Predecessor Sets of Dynamic Pushdown Networks with Tree-Regular Constraints

Peter Lammich, Markus Müller-Olm, and Alexander Wenner

Institut für Informatik, Fachbereich Mathematik und Informatik
Westfälische Wilhelms-Universität Münster
{peter.lammich,markus.mueller-olm,alexander.wenner}@uni-muenster.de

Abstract. Dynamic Pushdown Networks (DPNs) are a model for parallel programs with (recursive) procedures and process creation. The goal of this paper is to develop generic techniques for more expressive reachability analysis of DPNs.
In the first part of the paper we introduce a new tree-based view on executions. Traditional interleaving semantics model executions by totally ordered sequences. Instead, we model an execution by a partially ordered set of rule applications, that only specifies the per-process ordering and the causality due to process creation, but no ordering between rule applications on processes that run in parallel. Tree-based executions allow us to compute predecessor sets of regular sets of DPN configurations relative to (tree-) regular constraints on executions. The corresponding problem for interleaved executions is not effective.
In the second part of the paper, we extend DPNs with (well-nested) locks. We generalize Kahlon and Gupta's technique of acquisition histories to DPNs, and apply the results of the first part of this paper to compute lock-sensitive predecessor sets.

1 Introduction

Writing parallel programs is notoriously difficult, as concurrency-related bugs are hard to find and hard to reproduce due to the nondeterministic behavior of the scheduler. Hence there is a real need for automated methods for verifying parallel programs. The goal of this paper is to develop stronger techniques for reachability analysis of *Dynamic Pushdown Networks (DPNs)* [2], a formal model of parallel programs with (recursive) procedures and process creation.

DPNs generalize pushdown systems by rules that have the additional side effect of creating a new process that is then executed in parallel. The key concept for analyzing DPNs is computation of predecessor sets. Configurations of a DPN are represented as words over control and stack symbols, and for a regular set of configurations, the set of predecessor configurations is regular as well and can be computed efficiently [2]. Predecessor computations can be used for various interesting analyses, like kill/gen analysis on bitvectors, context-bounded model checking [1], and data race detection.

Usually, DPNs are analyzed w.r.t. an interleaving semantics, where an execution is a sequence of rule applications. Interleaving semantics models the execution on a single processor, that performs one step at a time and may switch the currently executing process after every step. However, the set of interleaved executions does not have nice properties, which makes them difficult to reason about. For example, it is undecidable whether there exists an execution with a given regular property. Moreover, a step of the interleaving semantics does not contain the information which process executed the step, making interleaving semantics inappropriate to track properties of specific processes, e.g. acquired locks.

In the first part of this paper, we introduce an alternative view on executions of DPNs. An execution is modeled as a partially ordered set of steps, rather than a (totally ordered) sequence. The new semantics only reflects the ordering between steps of the same process and the causality due to process creation, i.e. that steps of a created process must be executed after the step that created the process. However, it does not enforce any ordering between steps of processes running in parallel. The new semantics does not lead to any loss of information as the interleaved executions can be recovered as topological sorts of the partial ordering. The partial ordering of an execution has a tree shape, where thread creation steps have at most two successors and all other steps have at most one successor. Hence, we model an execution as a list of trees (called *execution hedge*), that contains one tree for each process in the start configuration of the execution.

Taking advantage of our new, tree-based view on executions, we increase the expressivity of predecessor computations. Specifically, we show that for a regular set of configurations C and a (tree-) regular set of execution hedges H, the set of configurations from which a configuration in C is reachable via an execution in H is, again, regular and can be computed efficiently. We call this set the H-constrained predecessor set of C. Note that the corresponding problem for the interleaving semantics, i.e. predecessor computation with a (word-)regular constraint on the interleaved execution, is not effective[1].

In the second part of this paper, we extend DPNs by adding mutual exclusion via well-nested locks. Locks are a commonly used synchronization primitive to manage shared resources between processes. A process may acquire and release a lock, and a lock may be owned by at most one process at the same time. If a process wants to acquire a lock that is already owned by another process, it has to wait until the lock is released. We assume that locks are used in a well-nested fashion, i.e. a process has to release locks in the opposite order of acquisition, an assumption that is often satisfied in practice. For instance, the

[1] We can use the regular constraint $(a\bar{a} + b\bar{b})^*$ to synchronize two pushdown systems, thus reducing the emptiness check of the intersection of two context free languages, which is well-known to be undecidable, to this problem. We should mention that, nevertheless, predecessor sets w.r.t. a special class of regular constraints, called *alphabetic path constraints*, can be computed. Alphabetic path constraints have the form $S_1^{m_1} \dots S_n^{m_n}$ with $S_1, \dots, S_n \subseteq L$ and $m_1, \dots, m_i \in \{1, *\}$.

synchronized-blocks of Java guarantee well-nested lock usage syntactically. Note that for non-well-nested locks even simple reachability problems are undecidable [5]. We can describe lock-sensitive executions by a tree-regular constraint, thus obtaining an algorithm for precise computation of lock-sensitive predecessor sets of regular sets of computations (lock-sensitive pre^*). For this purpose, we generalize acquisition histories [5, 4] to DPNs.

Summarizing, the contributions of this paper are:

- We present a tree-based view on DPN executions, and an efficient predecessor computation procedure that takes tree-regular constraints on the executions into account.
- We generalize the concept of acquisition histories to programs with potentially recursive procedures and process creation.
- We characterize lock-sensitive executions by a tree-regular constraint. Applying the predecessor computation procedure, this leads to an algorithm for computing lock-sensitive pre^*.

Related Work. The results presented in this paper have been formalized and verified with the interactive theorem prover Isabelle/HOL [11]. The proof document is available as a technical report [9]. The theorems in this paper are annotated with the section and name of the corresponding theorem in the technical report.

Acquisition histories have been introduced for the analysis of parallel pushdown systems without process creation [5, 4], in a setting where it is sufficient to regard only two parallel processes. They have been applied for conflict analysis of DPNs [10], where it is also sufficient to regard just two parallel processes at a time. Our generalization of acquisition histories is non-trivial, as we have to consider unboundedly many parallel processes and process creation.

