Contextual Locking for Dynamic Pushdown Networks *

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

[1] Technische Universität München, Germany
{lammich,seidl}@in.tum.de
[2] Institut für Informatik, Westfälische Wilhelms-Universität Münster, Germany
{markus.mueller-olm,alexander.wenner}@wwu.de

Abstract. Contextual locking is a scheme for synchronizing between possibly recursive processes that has been proposed by Chadha et al. recently. Contextual locking allows for arbitrary usage of locks within the same procedure call and Chadha et al. show that control-point reachability for two processes adhering to contextual locking is decidable in polynomial time. Here, we complement these results. We show that in presence of contextual locking, control-point reachability becomes PSPACE-hard, already if the number of processes is increased to three. On the other hand, we show that PSPACE is both necessary and sufficient for deciding control-point reachability of k processes for $k > 2$, and that this upper bound remains valid even if dynamic spawning of new processes is allowed. Furthermore, we consider the problem of regular reachability, i.e., whether a configuration within a given regular set can be reached. Here, we show that this problem is decidable for recursive processes with dynamic thread creation and contextual locking. Finally, we generalize this result to processes that additionally use a form of join operations.

1 Introduction

Analysing parallel programs is notoriously hard, especially in the presence of procedures and synchronisation. Ramalingam showed that even simple safety properties like reachability for programs with synchronous communication and procedures are undecidable [17]. The same holds for mutual exclusion via locks [10]. Undecidability can be avoided by using abstraction to over-approximate reachability [2] or by considering restricted classes of executions only to under-approximate reachability [5, 16]. Identifying synchronization patterns where exact reachability is decidable remains a challenging problem.

Chadha et al. propose contextual locking, where arbitrary locking may occur as long as it does not cross procedure boundaries [4]. On the one hand, this constraint on lock usage is shown to lead to a decidable simultaneous reachability problem for two processes. On the other hand Chadha et al. demonstrate that

* This work was partially funded by the DFG project OpIAT (Optimal Interprocedural Analysis of Programs with Thread Creation, MU 1508/1 and SE 551/13).

it is suitable to model common locking patterns and we refer the reader to their paper for a detailed justification of contextual locking and comparison with other locking schemes. The contribution of this paper is to extend their result to more general settings.

The main observation of Chadha et al. is, that it suffices to regard executions where the procedure calls of both processes occur well-nested. This reduces the problem of checking reachability for two processes to reachability of a single pushdown process. However, already for three processes, this reduction is no longer possible. Consider the following three processes T_1, T_2, T_3 with procedures P, Q using contextual locking:

$$
\begin{aligned}
T_1 &: \mathsf{rel}(B)_5;\ \mathsf{acq}(B)_8;\ \mathsf{rel}(C)_9;\ \mathsf{acq}(C)_{12};\\
T_2 &: P;\ \mathsf{rel}(D)_{13};\ \mathsf{acq}(D)_{16};\\
T_3 &: \mathsf{rel}(A)_1;\ \mathsf{acq}(A)_4;\ Q;\\
\\
P &: \mathsf{acq}(A)_2;\ \mathsf{rel}(A)_3;\ \mathsf{acq}(C)_{10};\ \mathsf{rel}(C)_{11};\\
Q &: \mathsf{acq}(B)_6;\ \mathsf{rel}(B)_7;\ \mathsf{acq}(D)_{14};\ \mathsf{rel}(D)_{15};
\end{aligned}
$$

Assume that the processes hold locks $\{B, C\}$, $\{D\}$ and $\{A\}$, respectively, at the beginning of the execution. Using further locks, this can be ensured even when starting with the empty set of locks for each process. All three processes can reach the end simultaneously, for example by following the annotated schedule. However, in any execution where all processes reach the end, the calls to procedures P and Q are necessarily non-nested. Lock A forces P to start before Q, locks B and C require Q to start before P ends and lock D ensures that Q only ends after P has ended. Therefore, in general, reachability of multi-pushdown processes cannot be easily reduced to reachability of a single pushdown process.

We show that simultaneous reachability for systems with at least three processes is PSPACE-complete (§2). Furthermore, the problem remains PSPACE-complete for systems with dynamic thread creation (§3), which better fit the concepts of real languages like Java and C with pthreads. While simultaneous reachability focuses on the states of constantly many processes in the configuration, regular reachability concerns the whole configuration, and thus is more natural for systems with an unbounded number of processes. By exploiting well-quasi orderings, similar to [6], for suitably abstracted configurations, we show that regular reachability is decidable for systems with dynamic thread creation (§4) and joins (§5) as additional synchronisation primitive.

In related work Bonnet et al. have recently combined contextual locking with reentrant locking [1]. In this setting simultaneous reachability is shown to be decidable under bounded context switches. Kahlon et al. have shown, that simultaneous reachability for two processes becomes decidable when locks are only used in a well-nested fashion, i.e., when the last lock acquired always is the first lock to be released [10]. This result was later generalized to systems with dynamic process creation [13], regular reachability [14, 12], and joins [7]. Furthermore, Kahlon et al. generalized their result to bounded lock-chains, where nesting of locking may be violated, as long as chains of overlapping lock regions

are bounded. In this setting, simultaneous reachability for two processes is still decidable [8, 9].

2 The Static Case

In [4] an algorithm is presented that decides whether two given control-states are simultaneously reachable by a system of two possibly recursive processes which use contextual locking. Their algorithm is exponential in the number of locks, but polynomial in the size of the program. It remained open whether and how this approach can be generalized to more than two threads. In the following, we provide an algorithm for deciding a slightly more general problem, namely, reachability of a control-sequence, for systems of k recursive processes with $k \geq 1$ which runs in PSPACE. Moreover we show that the original algorithm cannot be easily generalized to more than two threads by showing that reachability already for three processes is PSPACE hard—even for a constant number of locks.

We consider a multi-pushdown system P with a fixed number $k \in \mathbb{N}$ of processes with a shared finite set of locks $\mathcal{L}$. Each process $i \in \{1, \dots k\}$ maintains a thread-local state which is of the form (q, X) where q is from a finite set Q of process-local information and $X \subseteq \mathcal{L}$ is the set of currently held locks. Furthermore, we are given a finite set Γ of local information of possibly called procedures. Thus, each $\gamma \in \Gamma$ encodes the name of the current procedure together with a finite amount of information about the local state of the current call to the procedure. Accordingly, the current call-stack (or pushdown) of the process is represented by a sequence $w = \gamma_1 \dots \gamma_k \in \Gamma^*$. For convenience, we assume that the top of the pushdown is on the left side, i.e., equals γ_1. A configuration of a single process thus is given by a pair $((q, X), w)$ where $q \in Q, X \subseteq \mathcal{L}, w \in \Gamma^*$. Each process is defined by a finite set of rules of the form:

$$
\begin{array}{ll}
r : (q, \gamma) \xrightarrow{\tau} (q', \gamma') & \text{(computation step)} \\
r : (q, \gamma) \xrightarrow{\tau} (q', \gamma_1 \gamma_2) & \text{(procedure call)} \\
r : (q, \gamma) \xrightarrow{\tau} (q', \epsilon) & \text{(procedure exit)} \\
r : (q, \gamma) \xrightarrow{\mathsf{acq}(l)} (q', \gamma') & \text{(acquire lock } l \in \mathcal{L}) \\
r : (q, \gamma) \xrightarrow{\mathsf{rel}(l)} (q', \gamma') & \text{(release lock } l \in \mathcal{L})
\end{array}
$$

where r is a unique identifier for the corresponding rule, and τ is the empty label. The rules define the effects of actions onto the thread configuration, i.e., the process-local state, the (left end of the) call-stack and the effect onto the set of currently held locks. Let $\mathsf{eff}(r)$ be the effect on the set of locks of a rule r. A step $((q, X), \gamma w) \overset{r}{\Rightarrow} ((q', X'), w'w)$ on local configurations is defined if there is a rule $r : (q, \gamma) \xrightarrow{e} (q', w')$ and $X' = X \cup \{l\}$ with $l \notin X$ if $\mathsf{eff}(r) = \mathsf{acq}(l)$ or $X' = X \setminus \{l\}$ with $l \in X$ if $\mathsf{eff}(r) = \mathsf{rel}(l)$ or $X = X'$ if $\mathsf{eff}(r) = \tau$. For a sequence $\pi = r_1 \dots r_m$ of rules, we write $[\![\pi]\!]((q, X), w) = ((q', X'), w')$ if the sequence $r_1, \dots, r_m$ of rules is successively executable starting from the thread configuration $((q, X), w)$ and results in the configuration $((q', X'), w')$.

