a}{\hookrightarrow} p' \qquad \text{(Spawn) } p\gamma \overset{\mathsf{sp}}{\hookrightarrow} p'\gamma' \rhd p_s\gamma_s$$

Here, $p, p', p_s \in P$, $a \in \mathsf{Act}$, and $\gamma, \gamma', \gamma_1, \gamma_2 \in \Gamma$. Intuitively, DPNs extend pushdown systems by (Spawn)-rules, that create a new process as a side effect.

A *configuration* of a DPN $M = (\mathsf{Act}, P, \Gamma, \Delta)$ is an *unranked* tree where each node in the tree is labeled with an element from $P\Gamma^*$. Intuitively, each node of the tree denotes a process and is labeled with the current configuration of the process. The children of a process are the processes that were spawned by the parent process. Formally, we represent nodes in trees by sequences of natural numbers: A set $T \subseteq \mathbb{N}_+^*$ is called a *tree domain* iff (1) T is finite, and (2) prefix-closed, i.e., $l \in T$ implies $l' \in T$ for all prefixes l' of l, and (3) if $l\,i \in T$ with $i > 1$ then also $l\,(i-1) \in T$. The set of all tree domains is denoted by $\mathcal{T}^{\mathbb{N}_+}$. Accordingly, a configuration c is a partial mapping $c : \mathbb{N}_+^* \rightsquigarrow P\Gamma^*$ where the domain $\mathrm{dom}(c)$ is a tree domain. Let $\mathcal{C}$ denote the set of all configurations of M. Within a configuration c, processes are organized hierarchically. Each element $l \in \mathrm{dom}(c)$ denotes a process. The set $\{li \in \mathrm{dom}(c) \mid i \in \mathbb{N}_+\}$ contains all child-processes of l within the configuration c. For $c(l) = pw$ with $p \in P, w \in \Gamma^*$, p is the control-state and w the stack of the process l within the configuration c. We call a configuration c_0 an *initial configuration* iff $c_0 = \{\epsilon \mapsto p\gamma\}$ for some $p \in P$ and some $\gamma \in \Gamma$, i.e. it contains a single process with a single symbol on its stack.

A single execution step of a DPN on a configuration applies a local rewrite step to the control-state and stack of one of the processes of the configuration. For all transition rules $r \in \Delta$ and all processes $l \in \mathbb{N}_+^*$, we define the partial function $\mapsto_r^l : \mathcal{C} \rightsquigarrow \mathcal{C}$ that applies r to the process l as follows:

$$\mapsto^l_{p\gamma \overset{a}{\hookrightarrow} p'w'}(c) := c \oplus \{l \mapsto p'w'w\} \qquad \text{if } c(l) = p\gamma w$$
$$\mapsto^l_{p\gamma \overset{\mathsf{sp}}{\hookrightarrow} p'\gamma' \rhd p_s\gamma_s}(c) := c \oplus \{l \mapsto p'\gamma' w, \mathsf{nc}(l, \mathrm{dom}(c)) \mapsto p_s\gamma_s\} \qquad \text{if } c(l) = p\gamma w$$

Here, $\mathsf{nc}(l, T) := l \cdot \min\{i \in \mathbb{N}_+ \mid l \cdot i \notin T\}$ is the sequence of natural numbers denoting the next child of process l which can be added to the tree domain T preserving the tree domain property.

Constraint Structures. In this paragraph we define *constraint structures* that allow us to restrict sequences of execution steps of DPNs. The idea is to disable undesired execution steps depending on a global state, that is computed in parallel to the actual configuration of the DPN. Constraint structures will later be used to define join-lock sensitive executions. Formally a constraint structure is a tuple $\mathbb{C} = (\mathbb{S}, \mathbb{T}, s_0)$ where $\mathbb{S}$ is a non-empty set of *global states*, $s_0 \in \mathbb{S}$ is an *initial state*, and $\mathbb{T} : \mathbb{N}_+^* \times \overline{\mathsf{Act}} \to \mathbb{S} \rightsquigarrow \mathbb{S}$ is a *global state transformer*. For $l \in \mathbb{N}_+^*$, $a \in \overline{\mathsf{Act}}$, and $s \in \mathbb{S}$, $\mathbb{T}(l, a)(s)$ denotes the global state resulting from applying action a to process l in global state s, or is undefined. If $\mathbb{T}(l, a)(s)$ is undefined, we consider the constraint to be violated. In this case, the corresponding action cannot be performed, i.e. the corresponding transition rule is not applicable. In order to capture this, we define the partial function ${}^{\mathbb{C}}\!\mapsto_r^l : \mathbb{S} \times \mathcal{C} \rightsquigarrow \mathbb{S} \times \mathcal{C}$ by

$${}^{\mathbb{C}}\!\mapsto_r^l((s, c)) := (\mathbb{T}(l, \overline{\mathsf{Act}}(r))(s), \mapsto_r^l(c)) \quad \text{if } \mathbb{T}(l, \overline{\mathsf{Act}}(r))(s) \text{ and } \mapsto_r^l(c) \text{ are defined}$$

Here, $\overline{\mathsf{Act}}(r)$ denotes the action of the transition rule r. The transition rule r is applied to the process l as before and the global state is transformed according to the transformer of the constraint structure. Thus, the function is only defined if the transformation of the global state is defined. Whenever the next global state is undefined, the constraint structure $\mathbb{C}$ prevents the application of r to l.

The set of configurations reachable from an initial configuration c_0 under a constraint structure $\mathbb{C}$ is defined as $\mathsf{post}^*_{\mathbb{C},M}(c_0) = \{c_0\} \cup \{c \mid \exists k \in \mathbb{N}_+, l_1, \ldots, l_k \in \mathbb{N}_+^*, r_1, \ldots, r_k \in \Delta, s \in \mathbb{S}.\ (s, c) = {}^{\mathbb{C}}\!\mapsto_{r_k}^{l_k}(\cdots {}^{\mathbb{C}}\!\mapsto_{r_1}^{l_1}(s_0, c_0)\cdots)\}$. One goal of this paper is to present an algorithm to check whether a set R of configurations is reachable from an initial configuration c_0, given one of the constraint structures $\mathbb{C} \in \{\mathbb{C}^{\mathrm{u}}, \mathbb{C}^{\mathrm{j}}, \mathbb{C}^{\mathrm{l}}, \mathbb{C}^{\mathrm{jl}}\}$ defined in the following. Formally, this is equivalent to checking whether

$$\mathsf{post}^*_{\mathbb{C},M}(c_0) \cap R \neq \emptyset.$$

Join/Lock-Constraints. In the present paper we consider joins together with nested locking. For that, let $L = \{1, \ldots, n\}$ be a fixed, finite set of locks. We assume a fixed set of actions $\mathsf{Act} = \{\mathsf{jo}, \$, \epsilon\} \cup \{\mathsf{acq}(i), \mathsf{rel}(i) \mid i \in L\}$. The action jo blocks the current process until all child-processes of the current process are stopped. The action \$ stops a process. The action ϵ represents actions unrelated to joining and locking. $\mathsf{acq}(i)$ acquires and $\mathsf{rel}(i)$ releases the lock i.

Unconstrained. For dealing with *unconstrained* DPNs, we define the constraint structure $\mathbb{C}^{\mathrm{u}} := (\{\top\}, \mathbb{T}^{\mathrm{u}}, \top)$ by $\mathbb{T}^{\mathrm{u}}(l, a)(\top) := \top$ for all $l \in \mathbb{N}_+^*, a \in \overline{\mathsf{Act}}$. This constraint structure never disables any execution steps.