An efficient implementation of acquisition history based techniques for a model without process creation is currently studied in a joint work of one of the authors [7]. However, the focus of that work lies on exploiting symbolic techniques to get an efficient implementation, while the focus of this paper is the generalization to DPNs. The use of symbolic techniques for getting an efficient implementation is briefly discussed in the conclusion section.

Locks can be simulated by shared memory, and thus context bounded model-checking techniques for DPNs [1] can also handle locks. However, the number of lock operations is also limited by the context bound, while our technique handles unboundedly many lock operations. Moreover, the technique presented in [1] is based on predecessor computations, hence it is straightforward to extend it with our results, getting a model-checking algorithm that can handle boundedly many accesses to shared memory and unboundedly many lock operations.

2 Dynamic Pushdown Networks

Dynamic pushdown networks (DPNs) [2] are a model for parallel processes with potentially recursive procedures and process creation. They extend pushdown processes by the ability to create new processes. A DPN is a tuple $M =$

(P, Γ, L, Δ), 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. For the rest of this paper, we assume that we have a fixed DPN $M = (P, \Gamma, L, \Delta)$.

Interleaving Semantics. We briefly recall the interleaving semantics of a DPN as presented in [2]: Configurations $\mathsf{Conf}_M := (P\Gamma^*)^*$ 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 $\rightarrow_M \subseteq \mathsf{Conf}_M \times L \times \mathsf{Conf}_M$ is the least solution of the following constraints:

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

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 $\rightarrow_M^* \subseteq \mathsf{Conf}_M \times L^* \times \mathsf{Conf}_M$ to be the reflexive, transitive closure of $\rightarrow_M$. 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.

Example 1 (DPN-Execution). Consider a DPN with non-spawning rules $\Delta_N = \{p_1\gamma \overset{l_1}{\hookrightarrow} p_1\gamma_1\gamma_2, p_1\gamma_3 \overset{l_3}{\hookrightarrow} p_1, p_2\gamma \overset{l_4}{\hookrightarrow} p_2\gamma_2\gamma_3\}$ and spawning rules $\Delta_S = \{p_1\gamma_1 \overset{l_2}{\hookrightarrow} p_1\gamma_3 \rhd p_2\gamma\}$. It has the execution: $p_1\gamma \xrightarrow{l_1} p_1\gamma_1\gamma_2 \xrightarrow{l_2} p_2\gamma p_1\gamma_3\gamma_2 \xrightarrow{l_4} p_2\gamma_2\gamma_3 p_1\gamma_3\gamma_2 \xrightarrow{l_3} p_2\gamma_2\gamma_3 p_1\gamma_2$.

Tree Semantics. The interleaving semantics involves two types of nondeterministic choice: First, there may be more than one rule with a left-hand side matching the state of a process. Second, there may be more than one process that can make a step. In each step, the interleaving semantics nondeterministically chooses a process and a matching rule. We now separate these two types of nondeterministic choice: In a first step, we just fix the choice of the applied rules. The interleaving is then chosen in a second step.

The interleaving semantics models an execution as a sequence of steps, i.e. a total ordering of steps. We now model an execution as a partial ordering of steps, that totally orders steps of the same process, and additionally orders steps of a created process to come after the step that created the process. However, it does not order steps running in parallel. This ordering has a tree shape (cf.

Fig. 1, showing the partial ordering of steps corresponding to the execution in Example 1). Formally, we model an execution starting at a single process as an *execution tree* of type $T_M ::= \mathsf{N}\ L\ T_M \mid \mathsf{S}\ L\ T_M\ T_M \mid \mathsf{L}\ P\Gamma^*$. 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.

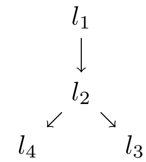


Fig. 1: Partial Ordering of Steps

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

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

An execution that starts at a configuration with multiple processes is modeled as a list of execution trees (called an *execution hedge*), with one tree per process. These executions are defined by overloading $\Longrightarrow_M$ with:

$$\begin{array}{lll}
c \overset{t_1 \ldots t_n}{\Longrightarrow}_M c' & :\Leftrightarrow & \exists c'_1 \ldots c'_n \in \mathsf{Conf}_M,\ p_1 \ldots p_n \in P,\ w_1 \ldots w_n \in \Gamma^*. \\
 & & \quad c = p_1 w_1 \ldots p_n w_n \ \wedge\ c' = c'_1 \ldots c'_n \\
 & & \wedge\ p_1 w_1 \overset{t_1}{\Longrightarrow}_M c'_1 \ \wedge \ldots \wedge\ p_n w_n \overset{t_n}{\Longrightarrow}_M c'_n
\end{array}$$

Intuitively, $c \overset{h}{\Longrightarrow}_M c'$ means that there is an execution from configuration c to configuration c' with execution hedge h. Figure 2 shows an execution tree t satisfying $p_1\gamma \overset{t}{\Longrightarrow}_M p_2\gamma_2\gamma_3 p_1\gamma_2$ for the DPN from Example 1. The execution tree of the spawned process is always drawn as the left successor of the S-node, and the corresponding edge is labeled with ,,spawn". Hence, the rightmost steps in an execution tree corresponds to the execution of the process at the root of the tree.