A process adheres to *contextual locking* if lock operations do not cross procedure boundaries, i.e. a lock acquired during a procedure call must be released in the same procedure call and no procedure called in the meantime may release it temporarily. Formally for all $X \subseteq \mathcal{L}$ and every call rule $(q, \gamma) \xrightarrow{\tau} (q', \gamma_1\gamma_2)$ and all sequences π with $[\![\pi]\!]((q', X), \gamma_1) = ((q'', X'), w)$, the following holds:

1. $X \subseteq X'$;
2. if $w = \epsilon$, then $X = X'$.

A configuration of a multi-pushdown system with k processes is a sequence

$$t = ((q_1, X_1), w_1) \dots ((q_k, X_k), w_k)$$

where we assume that the sets X_i of locks are pairwise disjoint. W.l.o.g. we may assume that all processes share the same set of rules, but may differ in their respective start configurations. For an initial configuration we assume $X_i = \emptyset$ and $w_i \in \Gamma$ for all $i \in \{1, \dots, k\}$. An execution of the system can be considered as an interleaving of executions of the participating threads $i \in \{1, \dots, k\}$. In order to distinguish the action of one process from the same action of another process, we identify actions by means of pairs (i, r) where i identifies the process and r the performed action. A sequence Π of pairs (i, r) starting from configuration t is *executable* resulting in configuration t', if either $\Pi = \epsilon$ and $t' = t$, or $\Pi = \Pi'(i, r)$ and the following holds:

1. Π' is executable for t resulting in $t'' = ((q_1'', X_1''), w_1) \dots ((q_k'', X_k''), w_k)$;
2. rule r is applicable *thread-locally* to the configuration $((q_i'', X_i''), w_i'')$ of the ith process resulting in some process configuration $((q_i', X_i'), w_i')$;
3. if $\mathsf{eff}(r) = \mathsf{acq}(l)$, then lock l is also *globally* available, i.e., $l \notin X_1'' \cup \dots \cup X_k''$;
4. $t' = ((q_1', X_1'), w_1') \dots ((q_k', X_k'), w_k')$, where $((q_j', X_j'), w_j') = ((q_j'', X_j''), w_j'')$ for $j \neq i$.

In this case, we write $t' = [\![\Pi]\!]\, t$. A configuration t' is *reachable* from a configuration t if $t' = [\![\Pi]\!]\, t$ for some global execution sequence Π. Likewise, a *set* T of configurations is reachable from t iff there is a configuration $t' \in T$ such that t' is reachable from t. We now extend simultaneous control-state reachability of two processes to *control-state sequence* reachability of k processes and formulate our first result:

Theorem 1. *Assume that $t = ((q_1, \emptyset), \gamma_1) \dots ((q_k, \emptyset), \gamma_k)$ is the initial configuration and $\sigma = (q_1', X_1') \dots (q_k', X_k')$ is a sequence of process-local states of length k. Then it is decidable in PSPACE for processes which adhere to contextual locking, whether the set*

$$T = \{((q_1', X_1'), w_1') \dots ((q_k', X_k'), w_k') \mid w_i' \in \Gamma^*\}$$

is reachable from t or not.

The main observation that leads to a PSPACE algorithm is that reachability is preserved, if only executions are considered where the sizes of occurring pushdowns are polynomially bounded. Intuitively, the pushdown of a process grows

whenever a procedure is called. For every such call, two cases can be distinguished. In the first case, the called procedure never returns. In this case, the pushed return location is dead, it will never make it to the top of the pushdown again and thus can be discarded. In the second case, the called procedure eventually returns. Thus, the pushdown grows only temporarily. In presence of recursion, the pushdown still may grow arbitrarily. In the following we therefore show, that in the case of deeply nested recursive calls that eventually return, the execution can be transformed into a shorter execution that still preserves reachability, but uses strictly smaller pushdowns.

For $i = 1, \ldots, k$, let proj_i denote the homomorphism which extracts from a global execution sequence Π, with $t' = [\![\Pi]\!]\, t$, the execution sub-sequence of the ith process, i.e. the homomorphism proj_i is defined by $\mathsf{proj}_i(i, r) = r$ and $\mathsf{proj}_i(i', r) = \epsilon$ for $i \neq i'$. In particular, $((q_i', X_i'), w_i') = [\![\mathsf{proj}_i(\Pi)]\!]\, ((q_i, X_i), w_i)$ if $((q_i, X_i), w_i)$ and $((q_i', X_i'), w_i')$ are the configurations of the ith process in t and t', respectively. The proof of Theorem 1 then is based on the following sequence of lemmas. Lemma 2 allows to discard return information of non-returning procedure calls by introducing new rules in the pushdown, that allow to effectively inline such a procedure call.

Lemma 2. *Given a multipushdown-system P, a system P' can be constructed such that any control sequence $\sigma = (q_1', X_1') \ldots (q_k', X_k')$ is reachable from an initial configuration t in P iff the sequence $\sigma' = (\langle q_1', \top\rangle, X_1') \ldots (\langle q_k', \top\rangle, X_k')$ is reachable from the initial configuration t in P' and all pushdowns are empty in the final configuration.*

Proof. The set of states of the new system consists of all old states and additional states $\langle q, \bot\rangle, \langle q, \top\rangle$. The set of pushdown symbols contains all old symbols in addition to new symbols $\langle \gamma, \#\rangle$. The system non-deterministically decides whether the execution will return to a level in the pushdown. The lowest level which will be visited again is marked by # in the pushdown. Since symbols below this level will never be at the top of the pushdown again, we construct the system to remove them directly, thus # marks the bottom of the pushdown in the new system. $\bot, \top$ in the state mark whether the pushdown is empty or not. The new set of rules consists of transitions $r : (q, \gamma) \xrightarrow{\tau} (\langle q, \bot\rangle, \langle \gamma, \#\rangle)$ which add the markers to the initial configuration. Furthermore, we have a rule $r' : (\langle q, \bot\rangle, \gamma) \xrightarrow{e} (\langle q', \bot\rangle, w')$ working above the marker in the pushdown for each rule $r : (q, \gamma) \xrightarrow{e} (q', w')$ of the old system. Additionally, we add rules that apply to the marked pushdown symbol. Computation- and lock-steps preserve the position of the marker, thus we add $r' : (\langle q, \bot\rangle, \langle \gamma, \#\rangle) \xrightarrow{e} (\langle q', \bot\rangle, \langle \gamma', \#\rangle)$ for every rule $r : (q, \gamma) \xrightarrow{e} (q', \gamma')$. Return below the marked level empties the pushdown and ends the execution, thus we add new rules $r' : (\langle q, \bot\rangle, \langle \gamma, \#\rangle) \xrightarrow{\tau} (\langle q', \top\rangle, \varepsilon)$ that reach a corresponding final state for each rule $r' : (q, \gamma) \xrightarrow{\tau} (q', \varepsilon)$. In case of a call-transition, the system non-deterministically decides whether it will return from the newly pushed symbol or not. For each call-transition $r : (q, \gamma) \xrightarrow{\tau} (q', \gamma_1\gamma_2)$ we add one rule $r_1' : (\langle q, \bot\rangle, \langle \gamma, \#\rangle) \xrightarrow{\tau} (\langle q', \bot\rangle, \gamma_1\langle \gamma_2, \#\rangle)$ that decides that the call is returning and thus preserves the position of the marker. A second rule