Joins. For dealing with joins we define a constraint structure that keeps track of whether existing processes have stopped or not. Processes can only make steps if they have not

stopped and jo-steps can only be performed if all child processes have stopped. Hence we define $\mathbb{C}^{\mathsf{j}} := (\mathbb{S}^{\mathsf{j}}, \mathbb{T}^{\mathsf{j}}, s_0^{\mathsf{j}})$ by $\mathbb{S}^{\mathsf{j}} := \mathbb{N}_+^* \rightsquigarrow \mathbb{B}$, $s_0^{\mathsf{j}}(\epsilon) = \bot$, $s_0^{\mathsf{j}}(l)$ is undefined for all $l \in \mathbb{N}_+^* \setminus \{\epsilon\}$, and

$$\mathbb{T}^{\mathsf{j}}(l,a)(s) := \begin{cases} s & \text{if } s(l) = \bot, a = \mathsf{jo}, \\ & \quad \forall i \in \mathbb{N}_+.\ li \in dom(s) \Rightarrow s(li) = \top \\ s \oplus \{l \mapsto \top\} & \text{if } s(l) = \bot, a = \$ \\ s \oplus \{\mathsf{nc}(l, \mathrm{dom}(s)) \mapsto \bot\} & \text{if } s(l) = \bot, a = \mathsf{sp} \\ s & \text{if } s(l) = \bot, a \notin \{\mathsf{jo}, \$, \mathsf{sp}\} \end{cases}$$

for all $s \in \mathbb{S}^{\mathsf{j}}$, all $l \in \mathrm{dom}(s)$, and all $a \in \overline{\mathsf{Act}}$. For all $s \in \mathbb{S}^{\mathsf{j}}$ and all $l \in \mathbb{N}_+^*$, $s(l) = \top$ iff the process l exists and it is stopped.

Locks. For dealing with locks we define a constraint structure that keeps track of the set of locks held by each existing process. acq-steps can only be performed if the corresponding lock is held by no process and rel-steps require the corresponding lock to be held by the process. Formally, we define $\mathbb{C}^{\mathsf{l}} := (\mathbb{S}^{\mathsf{l}}, \mathbb{T}^{\mathsf{l}}, s_0^{\mathsf{l}})$ by $\mathbb{S}^{\mathsf{l}} := \mathbb{N}_+^* \rightsquigarrow 2^L$, $s_0^{\mathsf{l}}(\epsilon) = \emptyset$, $s_0^{\mathsf{l}}(l)$ is undefined for all $l \in \mathbb{N}_+^* \setminus \{\epsilon\}$, and

$$\mathbb{T}^{\mathsf{l}}(l,a)(s) := \begin{cases} s \oplus \{l \mapsto (s(l) \cup \{i\})\} & \text{if } a = \mathsf{acq}(i), i \notin \bigcup_{l' \in \mathrm{dom}(s)} s(l') \\ s \oplus \{l \mapsto (s(l) \setminus \{i\})\} & \text{if } a = \mathsf{rel}(i), i \in s(l) \\ s \oplus \{\mathsf{nc}(l, \mathrm{dom}(s)) \mapsto \emptyset\} & \text{if } a = \mathsf{sp} \\ s & \text{if } a \notin \{\mathsf{acq}(i), \mathsf{rel}(i), \mathsf{sp} \mid i \in L\} \end{cases}$$

for all $s \in \mathbb{S}^{\mathsf{l}}$, all $l \in \mathrm{dom}(s)$, and all $a \in \overline{\mathsf{Act}}$. For all $s \in \mathbb{S}^{\mathsf{l}}$ and all $l \in \mathbb{N}_+^*$, $s(l)$ is the set of locks that are acquired by the process l, provided that the process l exists. Here, we only consider non-reentrant locks, i.e., a process that already owns a lock i is not allowed to acquire i again. However, under certain conditions, DPNs with reentrant locks can be simulated by DPNs with non-reentrant locks at a cost that is exponential in the maximal nesting depth of locks (see Kidd et al. [8], Lammich et al. [9]).

Products of Constraint Structures. For all constraint structures $\mathbb{C}_1 = (\mathbb{S}_1, \mathbb{T}_1, s_0^{(1)})$ and $\mathbb{C}_2 = (\mathbb{S}_2, \mathbb{T}_2, s_0^{(2)})$, we define the *product* $\mathbb{C}_1 \times \mathbb{C}_2$ of $\mathbb{C}_1$ and $\mathbb{C}_2$ by

$$\mathbb{C}_1 \times \mathbb{C}_2 := (\mathbb{S}_1 \times \mathbb{S}_2, \mathbb{T}_1 \times \mathbb{T}_2, (s_0^{(1)}, s_0^{(2)})),$$

where $(\mathbb{T}_1 \times \mathbb{T}_2)(l,a)((s_1,s_2)) := (\mathbb{T}_1(l,a)(s_1), \mathbb{T}_2(l,a)(s_2))$ for all $l \in \mathbb{N}_+^*$, all $a \in \overline{\mathsf{Act}}$, and all $(s_1,s_2) \in \mathbb{S}_1 \times \mathbb{S}_2$ iff $\mathbb{T}_1(l,a)(s_1)$ and $\mathbb{T}_2(l,a)(s_2)$ are defined.

Joins and Locks. For dealing with joins and locks simultaneously, we define the constraint structure $\mathbb{C}^{\mathsf{jl}} := \mathbb{C}^{\mathsf{j}} \times \mathbb{C}^{\mathsf{l}}$. A DPN together with the constraint structure $\mathbb{C}^{\mathsf{jl}}$ is called a *join-lock-DPN* for short.

Example 2. The behavior of the C program of Example 1 can be safely over-approximated by the join-lock-DPN with the following transition rules:

$$p_m m \xhookrightarrow{\mathsf{sp}} p_m m \rhd p_f f \quad p_m m \xhookrightarrow{\mathsf{jo}} \$_m m \quad p_f f \xhookrightarrow{\mathsf{acq}(1)} q_f f \quad q_f f \xhookrightarrow{\mathsf{rel}(1)} r_f f \quad r_f f \xhookrightarrow{\$} \$_f f$$

The main process $p_m m$ may create arbitrary many child-processes $p_f f$ and may terminate (go into the state $\$_m$) as soon as all child-processes are stopped. Each child-process acquires the lock 1, releases the lock 1 and then stops. Because of the semantics of locks, two different child-processes may never be in state q_f simultaneously. Moreover, because of the semantics of joins, all child-processes must be in state $\$_f$, before the main-process can reach state $\$_m$. □

3 Forward Reachability

In this section we present our approach for checking forward reachability using only operations on regular sets of trees.