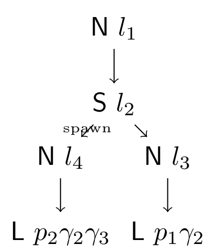


Fig. 2: Execution Tree

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

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

We call $\mathsf{sched}(h) := \{\bar{l} \in L^* \mid \exists h' \in (\mathsf{L}\ P\Gamma^*)^*.\ h \overset{\bar{l}}{\rightsquigarrow}{}^* h'\}$ the set of *schedules* of a hedge $h \in T_M^*$, where $(\mathsf{L}\ P\Gamma^*)^*$ is the set of all hedges that solely consist of L-nodes (i.e. have no more rules to execute), and $\rightsquigarrow^*$ is the reflexive, transitive closure of the scheduler $\rightsquigarrow$. It can be shown by straightforward induction that every execution of the interleaving semantics is a schedule of an execution of the tree semantics and vice versa:

Theorem 2 (Thm. sched-correct in Sec. 4 of [9]). *Let $c, c' \in \mathsf{Conf}_M$ and $\bar{l} \in L^*$, then $c \xrightarrow{\bar{l}}{}_M^* c'$ if and only if there is an execution hedge $h \in T_M^*$ with $c \overset{h}{\Longrightarrow}_M c'$ and $\bar{l} \in \mathsf{sched}(h)$.*

Predecessor Sets. Given a set $C' \subseteq \mathsf{Conf}_M$ of configurations, the set $\mathsf{pre}_M(C') := \{c \mid \exists c' \in C',\ l \in L.\ c \xrightarrow{l}_M c'\}$ 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, $\mathsf{pre}_M^*(C') := \{c \mid \exists c' \in C',\ \bar{l} \in L^*.\ c \xrightarrow{\bar{l}}{}_M^* c'\}$ 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, because it allows to cascade pre_M and pre_M^* computations and pose regular constraints on the intermediate configurations. For example, liveness of a global variable x at a control location u can be decided [2]: Let M be the DPN that models the program, $M|_{\mathsf{read}}$ be the DPN M restricted to those rules that correspond to program statements that read x, and $M|_{\mathsf{nowrite}}$ be the DPN M restricted to those rules that do not write x. Moreover, let $\mathsf{at}(u) \subseteq \mathsf{Conf}_M$ be the (regular) set of configurations where some process is at control location[2] u. Finally, let $c_0 \in \mathsf{Conf}_M$ be the initial configuration of the system. The variable x is live at control node u if and only if: $c_0 \in \mathsf{pre}_M^*(\mathsf{at}(u) \cap \mathsf{pre}_{M|_{\mathsf{nowrite}}}^*(\mathsf{pre}_{M|_{\mathsf{read}}}(\mathsf{Conf}_M)))$. In this fashion, all forward and backward kill/gen-bitvector problems can be decided in polynomial time. Other important analyses that can be done by predecessor computations are

[2] Usually, a control location is identified with the symbol at the top of the stack, such that $\mathsf{at}(u) = (P\Gamma^*)^* P u \Gamma^* (P\Gamma^*)^*$.

context-bounded model-checking of DPNs with shared memory [1], and checking of reachability properties (e.g. data races).

Hedge-Constrained Predecessor Sets. In the introduction we indicated that reachability w.r.t. regular constraints on the interleaved executions is undecidable. Now, we constrain the execution hedges: For a set of configurations $C' \subseteq \mathsf{Conf}_M$ and a set of execution hedges $H \subseteq T_M^*$, we define the *H-constrained predecessor set* of C' as: $\mathsf{pre}_M[H](C') := \{c \in \mathsf{Conf}_M \mid \exists c' \in C',\ h \in H.\ c \overset{h}{\Longrightarrow}_M c'\}$, i.e. those configurations that can make a transition with an execution hedge $h \in H$ to a configuration $c' \in C'$. We show that if H is (tree-)regular and C' is regular, then $\mathsf{pre}_M[H](C')$ is regular as well, and given a hedge automaton for H and an automaton for C', an automaton for $\mathsf{pre}_M[H](C')$ can be computed in polynomial time.

Hedge Automata. In order to characterize regular sets of execution hedges, we define a type of hedge automata [3] adjusted to acceptance of execution hedges: A hedge automaton $\mathcal{H} = (S, \mathcal{A}_0, D)$ consists of a finite set of states S, an initial automaton $\mathcal{A}_0$ with $L(\mathcal{A}_0) \subseteq S^*$, and a set $D = D_L \cup D_N \cup D_S$ of transition rules. Rules $s \hookrightarrow \mathcal{A} \in D_L$ consist of a state $s \in S$ and an automaton $\mathcal{A}$ that accepts process configurations, i.e. $L(\mathcal{A}) \subseteq P\Gamma^*$. These rules are used to label L-nodes with a state. Rules $s \overset{l}{\hookrightarrow} s' \in D_N$ with $s, s' \in S$ and $l \in L$ are used to label N-nodes and rules $s \overset{l}{\hookrightarrow} s' \rhd s_s \in D_S$ with $s, s_s, s' \in S$ and $l \in L$ are used to label S-nodes[3].

A hedge $h = t_1 \dots t_n \in T_M^*$ is accepted iff the trees $t_1, \dots, t_n$ can be labeled bottom-up according to the rules in D such that the sequence of states $s_1, \dots, s_n$ that label the roots of $t_1, \dots, t_n$ is accepted by the initial automaton $\mathcal{A}_0$. Formally, we define the relation $\mathsf{lab}_{\mathcal{H}} \subseteq S \times T_M$ as the least relation satisfying the following rules:

$$
\begin{array}{lll}
[\mathsf{leaf}] & \mathsf{lab}_{\mathcal{H}}(s, \mathsf{L}\ pw) & \text{if}\ \ s \hookrightarrow \mathcal{A} \in D_L\ \wedge\ pw \in L(\mathcal{A}) \\
[\mathsf{nospawn}] & \mathsf{lab}_{\mathcal{H}}(s, \mathsf{N}\ l\ t) & \text{if}\ \ s \overset{l}{\hookrightarrow} s' \in D_N\ \wedge\ \mathsf{lab}_{\mathcal{H}}(s', t) \\
[\mathsf{spawn}] & \mathsf{lab}_{\mathcal{H}}(s, \mathsf{S}\ l\ t_s\ t) & \text{if}\ \ s \overset{l}{\hookrightarrow} s' \rhd s_s \in D_S\ \wedge\ \mathsf{lab}_{\mathcal{H}}(s_s, t_s)\ \wedge\ \mathsf{lab}_{\mathcal{H}}(s', t)
\end{array}
$$

$\mathsf{lab}_{\mathcal{H}}(s, t)$ means that the root of the tree t can be labeled by the state s. We overload $\mathsf{lab}_{\mathcal{H}} \subseteq \bigcup_{n \in \mathbb{N}} S^n \times T_M^n$ for hedges by: $\mathsf{lab}_{\mathcal{H}}(s_1 \dots s_n, t_1 \dots t_n)\ :\Leftrightarrow \mathsf{lab}_{\mathcal{H}}(s_1, t_1) \wedge \dots \wedge \mathsf{lab}_{\mathcal{H}}(s_n, t_n)$ and define the language of the hedge automaton $\mathcal{H}$ by $L(\mathcal{H}) := \{h \mid \exists \bar{s} \in L(\mathcal{A}_0).\ \mathsf{lab}_{\mathcal{H}}(\bar{s}, h)\}$.