$r_2' : (\langle q, \bot\rangle, \langle \gamma, \#\rangle) \xrightarrow{\tau} (\langle q', \bot\rangle, \langle \gamma_1, \#\rangle)$ decides that the call is non-returning, moves the marker and discards the lower pushdown symbol by only pushing the upper symbol. To be able to reach a configuration inside a procedure with an empty pushdown, we finally add rules $r' : (\langle q, \bot\rangle, \langle \gamma, \#\rangle) \xrightarrow{\tau} (\langle q, \top\rangle, \varepsilon)$, that may terminate an execution by emptying a pushdown of size one, preserving the control state. The claim follows by induction on the length of an execution. The size of the resulting system only increases by a constant factor from the size of the original system.

Remark 3. Instead of using $\bot$ and $\#$ in the construction of Lemma 2, one can also use this annotation to store information about the discarded pushdown. For example one can impose a regular constraint on each pushdown in the final configuration. To this end we use states s, s' of a given automaton $\mathcal{A}$ over pushdown symbols and propagate the state when discarding a pushdown symbol, i.e., only add call rules $r_2' : (\langle q, s\rangle, \langle \gamma, s'\rangle) \xrightarrow{\tau} (\langle q', s\rangle, \langle \gamma_1, s''\rangle)$ of the second kind, where (s'', γ_2, s') is a transition of $\mathcal{A}$, return rules $r' : (\langle q, s\rangle, \langle \gamma, s'\rangle) \xrightarrow{\tau} (\langle q', \top\rangle, \varepsilon)$ where $s = s'$ and rules $r' : (\langle q, s\rangle, \langle \gamma, s'\rangle) \xrightarrow{\tau} (\langle q, \top\rangle, \varepsilon)$ ending the computation where (s, γ, s') is a transition of $\mathcal{A}$. By additionally requiring that s is an initial and s' a final state of $\mathcal{A}$ in rules $r : (q, \gamma) \xrightarrow{\tau} (\langle q, s\rangle, \langle \gamma, s'\rangle)$ for the initial marking, we ensure that reaching a final state implies that the discarded pushdown has an accepting run in the automaton.

Lemma 4 shows that we may disregard nested returning procedure calls, that are executed in a similar context. Recently, a similar statement was developed independently by Bonnet et al. for the main proof of [1].

Lemma 4. *Assume that $t' = [\![\Pi]\!]\, t$ for global configurations t, t' where the configuration of the ith process in t is given by $((q_i, X_i), w_i)$. Assume further that there is a call rule $r : (q, \gamma) \xrightarrow{\tau} (q', \gamma_1\gamma_2)$ together with a state p such that the following holds:*

- *$\mathsf{proj}_i(\Pi)$ can be written as $c_1 r \pi c_2$ with $((q, X), \gamma w) = [\![c_1]\!]\,((q_i, X_i), w_i)$ for some w where $[\![\pi]\!]\,((q', X), \gamma_1) = (p, \epsilon)$; and furthermore,*
- *$\pi = u_1 r \pi' u_2$ such that $((q, X'), \gamma w') = [\![u_1]\!]\,((q', X), \gamma_1)$ for some w' where $[\![\pi']\!]\,((q', X'), \gamma_1) = (p, \epsilon)$.*

Consider a factorization of the global execution $\Pi = C_1(i, r)U_1(i, r)\Pi' U_2 C_2$ with $\mathsf{proj}_i(C_j) = c_j$, $\mathsf{proj}_i(U_j) = u_j$ for $j \in \{1, 2\}$ and $\mathsf{proj}_i(\Pi') = \pi'$. Assume that for $j \in \{1, 2\}$, U_j' is obtained from U_j by removing all steps of the ith process. Then the sequence $C_1 U_1'(i, r)\Pi' U_2' C_2$ is an execution for t which also results in t'.

Proof. Let t_1 denote the configuration which is reached by the global execution C_1. In particular, X is the set of locks held by the ith process in t_1. We proceed by considering longer and longer prefixes of the executions. Let V and V' denote a prefix of U_1 and the corresponding prefix of U_1', respectively. By induction on the length of V, we prove that

- The set X is included in the set of locks held by the ith process in the configuration $[\![(i, r)V]\!]\, t_1$.

– V' is executable and the set X equals the set of locks held by the ith process in the configuration $[\![V']\!]\, t_1$.

Now consider the second occurrence of the call transition r of the ith process. We have proven so far, that in particular, $X \subseteq X'$. For all other processes, the sets of acquired locks after the executions $(i,r)U_1(i,r)$ and $U_1'(i,r)$ agree, since these processes have executed the same sequences of actions. Let $t_2 = [\![(i,r)U_1(i,r)]\!]\, t_1$ and $t_2' = [\![U_1'(i,r)]\!]\, t_1$. Due to contextual locking, the local execution π' of the ith process does not depend on any lock being in X' and only acquires locks that are not in X', thus it may also execute with the smaller initial set of locks X. Therefore, Π' is executable both in configurations t_2 and t_2' resulting in configurations t_3 and t_3', respectively. Since the processes adhere to contextual locking, the sets of locks held by the ith process in configurations t_3 and t_3', respectively, equal again X' and X, respectively. Now let V and V' denote a prefix of U_2 and the corresponding prefix of U_2', respectively. By induction on the length of V, we prove that

– The set X is included in the set of locks held by the ith process in the configuration $[\![V]\!]\, t_3$.
– V' is executable and the set X equals the set of locks held by the ith process in the configuration $[\![V']\!]\, t_3'$.

Due to contextual locking, the set of locks held by the ith process in configuration $[\![U_2]\!]\, t_3$ precisely equals X. It follows that the two configurations $[\![U_2]\!]\, t_3$ and $[\![U_2']\!]\, t_3'$ coincide. Accordingly, $C_1U_1'(i,r)U_2'C_2$ is a global execution sequence for t, and the configurations $[\![\Pi]\!]\, t$ and $[\![C_1U_1'(i,r)\Pi'U_2'C_2]\!]\, t$ agree.

Proof (Theorem 1). We can now essentially reduce the problem to checking reachability of a finite state system, whose configurations have polynomial size. Instead of checking reachability in the original system we check for reachability with an empty pushdown in the modified system of Lemma 2. According to Lemma 4, we can eliminate nested returning procedure calls, which have the same initial state q, pushdown symbol γ and final state p. It follows by a simple counting argument, that reachability can be checked using bounded pushdowns of size $O(|Q|^2 \cdot |\Gamma|)$, since each execution using a larger pushdown can be transformed into one using a smaller pushdown.

We now show that the PSPACE algorithm to establish the decidability of reachability in Theorem 1 cannot be improved in general. In fact, we show that control sequence reachability is PSPACE-hard already for three processes using contextual locking with a constant number of locks only. Note that this is in sharp contrast with the result of [4] for two processes with contextual locking where an upper bound is obtained which is polynomial in the size of the processes and exponential only in the number of locks.