Configurations. In order to explicitly describe sets of configurations of a DPN $M = (\mathsf{Act}, P, \Gamma, \Delta)$ using regular tree languages, i.e. tree automata, we represent the unranked trees c from the set $\mathcal{C}$ by *ranked* trees. For that, we use *lists* constructed with cons and nil. As a shorthand we later write lists $\mathsf{cons}(a_1, \ldots, \mathsf{cons}(a_k, \mathsf{nil})\ldots)$ as $[a_1; \ldots; a_k]$. The set C of ranked trees which we will use is specified by the tree grammar:

$$LG ::= \mathsf{nil} \mid \mathsf{cons}(\Gamma, LG) \qquad LC ::= \mathsf{nil} \mid \mathsf{cons}(C, LC) \qquad C ::= P(LG, LC)$$

Hence a configuration is a ranked tree where the root node is labeled with the control state of the initial process. The root node has two branches. One is the list of stack symbols of the initial process. The other is the list of processes spawned by the initial process, where the last process spawned is the first in the list. For each spawned process the list contains a configuration with the process as initial process. The function $f : \mathcal{C} \to C$ captures the relation between the unranked and ranked model of configurations. It is defined by $f(c) := f(c, \epsilon)$ for all $c \in \mathcal{C}$, where, for all $c \in \mathcal{C}$ and all $l \in \mathrm{dom}(c)$, $f(c, l) := p([a_1; \ldots; a_m], [c'_k; \cdots; c'_1])$ if $c(l) = pa_1 \ldots a_m, p \in P, a_1, \ldots, a_m \in \Gamma$, $\{i \in \mathbb{N}_+ \mid l \cdot i \in \mathrm{dom}(c)\} = \{1, \ldots, k\}$, and $c'_1 = f(c, l\,1), \ldots, c'_k = f(c, l\,k)$. Recall that $l\,1$ is the first and $l\,k$ the last process spawned by process l within configuration c. Since f is a bijection, all functions and concepts defined in Section 2 can be transferred to the new representation and in the following we redefine the set $\mathcal{C}$ to be the set C of ranked trees. Hence a configuration $c \in \mathcal{C}$ is now a ranked tree and an initial configuration is represented by a term of the form $p([\gamma], \mathsf{nil})$ with $p \in P$ and $\gamma \in \Gamma$.

Known Results. For all pushdown systems M, $\mathsf{pre}^*(R)$ and $\mathsf{post}^*(R)$ are regular, whenever $R \subseteq \mathcal{C}$ is a regular set of configurations. Bouajjani et al. [1] provide a construction which, starting from a finite automaton for R, successively adds transitions until an automaton for $\mathsf{pre}^*(R)$ is obtained. Bouajjani et al. [2] applied a generalization of this construction to implement pre^*-operators for DPNs with *stable* constraints. The post^*-operator for DPNs, however, does *not* preserve regularity as shown in Bouajjani et al. [2] for a word model of configurations. This also holds for the tree model of this paper:

Example 3 (A Non-Regular post^*-Image). We consider the DPN $M = (\mathsf{Act}, P, \Gamma, \Delta)$ that is defined by $P = \{p, q, p', q', r, r', \$\}$, $\Gamma = \{a\}$, and

$$\Delta = \{pa \overset{\mathsf{sp}}{\hookrightarrow} qa \rhd ra,\ ra \overset{\$}{\hookrightarrow} \$,\ qa \overset{\epsilon}{\hookrightarrow} paa,\ pa \overset{\mathsf{jo}}{\hookrightarrow} p',\ p'a \overset{\mathsf{sp}}{\hookrightarrow} q'a \rhd r'a,\ q'a \overset{\epsilon}{\hookrightarrow} p'\}.$$

Here, $\mathsf{post}^*_{\mathbb{C}^{\mathrm{u}},M}(p([a],\mathsf{nil}))$ is not regular, since the process can spawn at most k processes of type $r'a$ after spawning k processes of type ra. □

Execution Trees. We avoid the problem of non-regularity by introducing *execution trees*. Execution trees can be regarded as configuration trees with a history of rule applications that lead from an initial configuration to the reached configuration. Inner nodes of execution trees are annotated with rules of the DPN and leaf nodes contain information about the current control state and, if existent, the current top-most stack symbol.

Starting from the root node, the leftmost branch of an execution tree contains the rules applied to the initial process in the order of application. Whenever a new process was spawned, the corresponding sp-rule has a separate branch containing the rules applied to the new process. Additionally, if a stack symbol is pushed that is later popped, the push-node has two branches: The left one describes the execution up to popping the stack symbol and the right one describes the rest of the execution.

To this end, we consider the set Δ of all transition rules of the DPN M as a ranked alphabet where *Push*-rules have rank 1 or 2, *Spawn*-rules have rank 2, *Base*-rules have rank 1, and *Pop*-rules have ranks 0. Additionally, for $p \in P$ and $\gamma \in \Gamma$, we introduce symbols $\langle p, \gamma\rangle$ of rank 0. The set X of execution trees is then described by the following regular tree grammar:

$$\begin{aligned}
\mathrm{XT} &::= \langle\mathsf{Base}\rangle(\mathrm{XT}) \mid \langle\mathsf{Push}\rangle(\mathrm{XT},\mathrm{XT}) \mid \langle\mathsf{Pop}\rangle \mid \langle\mathsf{Spawn}\rangle(\mathrm{XT},\mathrm{X})\\
\mathrm{XN} &::= \langle\mathsf{Base}\rangle(\mathrm{XN}) \mid \langle\mathsf{Push}\rangle(\mathrm{XN}) \mid \langle\mathsf{Push}\rangle(\mathrm{XT},\mathrm{XN}) \mid \langle\mathsf{Spawn}\rangle(\mathrm{XN},\mathrm{X}) \mid \langle P\times\Gamma\rangle\\
\mathrm{X} &::= \mathrm{XT} \mid \mathrm{XN}
\end{aligned}$$

Here, Base is the set of Base-rules, Push is the set of Push-rules, Pop is the set of Pop-rules, and Spawn is the set of Spawn-rules. The elements of the set X are the execution trees, where XT (*terminated execution trees*) contains the execution trees that end with popping the stack-symbol of the current stacklevel, and XN (*not yet terminated execution trees*) contains the execution trees that do not pop the stack-symbol of the current stacklevel.

An execution tree t reaches a configuration $\mathsf{c}(t)$. We can extract this configuration by $\mathsf{c}(t) := \mathsf{c}(t,\mathsf{nil},\mathsf{nil})$ with:

$$\begin{aligned}
\mathsf{c}(\langle p,\gamma\rangle, w, \overline{c}) &:= p(\mathsf{cons}(\gamma, w), \overline{c})\\
\mathsf{c}(\langle p\gamma \overset{a}{\hookrightarrow} p'\gamma'\rangle(t), w, \overline{c}) &:= \mathsf{c}(t, w, \overline{c})\\
\mathsf{c}(\langle p\gamma \overset{a}{\hookrightarrow} p'\gamma_1\gamma_2\rangle(t), w, \overline{c}) &:= \mathsf{c}(t, \mathsf{cons}(\gamma_2, w), \overline{c})\\
\mathsf{c}(\langle p\gamma \overset{a}{\hookrightarrow} p'\gamma_1\gamma_2\rangle(t_1,t_2), w, \overline{c}) &:= \mathsf{c}(t_2, w, \mathsf{ch}(t_1, \overline{c}))\\
\mathsf{c}(\langle p\gamma \overset{a}{\hookrightarrow} p'\rangle, w, \overline{c}) &:= p'(w, \overline{c})\\
\mathsf{c}(\langle p\gamma \overset{\mathsf{sp}}{\hookrightarrow} p\gamma \rhd p_s\gamma_s\rangle(t,t_s), w, \overline{c}) &:= \mathsf{c}(t, w, \mathsf{cons}(\mathsf{c}(t_s), \overline{c}))\\
\mathsf{ch}(\langle p,\gamma\rangle, \overline{c}) &:= \overline{c}\\
\mathsf{ch}(\langle p\gamma \overset{a}{\hookrightarrow} p'\gamma'\rangle(t), \overline{c}) &:= \mathsf{ch}(t, \overline{c})\\
\mathsf{ch}(\langle p\gamma \overset{a}{\hookrightarrow} p'\gamma_1\gamma_2\rangle(t), \overline{c}) &:= \mathsf{ch}(t, \overline{c})\\
\mathsf{ch}(\langle p\gamma \overset{a}{\hookrightarrow} p'\gamma_1\gamma_2\rangle(t_1,t_2), \overline{c}) &:= \mathsf{ch}(t_2, \mathsf{ch}(t_1, \overline{c}))\\
\mathsf{ch}(\langle p\gamma \overset{a}{\hookrightarrow} p'\rangle, \overline{c}) &:= \overline{c}
\end{aligned}$$

$$\mathsf{ch}(\langle p\gamma \overset{\mathsf{sp}}{\hookrightarrow} p\gamma \rhd p_s\gamma_s\rangle(t,t_s),\overline{c}) := \mathsf{ch}(t,\mathsf{cons}(\mathsf{c}(t_s),\overline{c}))$$