Assume we have a DPN $M = (P, \Gamma, L, \Delta)$, a regular set of configurations $C' \subseteq \mathsf{Conf}_M$, and a hedge automaton $\mathcal{H}$ that accepts a set of hedges $L(\mathcal{H}) \subseteq T_M^*$. In order to compute $\mathsf{pre}_M[L(\mathcal{H})](C')$, we define a new DPN $M \times \mathcal{H} = (P \times S, \Gamma, L, \Delta')$, a new regular set of configurations $C' \times \mathcal{H} \subseteq \mathsf{Conf}_{M \times \mathcal{H}}$, and an operator $\mathsf{proj}_{\mathcal{H}} : 2^{\mathsf{Conf}_{M \times \mathcal{H}}} \to 2^{\mathsf{Conf}_M}$, such that $\mathsf{pre}_M[L(\mathcal{H})](C') = \mathsf{proj}_{\mathcal{H}}(\mathsf{pre}^*_{M \times \mathcal{H}}(C' \times \mathcal{H}))$. The

[3] A more standard notation of the rules would be $\mathcal{A}() \hookrightarrow s$, $l(s') \hookrightarrow s$, and $l(s_s, s') \hookrightarrow s$. However, our notation emphasizes the relation to DPN-rules.

constructions of $M \times \mathcal{H}$ and $C' \times \mathcal{H}$, as well as the operator $\mathsf{proj}_{\mathcal{H}}$, are effective, such that we can compute $\mathsf{pre}_M[L(\mathcal{H})](C')$ (for a given automaton for C') using the saturation algorithm for pre^* presented in [2]. The idea of this construction is to encode the states of the hedge automaton into the states of the DPN, and simulate the transitions of the hedge automaton within the transitions of the DPN. The new set of configurations $C' \times \mathcal{H}$ reflects the application of the D_L-rules of the hedge automaton, and the $\mathsf{proj}_{\mathcal{H}}$-operation removes configurations not compatible with the initial automaton $\mathcal{A}_0$, and projects the control states of $M \times \mathcal{H}$ back to states of M. The rules $\Delta' = \Delta'_N \cup \Delta'_S$ of the DPN $M \times \mathcal{H}$ are defined as follows:

$$\begin{array}{lll}
[\mathsf{nospawn}] & (p,s)\gamma \stackrel{l}{\hookrightarrow} (p',s')w \in \Delta'_N & \text{iff } \; p\gamma \stackrel{l}{\hookrightarrow} p'w \in \Delta_N \\
 & & \quad \wedge \;\; s \stackrel{l}{\hookrightarrow} s' \in D_N \\
[\mathsf{spawn}] & (p,s)\gamma \stackrel{l}{\hookrightarrow} (p',s')w \rhd (p_s,s_s)w_s \in \Delta'_S & \text{iff } \; p\gamma \stackrel{l}{\hookrightarrow} p'w \rhd p_s w_s \in \Delta_S \\
 & & \quad \wedge \;\; s \stackrel{l}{\hookrightarrow} s' \rhd s_s \in D_S
\end{array}$$

Notice that this definition generates $O(|\Delta|D)$ rules and can be implemented in time $O(|\Delta|D)$. The new set of configurations $C' \times \mathcal{H}$ is defined as:

$$\begin{aligned}
C' \times \mathcal{H} := \{(p_1,s_1)w_1 \dots (p_n,s_n)w_n \mid\ & p_1 w_1 \dots p_n w_n \in C' \\
& \wedge \;\; \forall 1 \le i \le n.\ \exists \mathcal{A}.\ s_i \hookrightarrow \mathcal{A} \in D_L \;\; \wedge \;\; p_i w_i \in L(\mathcal{A})\}
\end{aligned}$$

Finally, we define the projection operator as:

$$\begin{aligned}
\mathsf{proj}_{\mathcal{H}}(C_\times) := \{p_1 w_1 \dots p_n w_n \mid\ & \exists s_1, \dots, s_n \in S.\ s_1 \dots s_n \in L(\mathcal{A}_0) \\
& \wedge \;\; (p_1,s_1)w_1 \dots (p_n,s_n)w_n \in C_\times\}
\end{aligned}$$

We can show that $\mathsf{pre}_M[L(\mathcal{H})](C') = \mathsf{proj}_{\mathcal{H}}(\mathsf{pre}^*_{M\times\mathcal{H}}(C' \times \mathcal{H}))$ (Theorem *xdpn-correct* in Section 8 of [9]). Moreover, we can write $C' \times \mathcal{H}$ and $\mathsf{proj}_{\mathcal{H}}(C_\times)$ using only operations that preserve regularity and are computable in polynomial time for given automata (Theorem *projH-effective* in Section 8 of [9]). With the polynomial time algorithm for computing pre^*_M [2] we get the following theorem:

Theorem 3 (Thm. prehc-effective[4] in Sec. 8 of [9]). *Given a DPN M, a hedge automaton $\mathcal{H}$ and a regular set of configurations C', the set $\mathsf{pre}_M[L(\mathcal{H})](C')$ is regular and given an automaton for C', an automaton for $\mathsf{pre}_M[L(\mathcal{H})](C')$ can be computed in polynomial time.*

3 Lock-Sensitive DPNs

Locks are an important synchronization mechanism to guarantee mutual exclusion, e.g. to protect accesses to shared resources. Processes may *acquire* and

[4] This Theorem follows from Theorem *prehc-effective* in [9], some well-known results about tree-automata and the results from [2]. However, the tree-automata results and the results from [2] are not formally proven in [9].

release locks, but, at any time, each lock may be owned by at most one process. If a process wants to acquire a lock currently owned by another process, it blocks until the lock is released. We now extend DPNs by a finite number of locks and assume that they are acquired and released in a well-nested fashion, i.e. a process must release locks in the opposite order of acquisition. We assume locks to be non-reentrant, that is a process must not re-acquire a lock that it already owns. Note that we can simulate re-entrant locks with non-reentrant ones if the acquisition and release of locks is aligned with procedure calls [6] (like monitors and Java synchronized blocks).