Theorem 5. *For three processes using contextual locking with a constant number of locks, control sequence reachability is PSPACE-hard.*

Proof. The construction of the three processes builds on the observation that the set of successful runs of a linear space-bounded Turing Machine can be represented as an intersection $L_1 \cap L_2$ of two languages L_i over an alphabet of fixed size, each of which can be accepted by a pushdown automaton of polynomial size, that uses its pushdown in a disciplined fashion.

Configurations of a linear space-bounded Turing Machine, i.e. the contents of the tape together with the current control state, can be represented by words of fixed length $m = k \cdot (n+1)$ over a binary alphabet, where n is the space-bound and k depends logarithmically on the size of the alphabet and the number of control states of the Turing Machine. The control state is inserted to the left of the current position of the head on the tape and each tape symbol and the state of the Turing Machine is encoded using k bits.

A word of the language L_1 is a sequence of subwords of length m, where the first subword encodes an initial and the last subword is the reverse of a final configuration of the Turing Machine and the $(2l+1)$-th subword is the reverse of the $(2l+2)$-th subword. The language L_2 consists of words, where each word is again a sequence of subwords of length m and the $(2l+1)$-th subword is now the reverse encoding of a configuration reachable from the configuration encoded by the $2l$-th subword in one step of the Turing Machine.

We can construct two pushdown processes which accept the languages L_1 and L_2, respectively, together with an additional finite state process that checks the intersection. Instead of formally realizing these three processes as a multi-pushdown system, we prefer to use a more intuitive notation by means of programs with procedures. In our construction, reading a bit $i \in \{0,1\}$ is simulated by temporarily acquiring the lock A_i associated with that bit using $\mathsf{use}(A_i)$. We write $\mathsf{use}(Z) = \mathsf{acq}(Z); \mathsf{rel}(Z)$ for short for a lock Z. The third process tries to enforce that both pushdown processes read the same bit by only allowing access to one bit at a time. This is achieved by blocking all locks and only temporarily releasing the one associated with the intended bit using $\mathsf{free}(A_i)$, where we write $\mathsf{free}(Z) = \mathsf{rel}(Z); \mathsf{acq}(Z)$ for a lock Z.

This mechanism, though, is not yet sufficient to synchronize the two pushdown processes. The third process may allow a series of bits, but it is not ensured that each of these bits is read by both pushdown processes or that a pushdown does not use one release to read the same bit twice. The second problem can be solved by introducing an additional lock B. Reading a bit $i \in \{0,1\}$ is then represented by $\mathsf{use}(A_i); \mathsf{use}(B)$ and allowing a bit i to be processed by $\mathsf{free}(A_i); \mathsf{free}(B)$. Since one occurrence of $\mathsf{use}(A_i)$ is no longer directly followed by another one, two separate uses can no longer be associated with the same operation $\mathsf{free}(A_i)$.

Solving the first problem is more intricate. A first idea would be to use the same mechanism in reverse and introduce locks that are blocked by the pushdown processes and are meant to be acquired by the synchronizing process. These could be used after each bit to prevent the synchronizing process from going ahead before both pushdown processes have read the proposed bit. This, however, would violate contextual locking, since the pushdown processes would have to block these locks from the start and only release them temporarily after

each bit, which in general, occurs in a context different from the initial context. The second idea therefore is to exploit the disciplined pushdown usage of the two pushdown processes. Both processes read words consisting of pairs of subwords of a fixed length m. Each pair of subwords is independent from the next. Thus, the pushdown processes can be constructed in a way that they return to the initial context after reading $2m$ bits. In order to implement this idea, we introduce locks S_1, S_2, R_1, R_2 that synchronize the third process with the two pushdown processes exactly every $2m$ steps. In the following, we present the programs for each of the three processes.

The synchronizing process does not use push- or pop-operations and thus can be represented by a finite-state program:

$$
\begin{array}{rl}
 & \mathsf{acq}(A_0); \mathsf{acq}(A_1); \mathsf{acq}(B); \\
 & \mathsf{use}(Z_1); \mathsf{use}(Z_2); \mathsf{acq}(Y); \\
s_3 : & \textbf{while}\,(*)\,\{ \\
 & \quad ((\mathsf{free}(A_0) \vee \mathsf{free}(A_1)); \mathsf{free}(B))^m; \mathsf{use}(R_1); \mathsf{use}(R_2); \\
 & \quad ((\mathsf{free}(A_0) \vee \mathsf{free}(A_1)); \mathsf{free}(B))^m; \mathsf{use}(S_1); \mathsf{use}(S_2) \\
 & \} \\
l_3 : & \text{// program point to be reached}
\end{array}
$$

The processes reading L_1 and L_2 are given by:

$$
\begin{array}{rl}
 & \mathsf{acq}(R_1); \mathsf{acq}(R_2); \\
 & \mathsf{acq}(Z_1); \mathsf{use}(Y); \\
s_1 : & \mathsf{checkInput}; \mathsf{free}(R_1); \mathsf{free}(R_2); \\
 & \textbf{while}\,(*)\,\{ \\
 & \quad \mathsf{checkRev}; \mathsf{free}(R_1); \mathsf{free}(R_2) \\
 & \} \\
 & \mathsf{checkFinal}; \\
l_1 : & \text{// point to be reached}
\end{array}
\qquad
\begin{array}{rl}
 & \mathsf{acq}(S_1); \mathsf{acq}(S_2); \\
 & \mathsf{acq}(Z_2); \mathsf{use}(Y); \\
s_2 : & \textbf{while}\,(*)\,\{ \\
 & \quad \mathsf{checkStep}; \mathsf{free}(S_1); \mathsf{free}(S_2) \\
 & \} \\
l_2 : & \text{// point to be reached}
\end{array}
$$

The locks Y, Z_1, Z_2 and their usage pattern enforce, that all processes first have to reach their starting label s_i, and thus acquire the initial set of locks required to block the other processes. The sub-routines $\mathsf{checkInput}$ and $\mathsf{checkFinal}$ for verifying the first and last configurations, respectively, can be implemented by a finite-state program in a straight-forward way. The sub routines $\mathsf{checkRev}$ and $\mathsf{checkStep}$ can be implemented as follows, using procedures C_i with $0 \leq i \leq m$ and P_i with $3 \leq i \leq n+1$:

$$
\begin{array}{rl}
\mathsf{checkRev} : & C_m \\
C_i : & (\mathsf{use}(A_0); \mathsf{use}(B); C_{i-1}; \mathsf{use}(A_0); \mathsf{use}(B)) \\
 & \vee (\mathsf{use}(A_1); \mathsf{use}(B); C_{i-1}; \mathsf{use}(A_1); \mathsf{use}(B)) \\
C_0 : & \mathsf{skip} \\
\mathsf{checkStep} : & P_{n+1} \\
P_i : & \bigvee\{\mathsf{read}(a); P_{i-1}; \overline{\mathsf{read}}(a) \mid a \text{ is tape symbol}\} \\
 & \vee \bigvee\{\mathsf{read}(a_1a_2a_3); C_{(i-3)\cdot k}; \overline{\mathsf{read}}(b_1b_2b_3) \mid (a_1a_2a_3, b_1b_2b_3) \text{ is a step}\}
\end{array}
$$

We write $\mathsf{read}(w) = \mathsf{use}(A_{i_0}); \mathsf{use}(B); \ldots \mathsf{use}(A_{i_j}); \mathsf{use}(B);$ for reading the binary encoding $i_0 \ldots i_j$ of a word w over tape symbols and states of the Turing Machine

and $\overline{\mathsf{read}}(w)$ for the same operation using the reverse of the encoding of w. A pair $(a_1a_2a_3, b_1b_2b_3)$ is a step of the Turing Machine if a_2 is the control state, a_1, a_3 are tape symbols and $b_1b_2b_3$ describes the rewritten portion of the configuration after a step, including the movement of the head.