Here ch is an auxiliary function that extracts the list of configurations of all spawned processes in an execution tree. The function is used to collect the configurations of processes which were spawned in left branches of push-nodes. Note that initial configurations $p([\gamma],\mathsf{nil})$ are represented by execution trees consisting of a single node $\langle p,\gamma\rangle$.

The mappings c and ch for extracting the configuration can be interpreted as total deterministic *macro tree transducers* (see, e.g., Engelfriet and Vogler [3] for definitions and fundamental results), which use each of their input and output parameters exactly once. The pre-image of a regular set of trees w.r.t. the translation of any macro tree transducer is again regular [3]. By carefully inspecting the macro tree transducers realizing the mappings c and ch, we obtain:

Lemma 1. *For every regular set C of configurations, $\mathsf{c}^{-1}(C)$ is a regular set of execution trees. If C is given by a nondeterministic finite tree automaton with n states and m transitions, a finite tree automaton for $\mathsf{c}^{-1}(C)$ can be constructed in time $\mathcal{O}(m^2+|\Delta|\cdot n^4+|\Delta|\cdot m\cdot n^2+|P|\cdot|\Gamma|\cdot n)$.*

Due to lack of space, the proof of this and some other lemmas is deferred to the extended version of this paper [5].
We now define a semantics ${}^{\mathbb{C}}\!\Mapsto_r^l$ on execution trees that is compatible with the semantics ${}^{\mathbb{C}}\!\mapsto_r^l$ on configurations. For that, for all $r\in\Delta$, we define the partial function $\Mapsto_r : X \rightsquigarrow X$ that applies r to the root process of an execution tree as follows:

$$\begin{aligned}
\Mapsto_{p\gamma\overset{a}{\hookrightarrow}p'\gamma'}(\langle p,\gamma\rangle) &:= \langle p\gamma\overset{a}{\hookrightarrow}p'\gamma'\rangle(\langle p',\gamma'\rangle)\\
\Mapsto_{p\gamma\overset{a}{\hookrightarrow}p'\gamma_1\gamma_2}(\langle p,\gamma\rangle) &:= \langle p\gamma\overset{a}{\hookrightarrow}p'\gamma_1\gamma_2\rangle(\langle p',\gamma_1\rangle)\\
\Mapsto_{p\gamma\overset{a}{\hookrightarrow}p'}(\langle p,\gamma\rangle) &:= \langle p\gamma\overset{a}{\hookrightarrow}p'\rangle\\
\Mapsto_{p\gamma\overset{\mathsf{sp}}{\hookrightarrow}p'\gamma'\rhd p_s\gamma_s}(\langle p,\gamma\rangle) &:= \langle p\gamma\overset{\mathsf{sp}}{\hookrightarrow}p'\gamma'\rhd p_s\gamma_s\rangle(\langle p',\gamma'\rangle,\langle p_s,\gamma_s\rangle)\\
\Mapsto_r(\langle p\gamma\overset{a}{\hookrightarrow}p'\gamma'\rangle(t)) &:= \langle p\gamma\overset{a}{\hookrightarrow}p'\gamma'\rangle(\Mapsto_r(t))\\
\Mapsto_r(\langle p\gamma\overset{a}{\hookrightarrow}p'\gamma_1\gamma_2\rangle(t)) &:= \begin{cases}\langle p\gamma\overset{a}{\hookrightarrow}p'\gamma_1\gamma_2\rangle(t') & \text{if } t'\in \mathrm{XN}\\ \langle p\gamma\overset{a}{\hookrightarrow}p'\gamma_1\gamma_2\rangle(t',\langle s(\mathsf{c}(t')),\gamma_2\rangle) & \text{if } t'\in\mathrm{XT}\end{cases}\\
&\qquad \text{where } t'=\Mapsto_r(t)\\
\Mapsto_r(\langle p\gamma\overset{a}{\hookrightarrow}p'\gamma_1\gamma_2\rangle(t,t')) &:= \langle p\gamma\overset{a}{\hookrightarrow}p'\gamma_1\gamma_2\rangle(t,\Mapsto_r(t'))\\
\Mapsto_r(\langle p\gamma\overset{\mathsf{sp}}{\hookrightarrow}p'\gamma'\rhd p_s\gamma_s\rangle(t,t_s)) &:= \langle p\gamma\overset{\mathsf{sp}}{\hookrightarrow}p'\gamma'\rhd p_s\gamma_s\rangle(\Mapsto_r(t),t_s)
\end{aligned}$$

Here, $s(p(w,\overline{c})) := p$ for all $p(w,\overline{c})\in C$ extracts the control-state of the root process of an execution tree. It is used to extract the control state at the end of a left branch of a push-node. This control-state is the initial control-state for the continued execution on the right branch.

We define $\Mapsto_r^l(t)$ to be the application of $\Mapsto_r$ to the subtree of the execution tree which has the process identified by $l\in\mathbb{N}_+^*$ as root process. One can find the specific subtree by counting the number of sp-rules along the leftmost branch starting from the root node. If the correct sp-rule was found, we descend into the right branch

and continue. If we arrive at the end of a left branch of a push-node, we continue at the right branch of that node. Analogously to Section 2 we define ${}^{\mathbb{C}}\mapsto^{l}_{r}$ and the set $\mathsf{tpost}^{*}_{\mathbb{C},M}(\langle p,\gamma\rangle) = \{\langle p,\gamma\rangle\} \cup \{t \mid \exists k \in \mathbb{N}_{+}, l_1,\ldots,l_k \in \mathbb{N}^{*}_{+}, r_1,\ldots,r_k \in \Delta, s \in \mathbb{S}.\ (s,t) = {}^{\mathbb{C}}\mapsto^{l_k}_{r_k}(\cdots {}^{\mathbb{C}}\mapsto^{l_1}_{r_1}(s_0,\langle p,\gamma\rangle)\cdots)\}$ of successors of an execution tree representing an initial configuration for execution trees. An execution tree is called *consistent* iff there exists some $p \in P$ and some $\gamma \in \Gamma$ such that $t \in \mathsf{tpost}^{*}_{\mathbb{C}^{\mathrm{u}},M}(\langle p,\gamma\rangle)$, i.e. $\mathsf{tpost}^{*}_{\mathbb{C}^{\mathrm{u}},M}(\langle p,\gamma\rangle)$ is the set of consistent execution trees starting in $p\gamma$. We then have:

Lemma 2. $\mathsf{c}({}^{\mathbb{C}}\mapsto^{l}_{r}(t)) = {}^{\mathbb{C}}\mapsto^{l}_{r}(\mathsf{c}(t))$ *holds for all execution trees* $t \in \mathrm{X}$*, all transition rules* $r \in \Delta$*, all positions* $l \in \mathbb{N}^{*}_{+}$ *and constraint structures* $\mathbb{C}$*.*

Thus, to check reachability we can check $\mathsf{c}(\mathsf{tpost}^{*}_{\mathbb{C},M}(\langle p,\gamma\rangle)) \cap R \neq \emptyset$ or equivalently

$$\mathsf{tpost}^{*}_{\mathbb{C},M}(\langle p,\gamma\rangle) \cap \mathsf{c}^{-1}(R) \neq \emptyset.$$

We have already mentionend, that $\mathsf{c}^{-1}(R)$ is regular if R is regular. In order to obtain an algorithm which allows to check reachability solely based on operations on regular sets of trees, it remains to show that $\mathsf{tpost}^{*}_{\mathbb{C},M}(\langle p,\gamma\rangle)$ is regular as well. We first show this for unconstrained DPNs, i.e. for $\mathbb{C} = \mathbb{C}^{\mathrm{u}}$.

Regular characterization of $\mathsf{tpost}^{*}_{\mathbb{C},M}(\langle p,\gamma\rangle)$. Let $p \in P$ and $\gamma \in \Gamma$. Our goal is to compute the set $\mathsf{tpost}^{*}_{\mathbb{C}^{\mathrm{u}},M}(\langle p,\gamma\rangle)$ of all execution trees that can be reached starting from the execution tree $\langle p,\gamma\rangle$ without any restrictions due to the constraint structure. For that we construct a finite tree automaton which recognizes $\mathsf{tpost}^{*}_{\mathbb{C}^{\mathrm{u}},M}(\langle p,\gamma\rangle)$.

For all control states $p,p' \in P$ and all stack symbols $\gamma \in \Sigma$, we inductively define the set $\mathcal{T}^{p,\gamma,p'}$ (resp. $\mathcal{N}^{p,\gamma}$) of *terminated* (resp. *not yet terminated*) execution trees with initial control state p and top-most stack symbol γ and terminating with control state p' (resp. not terminating) as follows:

Open Leaf: For all $p \in P$, and all $\gamma \in \Gamma$: $\langle p,\gamma\rangle \in \mathcal{N}^{p,\gamma}$
Pop: For all $r = p\gamma \overset{a}{\hookrightarrow} p' \in \Delta$: $\langle r\rangle \in \mathcal{T}^{p,\gamma,p'}$
Base: For all $r = p\gamma \overset{a}{\hookrightarrow} p'\gamma' \in \Delta$, and all $p'' \in P$:
If $t \in \mathcal{T}^{p',\gamma',p''}$, then $\langle r\rangle(t) \in \mathcal{T}^{p,\gamma,p''}$.
If $t \in \mathcal{N}^{p',\gamma'}$, then $\langle r\rangle(t) \in \mathcal{N}^{p,\gamma}$.
Push: For all $r = p\gamma \overset{a}{\hookrightarrow} p'\gamma_1\gamma_2 \in \Delta$, and all $p'',p''' \in P$:
If $t_1 \in \mathcal{T}^{p',\gamma_1,p''}$ and $t_2 \in \mathcal{T}^{p'',\gamma_2,p'''}$, then $\langle r\rangle(t_1,t_2) \in \mathcal{T}^{p,\gamma,p'''}$.
If $t_1 \in \mathcal{T}^{p',\gamma_1,p''}$ and $t_2 \in \mathcal{N}^{p'',\gamma_2}$, then $\langle r\rangle(t_1,t_2) \in \mathcal{N}^{p,\gamma}$.
If $t_1 \in \mathcal{N}^{p',\gamma_1}$, then $\langle r\rangle(t_1) \in \mathcal{N}^{p,\gamma}$.
Spawn: For all $r = p\gamma \overset{\mathsf{sp}}{\hookrightarrow} p'\gamma' \rhd p_s\gamma_s \in \Delta$, and all $p'',p'_s \in P$:
If $t \in \mathcal{T}^{p',\gamma',p''}$ and $t_s \in \mathcal{T}^{p_s,\gamma_s,p'_s} \cup \mathcal{N}^{p_s,\gamma_s}$, then $\langle r\rangle(t,t_s) \in \mathcal{T}^{p,\gamma,p''}$.
If $t \in \mathcal{N}^{p',\gamma'}$ and $t_s \in \mathcal{T}^{p_s,\gamma_s,p'_s} \cup \mathcal{N}^{p_s,\gamma_s}$, then $\langle r\rangle(t,t_s) \in \mathcal{N}^{p,\gamma}$.

Thus, we get:

Theorem 1. *1.* $\mathsf{tpost}^{*}_{\mathbb{C}^{\mathrm{u}},M}(\langle p,\gamma\rangle) = \bigcup_{p' \in P} \mathcal{T}^{p,\gamma,p'} \cup \mathcal{N}^{p,\gamma}$.

2. The sets $\mathcal{T}^{p,\gamma,p'}$*,* $\mathcal{N}^{p,\gamma}$*, and finally* $\mathsf{tpost}^{*}_{\mathbb{C}^{\mathrm{u}},M}(\langle p,\gamma\rangle)$ *can be recognized by a finite tree automaton that can be constructed in time* $\mathcal{O}(|P|^2 \cdot (|\Gamma| + |\Delta|))$*.*

Proof (Sketch). Statement 1 can be shown by straight-forward induction. We focus on Statement 2: The states of the finite tree automaton correspond to the sets $\mathcal{T}^{p,\gamma,p'}$ and $\mathcal{N}^{p,\gamma}$. Hence, we have $|P|^2 \cdot |\Gamma|$ states. Moreover, there are $\mathcal{O}(|P| \cdot |\Gamma| + |P|^2 \cdot |\Delta|)$ transition rules. This gives us the desired complexity estimation. □

Hence, we can effectively check reachability for unconstrained DPNs using only operations on regular tree languages, i.e. finite tree automata. To close the gap to DPNs with a constraint structure, we observe, that the order in which rules are applied to processes running in parallel in an unconstrained DPN is irrelevant for the finally reached execution tree. However, this is not the case for DPNs with a constraint structure, since the order of rule applications determines how the global state of the constraint structure is transformed. Consequently, there may be orders which are not feasible, since a necessary transformation is undefined. To capture this, we define a *schedule* $(l_1, a_1), \ldots, (l_k, a_k)$ of an execution tree t as a sequence of process-labels and actions, such that there exists $p \in P$, $\gamma \in \Gamma$ and rules $r_1, \ldots, r_k \in \Delta$, with $\overline{\mathsf{Act}}(r_i) = a_i$, such that $t = \Mapsto_{r_k}^{l_k}(\cdots \Mapsto_{r_1}^{l_1}(\langle p, \gamma \rangle) \cdots)$. Each consistent execution tree has at least one schedule. A schedule is call *executable* under a constraint structure iff additionally $\mathbb{T}(l_k, a_k)(\cdots \mathbb{T}(l_1, a_1)(s_0) \cdots)$ is defined. Then, $\mathsf{tpost}^*_{\mathbb{C},M}(\langle p, \gamma \rangle)$ is exactly the set of consistent execution trees starting in $p\gamma$ that have an executable schedule under $\mathbb{C}$. As a consequence we can write $\mathsf{tpost}^*_{\mathbb{C},M}(\langle p, \gamma \rangle) = \mathsf{tpost}^*_{\mathbb{C}^{\mathsf{u}},M}(\langle p, \gamma \rangle) \cap H_{\mathbb{C}}$, where $H_{\mathbb{C}}$ is a set of execution trees that contains all consistent execution trees that have an executable schedule and no consistent execution tree that has no executable schedule. Then our reachability problem boils down to checking

$$\mathsf{tpost}^*_{\mathbb{C}^{\mathsf{u}},M}(\langle p, \gamma \rangle) \cap H_{\mathbb{C}} \cap \mathsf{c}^{-1}(R) \neq \emptyset.$$

In the remainder of this paper, we construct regular sets $H_{\mathbb{C}}$ for all constraint structures $\mathbb{C} \in \{\mathbb{C}^{\mathsf{j}}, \mathbb{C}^{\mathsf{l}}, \mathbb{C}^{\mathsf{jl}}\}$. Thus, checking reachability reduces to checking emptiness of intersections of regular sets of trees.