Model Definition. Let $\mathbb{L}$ be a finite set of locks. We label the rules by their actions w.r.t. the locks in $\mathbb{L}$, i.e. the labels[5] are $L ::= \mathsf{none} \mid \mathsf{acq}\ \mathbb{L} \mid \mathsf{rel}\ \mathbb{L}$. As mentioned, we assume that locks are accessed in a well-nested and non-reentrant fashion. In general there may exist executions from (unreachable) configurations that violate the well-nestedness or non-reentrance assumptions. Hence, we fix a start configuration $p_0\gamma_0 \in P\Gamma$, and make these assumptions only for executions from the start configuration. Moreover, we assume that, initially, a process does not own any locks, i.e. no execution from the start configuration or of a spawned process releases a lock that it has not acquired before. Note that it is straightforward to decide whether a given DPN satisfies our assumptions.

In order to simplify the presentation, we assume that the set of currently acquired locks is visible in the control state of a process, i.e. we assume that there is a function $\mathsf{locks} : P \to 2^{\mathbb{L}}$ that maps control states to the set of allocated locks. This function must satisfy the constraints $\mathsf{locks}(p_0) = \emptyset$ and

$$p\gamma \overset{l}{\hookrightarrow} p'w[\rhd\, p_s w_s] \in \Delta \;\Rightarrow\; \mathsf{locks}(p) \xrightarrow{l}\!\!\rhd \mathsf{locks}(p')\ [\wedge\ \mathsf{locks}(p_s) = \emptyset]$$

where $\rightarrow\!\!\rhd \subseteq 2^{\mathbb{L}} \times L \times 2^{\mathbb{L}}$ describes valid transitions on the set of acquired locks:

$$\begin{array}{lll}
[\mathsf{none}] & X \xrightarrow{\mathsf{none}}\!\!\rhd X' & :\Leftrightarrow X = X' \\
[\mathsf{acquire}] & X \xrightarrow{\mathsf{acq}\ x}\!\!\rhd X' & :\Leftrightarrow x \notin X \wedge X' = \{x\} \cup X \\
[\mathsf{release}] & X \xrightarrow{\mathsf{rel}\ x}\!\!\rhd X' & :\Leftrightarrow x \notin X' \wedge X = \{x\} \cup X'
\end{array}$$

Note that DPNs where locks are not visible in the control state can be transformed to DPNs with a locks-function, involving a blowup that is, in the worst case, exponential in the maximum nesting depth of locks, which is typically small. We overload the locks-function to configurations by: $\mathsf{locks}(p_1w_1 \dots p_nw_n) := \mathsf{locks}(p_1) \cup \dots \cup \mathsf{locks}(p_n)$. Also note that for an execution hedge h, all configurations c where h can start (i.e. $\exists c'.\ c \overset{h}{\Longrightarrow}_M c'$) hold the same set of locks. This set can be derived from the annotation of the final configurations at the leafs and the lock operations at the inner nodes. For an execution hedge h, we overload $\mathsf{locks}(h)$ to be the set of locks held by any configuration where h starts, i.e. we have $c \overset{h}{\Longrightarrow}_M c' \;\Rightarrow\; \mathsf{locks}(c) = \mathsf{locks}(h)$.

[5] Usually, one wants to make visible additional information in the labels. This could be done by using pairs of lock-actions and additional information as labels. However, we omit such a definition here for clarity.

The lock-sensitive step relation is defined as $c \xrightarrow{l}_{\mathsf{ls},M} c' \quad :\Leftrightarrow \quad c \xrightarrow{l}_M c' \wedge \mathsf{locks}(c) \overset{l}{\rhd} \mathsf{locks}(c')$. As usual, we write $\rightarrow^*_{\mathsf{ls},M}$ for its reflexive, transitive closure.

The extension of the tree-based semantics to locks can be described as a restriction of the scheduler. The $\Longrightarrow_M$-relation, that relates configurations to execution hedges, does not change. The original scheduler maps an execution hedge to all its topological sorts. However, we cannot model the set of lock-sensitive schedules as topological sorts. For example, Consider two processes that execute the steps 1:$\mathsf{acq}\ x$, 2:$\mathsf{rel}\ x$ and 3: $\mathsf{acq}\ x$, 4 : $\mathsf{rel}\ x$ for some lock x. When they are executed in parallel, the possible schedules are $1, 2, 3, 4$ and $3, 4, 1, 2$. However, if these would be topological sorts of some ordering, so would be the schedule $1, 3, 2, 4$, which is invalid.

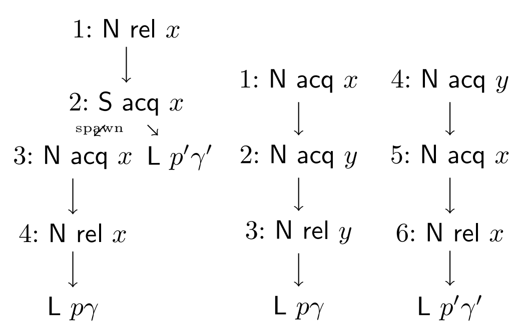
The lock-sensitive scheduler $\rightsquigarrow_{\mathsf{ls}} \subseteq T_M^* \times L \times T_M^*$ is a restriction of the original scheduler to those schedules that respect the semantics of locks:

$$h \overset{l}{\rightsquigarrow}_{\mathsf{ls}} h' :\Leftrightarrow h \overset{l}{\rightsquigarrow} h' \wedge \mathsf{locks}(h) \overset{l}{\rhd} \mathsf{locks}(h')$$