The description should have made it clear that, starting from their initial configurations with empty sets of held locks, the control-sequence

$$(l_1, \{R_1, R_2, Z_1\})(l_2, \{S_1, S_2, Z_2\})(l_3, \{A_0, A_1, B, Y\})$$

will be reachable if and only if the simulated Turing Machine has an accepting computation for the given initial configuration.

3 Spawning of New Processes

In this section, we show how the algorithm for control sequence reachability from the last section can be enhanced to multi-pushdown systems where new processes can be dynamically spawned. Programs now additionally may have transitions of the form:

$$r : (q, \gamma) \xrightarrow{(q_1, \gamma_1)} (q_2, \gamma_2) \qquad \text{(spawn step)}$$

where the effect of the transition is the spawning of a new thread with the initial configuration $((q_1, \emptyset), \gamma_1)$. The function $\mathsf{eff}(r)$ is extended accordingly. The pair (q_2, γ_2) on the right-hand side describes the continuation of the process executing this step. The local step relation $\dot{\Rightarrow}$ affects the current process similar to a compute-rule. The global step relation additionally extends the sequence of processes which are concurrently running, by one more process. As in [3, 7], we find it convenient to keep track of the ancestry between processes. For that, each local configuration of a process is equipped with an extra component which is meant to hold all successively spawned processes. Thus, a global configuration is now a rooted tree $(h, (q, X), w)$ where, as before, q is a process state, X is a finite set of locks, $w \in \Gamma^*$ is the pushdown and h is a (possibly empty) sequence of sub-trees representing the child processes. Again, we additionally demand that the different occurrences of sets of locks in a global configuration are mutually disjoint. Initial configurations consist only of a single process and are of the form $(\epsilon, (q, \emptyset), \gamma)$. In order to identify sub-configurations within a global configuration t we use sequences of positive integers called positions. In particular, ϵ is a position in t and the sub-configuration of t at position ϵ, denoted by t/ϵ, equals t itself. Furthermore, if $t = (t_1 \dots t_k, (q, X), w)$ and η is a position of t_i for $i = 1, \dots, k$, then $i\eta$ is a position in t with $t/i\eta = t_i/\eta$. Likewise, if $t = (t_1 \dots t_k, (q, X), w)$, then the root process of t, i.e. the process in t at position ϵ, denoted by $t[\epsilon]$, has the process-local configuration $((q, X), w)$ and has successively spawned the root processes of $t_1, \dots, t_k$. We write $t[\eta]$ for the root process of t/η, i.e. $t[\eta] = t/\eta[\epsilon]$.

Now, global steps are rules applied to sub-configurations. A global step (η, r) transforms a global configuration t into t', if the following holds:

- $t/\eta = (h, (q, X), w)$, $((q, X), w) \overset{r}{\dot{\Rightarrow}} ((q', X'), w')$ and $t'/\eta = (h', (q', X'), w')$

– $l \notin X''$ for all local configurations $t[\eta'] = ((q'', X''), w'')$, if $\mathsf{eff}(r) = \mathsf{acq}(l)$
– if $\mathsf{eff}(r) = (q_1, \gamma_1)$ then $h' = h(\varepsilon, (q_1, \emptyset), \gamma_1)$ else $h' = h$
– all other sub-configurations t/η' are preserved

In this case, we denote the resulting configuration also as $t' = [\![(\eta, r)]\!]\, t$ and extend the notation to sequences Π of global steps. Note that each newly created process initially holds the empty set of locks. A multipushdown system with dynamic process generation by means of spawn-rules has been called *dynamic pushdown network* or DPN [3]. The DPN adheres to contextual locking if each process of the DPN does so. We first consider reachability of a control sequence of a fixed length for DPNs. This means that we require only a subset of the processes to reach certain control states simultaneously. A control sequence

$$\sigma = (q'_1, X'_1) \dots (q'_k, X'_k)$$

is reachable from a global configuration t if a global configuration t' is reachable such that $t'[\eta_i] = ((q'_i, X'_i), w'_i)$ for a suitable sequence $\eta_1 < \ldots < \eta_k$ of positions in t' where the ordering $<$ on positions is given by the *left-right* ordering within the textual representation of t', i.e. $\eta < \eta'$ if η' is a proper prefix of η or $\eta = \eta_0 j \eta_1$ and $\eta' = \eta_0 j' \eta_2$ where $j < j'$. In this case, we also say that t' is *compatible* with the control sequence σ at positions $\eta_1, \dots, \eta_k$. The main theorem of this section is:

Theorem 6. *For every DPN P with contextual locking and control-sequence σ it is decidable in PSPACE whether or not σ is reachable from an initial configuration $(\epsilon, (q, \emptyset), \gamma)$ of the DPN.*

Since the lower-bound result from the last section also applies to DPNs, we conclude that control sequence reachability for DPNs is in fact, PSPACE-complete.

The key observation for the PSPACE upper bound is, that for reachability of a control sequence σ only steps of processes at one of the positions in the control sequence, or ancestors of such a process, must be considered.

Assume that $t_0 = (\epsilon, (p, \emptyset), \gamma)$ is an initial configuration of a DPN and $t = [\![\Pi]\!]\, t_0$. Then we call a position η *inactive* w.r.t. a global execution sequence Π, if Π does not contain any step (η, r) and thus also no step $(\eta\eta', r)$ for any η'. The following lemma can be proven by induction on the length of prefixes of Π.

Lemma 7. *Assume that t is compatible with the $\sigma = (q_1, X_1) \dots (q_k, X_k)$ at positions $\eta_1, \dots, \eta_k$. Let Π' denote the subsequence of Π which is obtained from Π by removing all steps (η, r) where η is* not *a prefix of any of the η_i. Then the following holds:*

1. *$t' = [\![\Pi']\!]\, t_0$ is still a global configuration which is compatible with the given control-sequence σ at positions $\eta_1, \dots, \eta_k$.*
2. *Every position η of t' is either inactive or a prefix of one of the η_i.* □

Let us call the configuration t' together with the global execution sequence Π' which is constructed according to Lemma 7, *purified* w.r.t. the control-sequence

σ. A purified global execution sequence may still be further reduced while preserving compatibility with the given control-sequence. For that, we first add transitions that skip spawning of inactive processes altogether.

For a given DPN P, consider the DPN P' which is obtained from P by adding a transition $r' : (q, \gamma) \xrightarrow{\tau} (p_2, \gamma_2)$ for every transition $r : (q, \gamma) \xrightarrow{(p_1,\gamma_1)} (p_2, \gamma_2)$. The resulting DPN has the same number of states and pushdown symbols as P and at most twice as many transitions. We have:

Lemma 8. *Consider a non-empty control sequence $\sigma = (q_1, X_1) \dots (q_k, X_k)$. Let $t_0 = (\epsilon, (q_0, \emptyset), \gamma_0)$ be an initial configuration. Then the following statements are equivalent:*

1. *a configuration t is reachable from t_0 w.r.t. P which is compatible with σ;*
2. *a configuration t' is reachable from t_0 w.r.t. P' which is compatible with σ;*
3. *a configuration t'' is reachable from t_0 w.r.t. P' which is compatible with σ at positions $\eta_1, \dots, \eta_k$ where t' has no inactive processes w.r.t. these positions.*

Proof. Assertion (2) follows from assertion (1) since every execution of DPN P is also an execution of DPN P'. Assertion (3) follows from assertion (2) in two stages. First, we may assume by Lemma 7 w.l.o.g. that the global execution sequence is purified. Then this global execution sequence is modified in such a way that spawning of inactive processes is replaced with the corresponding basic computation step which avoids the new process but preserves the process local successor state and pushdown. Note that not spawning inactive processes may cause a decrease in the number of spawned processes and thus may change the addresses of corresponding processes. Finally, given a global execution reaching t'' from t_0 w.r.t. DPN P' which is compatible with σ and does not spawn inactive processes, a global execution of DPN P can be recovered which is still compatible with σ essentially by introducing spawn-operations r again for the corresponding compute-operations r'. The additionally created processes will be treated as inactive processes.