4 Action Trees

At the end of the last section we have observed that, in order to check reachability under a constraint structure, we need to specify a regular set of execution trees that contains all consistent execution trees with an executable schedule and no consistent execution tree without an executable schedule. We have also observed that executability of a schedule depends only on the actions of the rules and identities of the processes taking part in the execution. Consequently, we do not need the full power of the execution tree model to capture executability. Indeed, it is simpler to characterize executability if the left branches of push-nodes are inlined as in the trees used by Lammich et al. [9]. Therefore, in this section, we introduce *action trees* as an abstraction of execution trees that does this inlining and omits information about control states and stack contents.

Formally, the set of action trees $\mathcal{T}^{\mathsf{Act}}$ is defined by the tree grammar $\mathcal{T}^{\mathsf{Act}} ::= \epsilon \mid \langle \mathsf{Act} \rangle(\mathcal{T}^{\mathsf{Act}}) \mid \langle \mathsf{sp} \rangle(\mathcal{T}^{\mathsf{Act}}, \mathcal{T}^{\mathsf{Act}})$. The root node and each right child of an sp-node within an action tree represents a process that performs the actions of the left-most path starting from this node and ending in a leaf.

Schedules. For all $l \in \mathbb{N}_+^*$, $a \in \overline{\mathsf{Act}}$, we define the partial function $\rightarrowtail_a^l : \mathcal{T}^{\mathsf{Act}} \rightsquigarrow \mathcal{T}^{\mathsf{Act}}$, that constructs action trees by

$$\begin{aligned}
\rightarrowtail_a^l(\epsilon) &:= \langle a \rangle(\epsilon) && \text{if } l = \epsilon, a \in \mathsf{Act} \\
\rightarrowtail_{\mathsf{sp}}^l(\epsilon) &:= \langle \mathsf{sp} \rangle(\epsilon, \epsilon) && \text{if } l = \epsilon \\
\rightarrowtail_a^l(\langle a' \rangle(t)) &:= \langle a' \rangle(\mapsto_a^l(t)) && \text{if } a' \in \mathsf{Act} \\
\rightarrowtail_a^l(\langle \mathsf{sp} \rangle(t, t_s)) &:= \langle \mathsf{sp} \rangle(\mapsto_a^l(t), t_s) && \text{if } l = \epsilon \\
\rightarrowtail_a^l(\langle \mathsf{sp} \rangle(t, t_s)) &:= \langle \mathsf{sp} \rangle(t, \mapsto_a^{l'}(t_s)) && \text{if } l = 1 \cdot l' \\
\rightarrowtail_a^l(\langle \mathsf{sp} \rangle(t, t_s)) &:= \langle \mathsf{sp} \rangle(\mapsto_a^{(i-1)\cdot l'}(t), t_s) && \text{if } l = i \cdot l', i > 1
\end{aligned}$$

$\rightarrowtail_a^l(t)$ appends the action a to the process l of the action tree. Identification of processes is similar as in the case of execution trees.

A schedule of an action tree t is a finite sequence $(l_1, a_1), \ldots, (l_k, a_k)$, where $l_i \in \mathbb{N}_+^*$ and $a_i \in \overline{\mathsf{Act}}$ for all $i \in \{1, \ldots, k\}$, with $t = \rightarrowtail_{a_k}^{l_k}(\cdots \rightarrowtail_{a_1}^{l_1}(\epsilon) \cdots)$. Action trees always have at least one schedule in contrast to execution trees, where only consistent execution trees have a schedule. Recall that a schedule is called executable under a constraint structure $\mathbb{C}$ iff $\mathbb{T}(l_k, a_k)(\cdots \mathbb{T}(l_1, a_1)(s_0) \cdots)$ is defined.

Execution Trees and Action Trees. Each execution tree t can be abstracted to an action tree $\mathsf{a}(t)$ by cutting away unnecessary information. We define $\mathsf{a}(t) := \mathsf{a}(t, \epsilon)$, where:

$$\begin{aligned}
\mathsf{a}(\langle p, \gamma \rangle, x) &:= x \\
\mathsf{a}(\langle p\gamma \overset{a}{\hookrightarrow} p'\gamma' \rangle(t), x) &:= \langle a \rangle(\mathsf{a}(t, x)) \\
\mathsf{a}(\langle p\gamma \overset{a}{\hookrightarrow} p'\gamma_1\gamma_2 \rangle(t), x) &:= \langle a \rangle(\mathsf{a}(t, x)) \\
\mathsf{a}(\langle p\gamma \overset{a}{\hookrightarrow} p'\gamma_1\gamma_2 \rangle(t_1, t_2), x) &:= \langle a \rangle(\mathsf{a}(t_1, \mathsf{a}(t_2, x))) \\
\mathsf{a}(\langle p\gamma \overset{a}{\hookrightarrow} p' \rangle, x) &:= \langle a \rangle(x) \\
\mathsf{a}(\langle p\gamma \overset{\mathsf{sp}}{\hookrightarrow} p\gamma \rhd p_s\gamma_s \rangle(t, t_s), x) &:= \langle \mathsf{sp} \rangle(\mathsf{a}(t, x), \mathsf{a}(t_s))
\end{aligned}$$

The mapping a for extracting action trees can again be interpreted as a total deterministic macro tree transducer. We get:

Lemma 3. *For every regular set T of action trees, $\mathsf{a}^{-1}(T)$ is a regular set of execution trees. If T is given by a nondeterministic finite tree automaton with n states and m transitions, a finite tree automaton for $\mathsf{a}^{-1}(T)$ can be constructed in time $\mathcal{O}(|P| \cdot |\Gamma| \cdot n + |\Delta| \cdot m \cdot n^2)$.*

Moreover, scheduling of action trees is compatible with that of execution trees:

Lemma 4. *A consistent execution tree t has an executable schedule under a constraint structure $\mathbb{C}$ iff $a(t)$ has an executable schedule.*

Consequently, for the set $T_{\mathbb{C}}$ of all action trees that have an executable schedule under the constraint structure $\mathbb{C}$, each consistent execution tree $t \in a^{-1}(T_{\mathbb{C}})$ has an executable schedule. Vice versa, each consistent execution tree $t \notin a^{-1}(T_{\mathbb{C}})$ has no executable schedule. Thus, we define $H_{\mathbb{C}} := a^{-1}(T_{\mathbb{C}})$. Since a^{-1} preservers regularity, $H_{\mathbb{C}}$ is regular if $T_{\mathbb{C}}$ is regular.