The set of *lock-sensitive schedules* of a hedge $h \in T_M^*$ is defined as: $\mathsf{sched}_{\mathsf{ls}}(h) := \{\bar{l} \mid \exists h' \in (\mathsf{L}\ P\Gamma^*)^*.\ h \overset{\bar{l}}{\rightsquigarrow}{}^*_{\mathsf{ls}} h'\}$, where $\rightsquigarrow^*_{\mathsf{ls}}$ is the reflexive, transitive closure of the lock-sensitive scheduler $\rightsquigarrow_{\mathsf{ls}}$. It is again straightforward to show that the lock-sensitive scheduler matches the interleaving semantics:

Theorem 4 (Thm. lsched-correct in Sec. 9 of [9]). *Let $c, c' \in \mathsf{Conf}_M$ and $\bar{l} \in L^*$, then $c \xrightarrow{\bar{l}}{}^*_{\mathsf{ls},M} c'$ if and only if there exists an execution hedge $h \in T_M^*$ with $c \overset{h}{\Longrightarrow}_M c'$ and $\bar{l} \in \mathsf{sched}_{\mathsf{ls}}(h)$.*

Acquisition Structures. As shown above, the lock-sensitive schedules cannot be characterized as the set of topological sorts of some order. However, in order to compute lock-sensitive predecessor sets, we do not need to consider every single schedule of an execution hedge, but just the existence of a schedule. We now introduce a finite state abstraction of execution hedges from which we can decide whether a lock sensitive schedule exists or not. Our approach generalizes *acquisition histories* [5, 4].



(a) (b)

Fig. 3: Executions With Locks

In order to sketch the basic idea, we have to introduce some wording: An execution hedge that starts at a reachable configuration is called *reachable*. Note that, due to the well-nestedness and non-reentrance assumptions, reachable execution hedges also use locks in a well-nested and non-reentrant fashion. Given a (reachable) execution hedge, an acquisition of a lock x without a matching release is called a *final acquisition* of x. A matching release means, that the same process releases the lock, i.e. that there is a release of x on the rightmost path starting at the acquisition node. Symmetrically, a release of x without a matching acquisition is called an *initial release*. Acquisitions and releases that are not final acquisitions or initial releases are called *usages*. For example consider the execution tree from Figure 3a: Node 1 is an initial release of the lock x, Node 2 is a final acquisition of x, and Nodes 3 and 4 are usages of x.

Given a reachable execution hedge h, we define its *release graph* $g_r(h) \subseteq \mathbb{L}\times\mathbb{L}$ and its *acquisition graph* $g_a(h) \subseteq \mathbb{L} \times \mathbb{L}$: $g_r(h)$ contains an edge $x \to x'$, iff h contains an initial release of x' and the ordering induced by h enforces a usage of x to be scheduled before the initial release of x'. Symmetrically, $g_a(h)$ contains an edge $x \to x'$ iff h contains a final acquisition of x that has to be scheduled before a usage of x'. We now identify some necessary criteria for a reachable execution hedge h to have a lock-sensitive schedule:

1. All locks that are used or finally acquired in h are either also initially released in h or are not contained in the initial set of locks $\mathsf{locks}(h)$.
2. h does not contain multiple final acquisitions of the same lock.
3. The acquisition graph and the release graph of h are acyclic.

To justify Criterion 1, we observe that all locks in $\mathsf{locks}(h)$ that are not initially released remain allocated throughout the entire execution. Hence, no acquisition (neither a usage nor a final acquisition) of such a lock can be scheduled. The intuition behind Criterion 2 is, that a finally acquired lock will not be freed for the rest of the execution. Hence, if there are multiple final acquisitions of the same lock, only one of them can be scheduled. Note that we do not check the symmetric condition for initial releases, as reachable execution hedges cannot contain multiple initial releases of the same lock. To illustrate Criterion 3, assume that the acquisition graph $g_a(h)$ has a cycle $x_1 \to x_2 \to \ldots \to x_n = x_1$ with $n > 1$. Then, h has no lock sensitive schedule: From the definition of the acquisition graph, we see that, first, each lock $x_i, i < n$ has a final acquisition in h (because it has an outgoing edge in $g_a(h)$) and, second, that this final acquisition of x_i precedes a usage of x_{i+1} in each schedule of h. In a lock sensitive schedule, this usage of x_{i+1} must preceed the final acquisition of x_{i+1}. Because of the cycle $x_1 \to \ldots \to x_n = x_1$ this would imply that the final acquisition of x_1 precedes itself, which is a contradiction. The argumentation for a cycle in the release graph is similar. For example, the execution tree depicted in Figure 3a has no schedule, as the usage of x (Nodes 3 and 4) cannot be scheduled before the final acquisition of x (Node 2). Its acquisition graph $\{x \to x\}$ is trivially cyclic. The execution hedge from Figure 3b also has no schedule: We have to schedule the final acquisition of x (Node 1) or y (Node 4) as first step. However, if we schedule Node 1 first, we will not be able to schedule the usage of x (Nodes 5 and 6) and, symmetrically, if we schedule Node 4 first, we will not be able to schedule the usage of y (Nodes 2 and 3). The acquisition graph is $\{x \to y, y \to x\}$, which has the cycle $x \to y \to x$.

An *acquisition structure* is a finite domain abstraction of an execution hedge that can be computed inductively over the execution hedge and contains enough information to decide the criteria 1-3 depicted above. An acquisition structure is either the special symbol $\perp$ or a five-tuple (r, u, a, g_r, g_a), where $r \subseteq \mathbb{L}$ is the set of initially released locks, $u \subseteq \mathbb{L}$ is the set of used locks, and $a \subseteq \mathbb{L}$ is the set of finally acquired locks. $g_r, g_a \subseteq \mathbb{L} \times \mathbb{L}$ are the acquisition and release graphs. We define AS to be the set of all acquisition structures.