Henceforth, we call an execution sequence according to statement (3) of Lemma 8 *strongly* purified. In a strongly purified execution, a process may still have an arbitrary number of ancestors. Thus still an arbitrary number of processes would have to be tracked in order to check reachability. However, here our second main observation comes in handy, namely, that similar to deeply nested recursive procedure calls, also deeply nested recursive spawns can be cut out of a given execution. Consider a situation where a process spawns a second process. The second process in turn spawns a third process with the same initial configuration as the second process and no other processes are spawned by the second process. In this case, the execution of the second process can be replaced by the execution of the third process. This eliminates one ancestor from the execution. This observation can be used to derive a bound on the number of processes that must be tracked in order to decide control-sequence reachability.

Lemma 9. *Assume that $t' = [\![\Pi']\!]\, t_0$ where t' is compatible with the control sequence σ at positions $\eta_1, \dots, \eta_k$ and t' together with Π' is strongly purified*

w.r.t. the control sequence σ. Then there is a subsequence Π'' of Π' such that the following holds:

1. *$t'' = [\![\Pi'']\!]\, t_0$ is still a global configuration which is compatible with the given control sequence σ – but now at positions $\eta'_1, \ldots, \eta'_k$ where the number of distinct non-empty prefixes of $\eta'_1, \ldots, \eta'_k$ is at most $(2k-1) \cdot |Q| \cdot |\Gamma|$.*
2. *The number of active positions in Π'' is bounded by $(2k-1) \cdot |Q| \cdot |\Gamma| + 1$.*

Proof. For the first statement, we purge positions as follows. Assume that $\eta_i = \eta\eta'\eta''$ and the processes at positions η and $\eta\eta'$ are spawned with the same initial configuration $(\epsilon, (q, \emptyset), \gamma)$ and additionally, there is no proper prefix η''' of η' such that $\eta\eta'''$ is the longest common prefix of η_i and some η_j, $i \neq j$. Then $\eta\eta'$ is replaced in all positions η_j where it occurs as a prefix, with η, and the global execution sequence Π is reduced accordingly. This means that all steps $(\eta\chi, r)$ are removed from Π' where χ is a prefix of η', and then all steps $(\eta\eta'\chi, r)$ are replaced with $(\eta\chi, r)$.

This reduction is performed until it is no longer applicable. Let $\eta'_1, \ldots, \eta'_k$ denote the resulting sequence of positions, and Π'' the resulting global execution sequence. Assume for a contradiction that the number of distinct non-empty prefixes of $\eta'_1, \ldots, \eta'_k$ exceeds $(2k-1) \cdot |Q| \cdot |\Gamma|$. As there are only $|Q| \cdot |\Gamma|$ distinct initial configurations of spawned processes, Π'' must create at least $2k$ of the sub-processes represented by these non-empty prefixes with the same initial configuration, say $(\epsilon, (q, \emptyset), \gamma)$. Let $\rho_1, \ldots, \rho_l$, $l \geq 2k$, be (all) the non-empty prefixes of $\eta'_1, \ldots, \eta'_k$ created with this initial configuration $(\epsilon, (q, \emptyset), \gamma)$. Consider the (potentially multi-rooted) tree induced on $\rho_1, \ldots, \rho_l$ by the prefix relation, i.e. ρ_j is a successor of ρ_i in the tree, if ρ_i is a proper prefix of ρ_j but there is no ρ_h that is a proper prefix of ρ_j and a proper suffix of ρ_i. This tree has at most k leafs as any leaf must be a maximal prefix among the $\rho_1, \ldots, \rho_l$ of one of the positions $\eta'_1, \ldots, \eta'_k$. This implies that at most $k-1$ inner nodes can be branching. On the other hand, there are at least k non-maximal prefixes. Hence, at least one of the non-maximal prefixes, say $\eta = \rho_i$, is non-branching, i.e. has just one successor $\rho_j = \eta\eta'$. This implies that the above reduction can be applied with η and η' resulting in a sequence of shorter positions–contradiction. Due to strong purification only ε and non-empty prefixes of the purged positions $\eta'_1, \ldots, \eta'_k$ can be active in Π''. Hence, the second statement follows from the first one.

Proof (Theorem 6). For deciding whether a control sequence σ of length k is reachable from the initial configuration, it suffices by Lemma 8 to consider strongly purified global executions only. By Lemma 9, only global configurations must be considered where the number of active positions is bounded by $(2k-1) \cdot |Q| \cdot |\Gamma|$. Additionally, the construction of Lemma 2 can be extended to DPNs so that only the case must be considered where all processes in the final configuration have an empty pushdown. Then we proceed analogous to the proof of Lemma 4 and derive a bound on the pushdown of each process. The bound now must take the length k of the control sequence into account, since

recursive calls may not be removed in which a process needed for reachability of σ is spawned.

Assume that $t = [\![\Pi]\!]\, t_0$ for an initial configuration t_0 and a global configuration t. Assume that $\sigma = (q'_1, X'_1) \dots (q'_k, X'_k)$ is a control sequence in t at positions $\eta_1 < \ldots < \eta_k$. Assume further that the execution sequence is strongly purified w.r.t. σ. If during Π a call rule $r : (q, \gamma) \xrightarrow{\tau} (q', \gamma_1\gamma_2)$ is called more often than $k \cdot |Q|$ times for the same position η, then there is a state p such that Π can be factored into $\Pi = C_1(r,\eta)U_1(r,\eta)\Pi_1 U_2 C_2$ and the following holds:

- U_1, U_2 do not spawn any processes;
- $\mathsf{proj}_\eta((r,\eta)\Pi_1)$ as well as $\mathsf{proj}_\eta((r,\eta)U_1(r,\eta)\Pi_1 U_2)$ are same-level computations for the subconfiguration $t[\eta]$ resulting in the same control state p.

For $i = 1, 2$, let U'_i be the sequence obtained from U_i by removing all steps of process η. Then the sequence $\Pi' = C_1 U'_1(r,\eta)\Pi_1 U'_2 C_2$ is again a computation sequence for t which is compatible with the control sequence σ at the same positions $\eta_1 < \ldots < \eta_k$.

We conclude that a configuration compatible with σ can be reached by an execution where the depth of each intermediately occurring call-stack is bounded by a polynomial, now in the number of positions in the control sequence and the size of the DPN. Overall, we find that space polynomial in the length of the control-sequence σ and the size of the DPN P is sufficient to verify for P whether σ is reachable by P from the initial configuration $(\epsilon, (p, \emptyset), \gamma)$.