Lammich et al. [9] showed how to construct a tree automaton $\mathsf{check}^{\mathsf{l}}$ that accepts the set $T_{\mathbb{C}^{\mathsf{l}}}$ of all action trees that can be scheduled lock-sensitively, i.e. have an executable

schedule under $\mathbb{C}^{\mathsf{l}}$. A tree automaton $\mathsf{check}^{\mathsf{j}}$ that accepts the set $T_{\mathbb{C}^{\mathsf{j}}}$ of all action trees that can be scheduled join-sensitively, i.e. have an executable schedule under $\mathbb{C}^{\mathsf{j}}$, can be – rather straightforwardly – constructed in polynomial time as well. The tree automaton $\mathsf{check}^{\mathsf{j}}$ checks that all child-processes that are spawned before a jo-action are stopped using the action \$ and the action \$ is only performed at the end of branches.

The language $T_{\mathbb{C}^{\mathsf{jl}}}$, however, need not to be the intersection of the languages $T_{\mathbb{C}^{\mathsf{j}}}$ and $T_{\mathbb{C}^{\mathsf{l}}}$.The reason is that the set of join-sensitive schedules and the set of lock-sensitive schedules of t might be disjoint.

5 Join-Lock-Sensitive Schedules of Action Trees

In this section we construct a deterministic bottom-up tree automaton $\mathsf{check}^{\mathsf{jl}}$ which recognizes the set of all action trees that can be scheduled join-lock-sensitively. We restrict ourselves to action trees where each single process in isolation uses locks in a well-nested fashion, releases all locks before stopping, and does not try to perform actions after the stop action. We call such action trees join-lock-well-formed, a notion defined more formally in the next paragraph.

Well-Formedness. A sequence of actions $a_1 \cdots a_k$ is called *well-nested* iff its lock operations are a prefix of a sequence from the context-free language $U ::= \epsilon \mid U \cdot U \mid \mathsf{acq}(1) \cdot U \cdot \mathsf{rel}(1) \mid \cdots \mid \mathsf{acq}(n) \cdot U \cdot \mathsf{rel}(n)$. It is called *non-reentrant* iff each acquired lock is released before it is acquired again, i.e., for each $a_i = \mathsf{acq}(j),\ a_{i'} = \mathsf{acq}(j),\ i < i'$, there is an i'' with $i < i'' < i'$ and $a_{i''} = \mathsf{rel}(j)$, and, symmetrically, each released lock is re-acquired before it is released again, i.e., for each $a_i = \mathsf{rel}(j),\ a_{i'} = \mathsf{rel}(j),\ i < i'$ there is an i'' with $i < i'' < i'$ and $a_{i''} = \mathsf{acq}(j)$. A sequence of actions is called *well-formed* iff it is both well-nested and non-reentrant. It is called *join-lock-well-formed* iff (1) it is well-formed, (2) there is no action after the action \$, and (3) if the sequence ends with the action \$, then its sequence of lock operations is from U, i.e., all acquired locks are released before terminating with the action \$.

An action tree is called *join-lock-well-formed* iff the actions of each process form a join-lock-well-formed sequence. A join-lock-DPN M is called *join-lock-well-formed* iff all action trees that correspond to consistent execution trees of M are join-lock-well-formed. Note that the set of join-lock-well-formed action trees is regular. Thus, we can use the techniques presented above to check whether a DPN is join-lock-well-formed.

Acquisition-Structures. For defining the tree automaton $\mathsf{check}^{\mathsf{jl}}$, we use a modified version of *acquisition structures* as introduced by Lammich et al. [9].

An acquisition of a lock i without a matching release is called a *final acquisition* of i. A matching release means, that the same process releases the lock, i.e., that there is a release of i on the leftmost path starting at the acquisition node. Symmetrically, a release of i without a matching acquisition is called an *initial release*. Acquisitions and releases that are not final acquisitions or initial releases are called *usages*.

Let t be a join-lock-well-formed action tree. We first check the following properties, that are obviously necessary for the existence of a join-lock-sensitive schedule: (1) Every lock is finally acquired at most once. (2) For every process, the last action of all

child-processes that are spawned before some join is \$. We write $\mathsf{ok}(t)$ iff t fulfills the requirements (1) and (2). The evaluation of the predicate ok can easily be implemented by a deterministic bottom-up tree automaton where the complexity of the construction is exponential in the number of locks.

To each subtree of a join-lock-well-formed action tree, we assign an acquisition structure that can be used for constructing a join-lock-sensitive schedule. In this paper we define an acquisition structure either to be $\bot$ or a tuple $(A, R, R_{\mathsf{jo}}, U, \rightarrow_A, J)$ where:

- A is the set of acquired locks (final acquisitions and usages).
- R is the set of initially released locks. Note that, due to well-nestedness, only the root process of the subtree may contain initial releases.
- R_{jo} is the set of locks that are initially released after the next join. If the root process contains no join operation, we have $R_{\mathsf{jo}} = \emptyset$. Moreover, we have $R_{\mathsf{jo}} \subseteq R$.
- U is the set of locks whose usages are required for termination of the root process, including locks whose usages are required for termination of spawned processes the root process joins with.
- $\rightarrow_A \subseteq L^2$ is the *acquisition graph*. We have $i \rightarrow_A j$ iff there is an acquisition of j after a final acquisition of i.
- $J \in \mathbb{B}$ is $\top$ iff the root process performs at least one jo.

The acquisition structure $\mathsf{as}(t)$ of an action tree t is defined by the following rules, if the subtrees of t have non-bottom acquisition structures and the side conditions of the rules are fulfilled. In any other case, we set $\mathsf{as}(t) := \bot$.

$$
\begin{aligned}
\mathsf{as}(\epsilon) := \mathsf{as}(\langle \$ \rangle(\epsilon)) &:= (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \bot) \\
\mathsf{as}(\langle \epsilon \rangle(t)) &:= \mathsf{as}(t) \\
\mathsf{as}(\langle \mathsf{acq}(i) \rangle(t)) &:= (A \cup \{i\}, R \setminus \{i\}, R_{\mathsf{jo}} \setminus \{i\}, U \cup \{i\}, \rightarrow_A, J), && \text{if } i \in R \\
\mathsf{as}(\langle \mathsf{acq}(i) \rangle(t)) &:= (A \cup \{i\}, R, R_{\mathsf{jo}}, U, \rightarrow_A \cup \{i\} \times A, J), && \text{if } i \notin R \\
\mathsf{as}(\langle \mathsf{rel}(i) \rangle(t)) &:= (A, R \cup \{i\}, R_{\mathsf{jo}}, U, \rightarrow_A, J) \\
\mathsf{as}(\langle \mathsf{jo} \rangle(t)) &:= (A, R, R, U, \rightarrow_A, \top) \\
\mathsf{as}(\langle \mathsf{sp} \rangle(t, t')) &:= (A \cup A', R, R_{\mathsf{jo}}, U, \rightarrow_A \cup \rightarrow_{A'}, \bot), && \text{if } J = \bot \\
\mathsf{as}(\langle \mathsf{sp} \rangle(t, t')) &:= (A \cup A', R, R_{\mathsf{jo}}, U \cup U', \rightarrow_A \cup \rightarrow_{A'}, \top), \\
& && \text{if } J = \top \text{ and } R_{\mathsf{jo}} \cap U' = \emptyset
\end{aligned}
$$

where $\mathsf{as}(t) = (A, R, R_{\mathsf{jo}}, U, \rightarrow_A, J)$ and $\mathsf{as}(t') = (A', R', R'_{\mathsf{jo}}, U', \rightarrow_{A'}, J')$. We prove that a join-lock-well-formed action tree t can be scheduled join-lock-sensitively iff $\mathsf{ok}(t)$ and $\mathsf{as}(t) = (A, R, R_{\mathsf{jo}}, U, \rightarrow_A, J)$ with acyclic $\rightarrow_A$. Since the number of acquisition structures is finite, a deterministic bottom-up tree automaton $\mathsf{check}^{\mathsf{jl}}$ can be constructed that accepts the action trees that satisfy the above property:

Theorem 2. *There is a deterministic bottom-up tree automaton* $\mathsf{check}^{\mathsf{jl}}$ *which accepts a join-lock-well-formed action tree* t *iff* t *can be scheduled join-lock-sensitively;* $\mathsf{check}^{\mathsf{jl}}$ *can be constructed in time at most exponential in the number of locks.*

Proof (Sketch). Due to lack of space only a sketch is presented here. The complete proof can be found in the extended version of this paper [5].

The complexity estimate for $\mathsf{check}^{\mathsf{jl}}$ follows from the fact that the number of acquisition structures is exponentially bounded in the number of locks.

Assume that $\mathsf{check}^{\mathsf{jl}}$ does not accept t. Hence, not $\mathsf{ok}(t)$ or $\mathsf{as}(t) = \bot$ or $\mathsf{as}(t) = (A, R, R_{\mathsf{jo}}, U, \rightarrow_A, J)$ with cyclic $\rightarrow_A$. First of all, it is easy to see that there is no join-lock sensitive schedule if $\mathsf{ok}(t)$ does not hold. If $\mathsf{as}(t) = \bot$, this results from a failed $R_{\mathsf{jo}} \cap U' = \emptyset$ check in the second sp-rule of m. This means that there is a process that spawns another process and then joins, while it holds a lock that is required by the spawned process for termination. Hence, no lock-sensitive schedule is possible. The edges in $\rightarrow_A$ capture ordering constraints on final acquisitions of locks: $i \rightarrow_A j$ means, that there is a final acquisition of i that must be scheduled before the final acquisition of j, if any. If $\rightarrow_A$ is cyclic, these constraints obviously cannot be fulfilled.

If $\mathsf{check}^{\mathsf{jl}}$ accepts t, then $\mathsf{ok}(t)$ holds and $\mathsf{as}(t) = (A, R, R_{\mathsf{jo}}, U, \rightarrow_A, J)$ with an acyclic $\rightarrow_A$. We first choose a total ordering $<$ on the locks with $\rightarrow_A \subseteq <$. Then, we construct inductively a schedule $s(t')$ for each subtree t' of t with the following properties: (1) The actions in $s(t')$ form a suffix of a join-lock-well-formed action sequence. (2) $s(t')$ is join-sensitive, i.e. steps of spawned processes are scheduled before the respective join operations. (3) Acquisitions in $s(t')$ respect the ordering $<$ in the sense that only locks $i > j$ are acquired after an unmatched acquisition of j. (4) Only required locks are used before termination, i.e. before termination of the root process, all acquired locks are released and are from U' (where $\mathsf{as}(t') = (A', R', R'_{\mathsf{jo}}, U', \rightarrow_{A'}, J')$).

The construction of this schedule is straightforward except for subtrees of the form $t' = \langle \mathsf{sp} \rangle(t_1, t_2)$. In this case, $s(t')$ is constructed as a specific shuffling of $s(t_1)$ and $s(t_2)$. Then, by (1) and (2), the schedule $s(t)$ constructed for the entire tree t is join-lock sensitive. □

We are now prepared to put all parts together:

Theorem 3. *Let $M = (\mathsf{Act}, P, \Gamma, \Delta)$ be a join-lock-well-formed join-lock-DPN, $p \in P$, $\gamma \in \Gamma$, and R a regular set of configurations. It can be decided whether a configuration from R can be reached by M, starting from the configuration $p([\gamma], \mathsf{nil})$ through an execution that respects joins and locks simultaneously, i.e., whether or not*

$$\mathsf{post}^*_{\mathbb{C}^{\mathsf{jl}}, M}(p([\gamma], \mathsf{nil})) \cap R \neq \emptyset$$

holds, by checking the following intersection of regular sets:

$$\mathsf{tpost}^*_{\mathbb{C}^{\mathsf{u}}, M}(\langle p, \gamma \rangle) \cap a^{-1}(T) \cap \mathsf{c}^{-1}(R) \neq \emptyset$$

where T is the language of $\mathsf{check}^{\mathsf{jl}}$. The complexity is polynomial in the size of M and in that of a tree automaton recognizing R and exponential only in the number of locks.

6 Conclusion

We have developed a *forward* propagating algorithm for analyzing reachability in Dynamic Pushdown Networks. The key idea was to represent all possible executions by means of a regular set of execution trees. We then showed how to restrict the set of all possible executions to executions that respect joins and locks simultaneously. By that, we obtained an algorithm for analyzing reachability w.r.t. join-lock-sensitive executions. Our algorithms are polynomial in the size of the DPN as well as in the size of the

finite tree automaton describing the set of configurations to be reached and exponential only in the number of locks used by the DPN.

It would be interesting to explore to what extent nested locking can be combined with concepts that are more general than joins, e.g. *stable* constraints as considered by Bouajjani et al. [2]. It might also be challenging to see which other properties can be decided for the different classes of DPNs beyond reachability. Finally, it is left to future work to practically evaluate the provided techniques on real world examples.

Bibliography

[1] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In A. W. Mazurkiewicz and J. Winkowski, editors, *CONCUR*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 1997. ISBN 3-540-63141-0.

[2] A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *Concurrency Theory. 16th Int. Conf. (CONCUR)*, pages 473–487. LNCS 3653, Springer, 2005.

[3] J. Engelfriet and H. Vogler. Macro tree transducers. *J. Comput. Syst. Sci.*, 31(1): 71–146, 1985.

[4] J. Esparza and A. Podelski. Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In *POPL*, pages 1–11, 2000.

[5] T. M. Gawlitza, P. Lammich, M. Müller-Olm, H. Seidl, and A. Wenner. Join-lock-sensitive forward reachability analysis for concurrent programs with dynamic process creation. Available from `http://cs.uni-muenster.de/sev/publications/`. Extended Version.

[6] 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.

[7] V. Kahlon, F. Ivancic, and A. Gupta. Reasoning about threads communicating via locks. In *Proc. of CAV 2005*, volume 3576 of *LNCS*. Springer, 2005.

[8] N. Kidd, A. Lal, and T. W. Reps. Language strength reduction. In M. Alpuente and G. Vidal, editors, *SAS*, volume 5079 of *Lecture Notes in Computer Science*, pages 283–298. Springer, 2008. ISBN 978-3-540-69163-1.

[9] P. Lammich, M. Müller-Olm, and A. Wenner. Predecessor sets of dynamic pushdown networks with tree-regular constraints. In A. Bouajjani and O. Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 525–539. Springer, 2009. ISBN 978-3-642-02657-7.

[10] M. Müller-Olm and H. Seidl. On optimal slicing of parallel programs. In *STOC*, pages 647–656, 2001.

[11] H. Seidl and B. Steffen. Constraint-based inter-procedural analysis of parallel programs. *Nord. J. Comput.*, 7(4):375–, 2000.