$\mathsf{as}(\mathsf{L}\ pw) := (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$
$\mathsf{as}(\mathsf{N}\ l\ t) := \mathsf{upd}(l, \mathsf{as}(t))$
$\mathsf{as}(\mathsf{S}\ l\ t_s\ t) := \mathsf{upd}(l, \mathsf{as}(t_s) \parallel \mathsf{as}(t))$
$\mathsf{as}(t_1 \dots t_n) := \mathsf{as}(t_1) \parallel \dots \parallel \mathsf{as}(t_n)$

$\mathsf{upd}(\mathsf{none}, s) := s$
$\mathsf{upd}(n, \perp) := \perp$
$\mathsf{upd}(\mathsf{acq}\ x, (r, u, a, g_r, g_a)) := \mathsf{if}\ x \in r\ \mathsf{then}$
$\quad (r \setminus \{x\}, u \cup \{x\}, a,$
$\quad\quad (g_r \setminus (\mathbb{L} \times \{x\})) \cup (\{x\} \times r \setminus \{x\}), g_a)$
$\mathsf{else\ if}\ x \notin a\ \mathsf{then}$
$\quad (r, u, a \cup \{x\}, g_r, g_a \cup (\{x\} \times u))$
$\mathsf{else}\ \perp$
$\mathsf{upd}(\mathsf{rel}\ x, (r, u, a, g_r, g_a)) := (r \cup \{x\}, u, a, g_r, g_a)$

$\_ \parallel \perp := \perp$
$\perp \parallel \_ := \perp$
$(r, u, a, g_r, g_a) \parallel (r', u', a', g'_r, g'_a) :=$
$\mathsf{if}\ r \cap r' = \emptyset \wedge a \cap a' = \emptyset\ \mathsf{then}$
$\quad (r \cup r', u \cup u', a \cup a', g_r \cup g'_r, g_a \cup g'_a)$
$\mathsf{else}\ \perp$

Fig. 4: Definition of acquisition structures

Acquisition structures can be computed bottom-up along the structure of an execution hedge. Figure 4 shows the computation in a pseudo-functional language. The $\mathsf{upd}(l, \_)$-function describes the effect of prepending a node labeled with l to the acquisition structure. The $\parallel$-operator composes the acquisition structures of two processes that are executed in parallel. An acquisition structure $(r, u, a, g_r, g_a) \in \mathsf{AS}$ is called *consistent* w.r.t. a set of initial locks $X \subseteq \mathbb{L}$, if and only if $(X \setminus r) \cap (u \cup a) = \emptyset$ and both, g_r and g_a, are acyclic. This models Criteria 1 and 3. The acquisition structure $\perp$ is not consistent w.r.t. any set of locks. This models Criterion 2, as the acquisition structure becomes $\perp$ if it is violated. We define $\mathsf{cons}(X)$ to be the set of all acquisition histories that are consistent w.r.t. X. We can show the following theorem:

Theorem 5 (Thm. acqh-correct in Sec. 11 of [9]). *For any reachable execution hedge h, we have $\mathsf{sched}_{\mathsf{ls}}(h) \neq \emptyset$ if and only if $\mathsf{as}(h) \in \mathsf{cons}(\mathsf{locks}(h))$.*

The proof of the $\Rightarrow$-direction follows the intuition described above. For the proof of the $\Leftarrow$-direction, a valid schedule of an execution hedge with a consistent acquisition history is constructed inductively. For the details we refer to [9]. Note that Theorem 5 implies that the criteria 1-3 are not only necessary, but also sufficient for existence of a lock-sensitive schedule of an execution hedge.

Encoding Acquisition Structures into Hedge Automata. In this section, we show that the set of all execution hedges that have a lock-sensitive schedule is regular. For this purpose, we design a hedge automaton $\mathcal{H}_{\mathsf{ls}}$ that computes $\mathsf{locks}(h)$ and $\mathsf{as}(h)$ within its states, and checks consistency by its initial automaton. The Definition of $\mathcal{H}_{\mathsf{ls}}$ is shown in Figure 5. The rules label each tree t with $\mathsf{locks}(t)$ and $\mathsf{as}(t)$. The initial automaton $\mathcal{A}_0$ computes the locks and the acquisition structure of the hedge. A hedge h is accepted iff $\mathsf{as}(h)$ is consistent w.r.t. $\mathsf{locks}(h)$. The rules of the hedge-automaton use the function $\mathsf{eff}^{-1} : L \times 2^{\mathbb{L}} \to 2^{\mathbb{L}}$ for bottom-up computation of $\mathsf{locks}(h)$. It describes the reverse effect of a lock operation. With the definitions above and Theorem 5 we get the following theorem:

Theorem 6 (Thm. reachable-hls-char in Sec. 14 of [9]). *For any reachable configuration c and execution hedge h that starts at c (i.e. $\exists c'.\ c \xRightarrow{h}_M c'$), we have $\mathsf{sched}_{\mathsf{ls}}(h) \neq \emptyset$ if and only if $h \in L(\mathcal{H}_{\mathsf{ls}})$.*

Cascaded, lock-sensitive predecessor sets can now be computed using the lock-insensitive, hedge-constrained predecessor computation presented in the first part of this paper: A lock-sensitive predecessor set is computed by $\mathsf{pre}_M[\mathcal{H}_{\mathsf{ls}} \cap \mathcal{H}_{\mathsf{addc}}]$, where $\mathcal{H}_{\mathsf{addc}}$ may contain additional constraints on the execution hedge, like restrictions of the rule labels or the number of applied rules (e.g. to compute immediate predecessor sets). Note that Theorem 6 is only applicable for reachable configurations. Hence, we have to ensure that we only derive information from reachable configurations during cascaded predecessor set computations. However, we observe that most applications of cascaded predecessor computations are used to query whether the start configuration $p_0\gamma_0$ is contained in the outermost predecessor set, i.e. they have the shape $p_0\gamma_0 \in \mathsf{pre}_M[\mathcal{H}_{\mathsf{ls}} \cap \_](\ldots \mathsf{pre}_M[\mathcal{H}_{\mathsf{ls}} \cap \_](\_) \ldots)$. By definition, only reachable configurations can be reached from the start configuration. Thus, the result of such a query depends only on reachable configurations in the intermediate predecessor sets, and, using Theorem 6, we can show that the result is sound and precise w.r.t. the lock-sensitive semantics.