4 Regular Reachability

In this section we introduce *regular control reachability* as a reachability property, that allows to specify properties of configurations of an arbitrary and varying number of processes. Here the word of control states obtained by postorder traversal of a configuration must be contained in a regular language. For that, we define the yield of a configuration $t = (t_1 \dots t_k, ((q, X), w))$ of a DPN as

$$\mathsf{yield}(t) = \mathsf{yield}(t_1) \dots \mathsf{yield}(t_k)(q, X)$$

In the following, we show that regular control reachability is decidable for DPNs with contextual locking.

Theorem 10. *For a DPN P with contextual locking and a regular language L over the alphabet $Q \times 2^{\mathcal{L}}$, it is decidable whether or not a configuration t with $\mathsf{yield}(t) \in L$ is reachable from an initial configuration in P.*

In order to prove Theorem 10, we first show that regular control reachability for a DPN can be reduced to *control-set reachability* of a DPN. In a second step we explicitly reduce each pushdown system to a finite state system, using the same argument for recursive calls as before. A DPN without a pushdown is also called dynamic finite-state network (DFN). W.r.t. control-set reachability, configurations of DFNs can be further abstracted by just abstracting configurations

to vectors which only keep the multiplicities of occurring process-local states. In the following, we are going to make these ideas precise.

First, we reduce regular control reachability to control-set reachability. For that, we define the state set of a configuration $t = (t_1 \dots t_k, ((q, X), w))$ by:

$$\mathsf{states}(t) = \mathsf{states}(t_1) \cup \dots \cup \mathsf{states}(t_k) \cup \{(q, X)\}$$

Lemma 11. *For a DPN P with contextual locking and a regular language L over the alphabet $Q \times 2^{\mathcal{L}}$, there exists a DPN P' with states Q' and contextual locking, and a set $Q'_0 \subseteq Q' \times 2^{\mathcal{L}}$, such that a configuration t with $\mathsf{yield}(t) \in L$ is reachable from an initial configuration in P iff a configuration t' with $\mathsf{states}(t') \subseteq Q'_0$ is reachable from a corresponding initial configuration in P'.*

Proof. Assume that L is given by the finite automaton $\mathcal{A} = (S, Q \times 2^{\mathcal{L}}, \delta, s_0, F)$ where S is the finite set of states of $\mathcal{A}$. We construct a new DPN that encodes the regular reachability into its control states. The yield of a configuration is accepted by the automaton iff there is a run of the automaton that accepts it. The idea is to guess and verify this accepting run during an execution of the DPN. Since the yield of a configuration is constructed from the local process configurations it suffices to guess a partial run for each local configuration and make sure that the partial runs form a run of the automaton. To this end we introduce new control states $\langle s, q, s' \rangle$ where $s, s' \in S$. A control state $\langle s, q, s' \rangle$ signals that s and s' have been guessed as initial and final states for the partial run that recognizes the yield of the subconfiguration generated by this process, and all process it has yet to spawn. As a first step, an initial guess is made. For that, we add transitions $r : (q, \gamma) \xrightarrow{\tau} (\langle s_0, q, s \rangle, \gamma)$ where $s \in F$. We proceed by replacing each non-spawn-transition $r : (q, \gamma) \xrightarrow{e} (q', w')$ with a transition preserving the guess $r' : (\langle s, q, s' \rangle, \gamma) \xrightarrow{e} (\langle s, q', s' \rangle, w')$. In case of a spawn-transition $r : (q, \gamma) \xrightarrow{(q_1, \gamma_1)} (q_2, \gamma_2)$, the guess for the spawned process is initialized by splitting the guess for the parent and distributing it. Therefore, we add transitions $r' : (\langle s, q, s' \rangle, \gamma) \xrightarrow{(\langle s, q_1, s'' \rangle, \gamma_1)} (\langle s'', q_2, s' \rangle, \gamma_2)$. If all processes in an execution of P' reach local configurations $((\langle s_1, q, s_2 \rangle, X), w)$ where $(s_1, (q, X), s_2)$ is a transition of $\mathcal{A}$, then all guesses have been correct, implying that there is an accepting path for the yield of this configuration. If on the other hand an execution of P reaches a configuration whose yield is accepted by $\mathcal{A}$ we can annotate the guesses to obtain an execution of P'. Checking regular reachability thus reduces to checking control set reachability of the set $Q'_0 = \{(\langle s_1, q, s_2 \rangle, X) \mid (s_1, (q, X), s_2) \in \delta\}$.

Remark 12. One can modify the construction from Remark 3 such that it allows to reduce general regular reachability, which also includes the stack content of each process, to regular control reachability. To see this, consider a finite automaton $\mathcal{A}$, now over the input alphabet $(Q \times 2^{\mathcal{L}}) \cup \Gamma$. Then the initial marking of a process can be used to further split the guess from Lemma 11 into parts for the state and the pushdown, i.e. we only add marking rules $r : (\langle s_1, q, s_2 \rangle, \gamma) \xrightarrow{\tau} (\langle \langle s_1, q, s'_2 \rangle, s'_2 \rangle, \langle \gamma, s_2 \rangle)$. The remaining construction proceeds as in Remark 3 using the transitions of $\mathcal{A}$.

In the next step, we reduce control set reachability of a DPN to control set reachability of a DPN without push or pop operations, i.e., a DFN:

Lemma 13. *For a DPN P with contextual locking and a set $Q_0 \subseteq Q \times 2^{\mathcal{L}}$ of control states, there exists a DPN P' with contextual locking, no push or pop operations and a set $Q_0' \subseteq Q' \times 2^{\mathcal{L}}$, such that a configuration t with $\mathsf{states}(t) \subseteq Q_0$ is reachable from an initial configuration in P iff a configuration t' with $\mathsf{states}(t') \subseteq Q_0'$ is reachable from a corresponding initial configuration in P'.*

Proof. First we apply the construction of Lemma 2 to only consider reachability where all pushdowns are empty. Using the same arguments as Lemma 4 and the proof of Theorem 6 we can derive a polynomial bound on the size of pushdown needed to check reachability. Assume that a pushdown is reached during an execution whose size exceeds $|Q|^2 \cdot |\Gamma|$ symbols. This translates to a process with more than $|Q|^2 \cdot |\Gamma|$ nested returning procedure calls. Each nested procedure call can be tagged with the initial control state and topmost pushdown symbol together with the final control state. Since the number of procedure calls exceeds the number of possible tags, there are at least two procedure calls, whose starting and ending situation are the same. Then the outer procedure call can be replaced with the inner call, by removing all steps of the outer procedure call as well as of all processes spawned by it. As before, because of contextual locking and processes starting with an empty set of locks, removing these steps does not impose additional constraints on an execution. Since all remaining processes still reach a state in Q_0, whenever that was the case before the replacement, control-set reachability is preserved if the sizes of all occurring pushdowns are restricted to size at most $|Q|^2 \cdot |\Gamma|$.

Using this result a DFN can be defined with states $Q' = \{(q, w) \mid q \in Q, w \in \Gamma^*, |w| \leq |Q|^2 \cdot |\Gamma|\}$ where the bounded pushdown is encoded into the control state. We introduce an artificial pushdown symbol # and define transitions:

$$((q,w),\#) \xrightarrow{\mathsf{eff}(r)} ((q',w'),\#) \text{ if } (q,w) \overset{r}{\Rightarrow} (q',w') \text{ and } \mathsf{eff}(r) \neq (q_1,\gamma_1)$$
$$((q,w),\#) \xrightarrow{((q_1,\gamma_1),\#)} ((q',w'),\#) \text{ if } (q,w) \overset{r}{\Rightarrow} (q',w') \text{ and } \mathsf{eff}(r) = (q_1,\gamma_1)$$

An initial configuration $(\varepsilon, (q, \emptyset), \gamma)$ is translated into an initial configuration $(\varepsilon, ((q, \gamma), \emptyset), \#)$. Finally, the control set for reachability in the new DFN is set to $Q_0' = \{((q, \varepsilon), X) \in Q' \mid (q, X) \in Q_0\}$. The executions of the new DFN are in one-to-one correspondence to the executions of the original DPN that do not violate the pushdown bound. Thus, we have reduced control-set reachability for a DPN to control-set reachability for a (possibly exponentially larger) DFN.