$$
\begin{aligned}
\mathcal{H}_{\mathsf{ls}} &:= (2^{\mathbb{L}} \times \mathsf{AS}, \mathcal{A}_0, D_L \cup D_N \cup D_S)\\
\mathcal{A}_0 &:= (2^{\mathbb{L}} \times \mathsf{AS}, q_0, F_0, \delta_0)\\
q_0 &:= (\emptyset, (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset))\\
F_0 &:= \{(X, s) \mid s \in \mathsf{cons}(X)\}\\
\delta_0 &:= \{(X, s) \xrightarrow{(X', s')} (X \cup X', s \parallel s') \mid\\
&\qquad X, X' \subseteq \mathbb{L} \ \wedge\ s, s' \in \mathsf{AS}\}\\
D_L &:= \{(\mathsf{locks}(p), (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)) \hookrightarrow \{p\}\Gamma^* \mid p \in P\}\\
D_N &:= \{(X, \mathsf{upd}(l, s)) \overset{l}{\hookrightarrow} (\mathsf{eff}^{-1}(l, X), s) \mid\\
&\qquad X \subseteq \mathbb{L},\ s \in \mathsf{AS},\ l \in L\}\\
D_S &:= \{(X, \mathsf{upd}(l, s_s \parallel s)) \overset{l}{\hookrightarrow} (\mathsf{eff}^{-1}(l, X), s) \rhd (\emptyset, s_s) \mid\\
&\qquad X \subseteq \mathbb{L},\ s, s_s \in \mathsf{AS},\ l \in L\}\\
\mathsf{eff}^{-1}(\mathsf{none}, X) &:= X\\
\mathsf{eff}^{-1}(\mathsf{acq}\ x, X) &:= X \cup \{x\}\\
\mathsf{eff}^{-1}(\mathsf{rel}\ x, X) &:= X \setminus \{x\}
\end{aligned}
$$

Fig. 5: Definition of $\mathcal{H}_{\mathsf{ls}}$.

Complexity. The problem of deciding whether the initial configuration is contained in the lock-sensitive predecessor set of some regular set of configurations is NP-hard, even if we store the set of allocated locks in the control states of the processes and have no procedures at all. This can be shown by a reduction from the boolean satisfiability problem (SAT). The reduction uses the same idea as [10], but uses multiple processes instead of recursive procedures. Hence, we cannot expect a polynomial time algorithm. And indeed, the number of states of the automaton $\mathcal{H}_{\mathsf{ls}}$ is exponential in the number of locks.

4 Conclusion

We have presented a tree-based semantics for DPNs, where executions are modeled as hedges, which reflect the ordering of steps of a single process and the causality due to process creation, but enforce no ordering between steps of processes running in parallel. We have shown how to efficiently compute predecessor sets of regular sets of configurations with tree-regular constraints on the execution hedges. Our algorithm encodes a hedge-automaton into the DPN, thus reducing the problem to unconstrained predecessor set computation, for which a polynomial time algorithm exists. In the second part of this paper, we have presented a generalization of acquisition histories to DPNs, and used it to construct tree-regular constraints for lock-sensitive executions. With the techniques from the first part of this paper, we obtained an algorithm to precisely compute cascaded, lock-sensitive predecessor sets.

As many analyses of DPNs are based on the computation of predecessor sets, it is now straightforward to make them lock-sensitive by substituting the original, lock-insensitive predecessor computation by our new, lock-sensitive one. For example, we can construct lock-sensitive bitvector analyses, and extend context-bounded analysis of DPNs with shared memory [1], as discussed in the introduction.

Future Research. A main direction of future research is implementation and experimental evaluation of the techniques presented in this paper. Possible starting points may be [8] and [7]. In [8], Java programs are abstracted to an intermediate model based on pushdown systems without process creation, and then analyzed using a semi-decision procedure. The techniques in this paper provide a decision procedure for an intermediate model that is more suited to the abstraction of Java programs, as thread creation can be abstracted more precisely. In a joint work of one of the authors [7], an acquisition history based decision procedure for the original intermediate model of [8], that does not support process creation, has been constructed. It uses many optimizations to increase the efficiency of the implementation. The most notable optimization is that the consistency check is not performed after each intermediate computation of a cascaded computation (which is likely to exponentially blow up the size of the automaton), but is delayed until all intermediate computations are completed. For this purpose, the analysis works with vectors of so called *extended acquisition histories*. Moreover, instead of encoding information into the control states, weighted pushdown systems [12] are used. Similar optimizations may apply to our technique, using vectors of acquisition structures and weighted DPNs [13], such that there is hope that an implementation might be practical despite of the NP-hardness of the problem.

Acknowledgment. We thank Ahmed Bouajjani, Nicholas Kidd, Thomas Reps, and Tayssir Touili for helpful and inspiring discussions on analysis of concurrent pushdown systems with and without process creation.

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. A. Bruggemann-Klein, M. Murata, and D. Wood. Regular tree and regular hedge languages over unranked alphabets. Research report, 2001.
4. 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.
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. N. Kidd, A. Lal, and T. Reps. Language strength reduction. In *Proc. of SAS'08*, volume 5079 of *LNCS*. Springer, 2008.
7. N. Kidd, P. Lammich, T. Touili, and T. Reps. A decision procedure for detecting atomicity violations for communicating processes with locks. Submitted for publication.
8. N. Kidd, T. Reps, J. Dolby, and M. Vaziri. Finding concurrency-related bugs using random isolation. In *Proc. of VMCAI'09*, volume 5403 of *LNCS*, 2009.
9. P. Lammich. Isabelle formalization of hedge-constrained pre* and DPNs with locks. Available from `http://cs.uni-muenster.de/sev/publications/`. Technical Report.
10. P. Lammich and M. Müller-Olm. Conflict analysis of programs with procedures, dynamic thread creation, and monitors. In *Proc. of SAS'08*, volume 5079 of *LNCS*. Springer, 2008.
11. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
12. T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. Comput. Program.*, 58(1-2):206–263, 2005.
13. A. Wenner. Optimale Analyse gewichteter dynamischer Push-Down Netzwerke. Master's thesis, University of Münster, August 2008.