For control-set reachability of a DFN, the precise ordering of processes within a configuration is irrelevant. Therefore, we now abstract configurations of a DFN to multisets of local process configurations. The proof of Theorem 10 then is based on a monotonicity property of control-set reachability. This monotonicity property states that whenever a control set Q_0 is reachable from a (multi set) configuration v, then this is also the case for any multi sub-set of v.

Proof (Theorem 10). We apply Lemma 11 and Lemma 13 to only consider control set reachability of a set Q_0 for a DFN. For the proof, the ordering of processes within a configuration is irrelevant. Therefore, configurations are abstracted as a vector v mapping pairs (q, X) of states and sets of held locks to the number $v(q, X)$ of processes that are currently in state q and hold the set X of locks. Thus, $v(q, X) > 1$ only if $X = \emptyset$, and for $X \neq \emptyset$, $v(q, X) = 1$ implies $v(q', Y) = 0$ for all $q' \neq q$ and $X \cap Y \neq \emptyset$. Let us call such vectors v *abstract* configurations. Every transition of the DFN P induces a corresponding abstract transition on abstract configurations. Let P' denote the transition system on abstract configurations corresponding to the DFN P. Note that, due to unbounded application of spawn-transitions, the transition system P' is still infinite. The initial configuration t_0 of P corresponds to the abstract configuration $v_0 = \{(q_0, \emptyset) \mapsto 1\}$ where q_0 is the initial state of P. Let V_0 be the set of all abstract configurations such that $v(q, X) = 0$ for all $q \notin Q_0$. Then the DFN P may reach a configuration from t_0 where all occurring states are in Q_0 iff a configuration v is abstractly reachable from v_0 where $v \in V_0$. On configurations of P', we consider the elementwise partial ordering defined by $v \preceq v'$ iff $v(q, X) \leq v'(q, X)$ for all (q, X).

By case distinction, we verify that, if $v \preceq v'$ and w' is reachable in P' from v' in one abstract step, then either $v \preceq w'$ or there is an abstract configuration w which is reachable from v in one step such that $w \preceq w'$.

From this fact, we conclude that whenever a configuration in V_0 is reachable from v' and $v \preceq v'$, then a configuration in V_0 is also reachable from v.

Let W denote the set of abstract configurations reachable from v_0 (w.r.t. P') and $\mathsf{min}(W)$ the set of minimal elements in W w.r.t. the ordering $\preceq$. Then $V_0 \cap W \neq \emptyset$ iff $V_0 \cap \mathsf{min}(W) \neq \emptyset$. Thus, it suffices to determine the set of *minimal* configurations which are reachable from v_0. The set $\mathsf{min}(W)$ can be determined by iteratively accumulating the set of reachable configurations where during every step, only those configurations are maintained which are currently minimal. Since in a set of minimal configurations, vectors are pairwise incomparable, Dickson's Lemma can be applied—implying that the algorithm terminates.

5 Joining of Processes

In [7], DPNs have been considered that are additionally equipped with a join operation. A join can only be executed if all immediate children of a process which have been spawned up to this point, have terminated. We show for DPNs extended with such joins that regular control reachability as considered in the last section, is still decidable.

Theorem 14. *Assume that P is a DPN with joins and contextual locking, and L is a regular language over the alphabet $Q \times 2^{\mathcal{L}}$. Then it is decidable if a configuration t with $\mathsf{yield}(t) \in L$ is reachable from an initial configuration in P.*

Formally, a DPN with joins is a DPN where the set of rules additionally may include dedicated transitions of the form

$$r : (q, \gamma) \xrightarrow{\mathsf{join}} (q', \gamma') \qquad \text{(join step)}$$

The intended semantics is that a process may execute the join-transition only after all processes spawned by the process executing the join-transition, have already been terminated. For that, we assume that termination is signaled by reaching a control state in a set $Q_t \subseteq Q$ from which no further transitions can occur. Thus, the following condition must additionally hold for a step as defined in Section 3:

- if $\mathsf{eff}(r) = \mathsf{join}$ and $h = t_1 \dots t_k$ then $q_i \in Q_t$ for all $i \in \{1, \dots, k\}$, where $t_i = (h_i, (q_i, X_i), w_i)$.

The same arguments as in Section 4 can be applied to show that regular control reachability of a language L for a DPN P with contextual locking and joins can be reduced to control set reachability of a set Q_0 for a DFN P'. This is due to the fact that removing join operations only lessens the constraints on an execution and otherwise removing steps from an execution does not change a thread from terminating to not terminating. As in Lemma 13, we represent the trivial pushdown of a DFN by means of #.

Remark 15. Using the same method as in in Remark 12 Theorem 14 can be extended to regular reachability which includes all pushdowns.

Control-set reachability for DFNs with joins, however, can no longer be naturally reduced to the computation of minimal elements of suitable sets of vectors of natural numbers. Whether or not a join can be executed, does not depend on the multiplicities by which individual process-local states (q, X) are reached but on whether the right subset of processes have terminated. Accordingly, the abstraction of configurations through vectors of numbers is no longer sufficient. Instead, the nesting of processes as given by configurations must be maintained in order to identify the processes to be waited for. In order to apply an analogous argument as in Section 4, a well-quasi-ordering on (suitably abstracted) configurations is required, that preserves reachability. Since configurations are ordered trees, a candidate ordering is the *embedded subtree ordering*. From $t \preceq t'$, however, it not necessarily follows that every sequence of transitions for t' gives rise to a sequence of transitions for t resulting again in a smaller configuration. Here, a configuration is smaller if it can be obtained from the larger configuration by removing a subtree or by removing a node and replacing it with one of its descendants. But removing and replacing a process may cause its parent to wait for termination of a process which does not terminate. A corresponding monotonicity property, though, is crucial in Section 4 for restricting reachability analysis to maintaining sets of minimal elements only.

We observe that a process may be replaced by a descendant, if all processes in the hierarchy inbetween participate in a join. Since a join requires termination of all children and can only be executed by a process before its termination, this ensures that termination of the original process is preceded by termination of the process it is replaced with. Consequently in the shortened execution no join is blocked, since all required processes are still able to terminate.

We now construct an abstraction of a DFN, where configurations are multisets of unordered trees and indicate how the monotonicity property can be

enforced. The idea is to include processes into the tree only when they participate in a join. All others are added as additional roots to the top-level. We show that abstracting a DFN with joins in this way, preserves control-set reachability. A similar argument as in Section 4, then allows us to show decidability. In the following, we present the outlined proof sketch in detail.

For each spawn-transition $r : (q, \#) \xrightarrow{(q_1,\#)} (q_2, \#)$ we introduce a spawn'-transition with the same semantics:

$$r' : (q, \#) \xrightarrow{\langle q_1,\#\rangle} (q_2, \#) \qquad \text{(spawn' step)}$$

Clearly, each configuration t which was reachable w.r.t. the original DFN is also reachable w.r.t. to the DFN with the extra spawn'-transitions by an execution where the following property holds:

- $\mathbf{S}_1$ Every spawn'-transition is eventually followed by a join-transition in the same process;
- $\mathbf{S}_2$ After every spawn-transition, no join-transition occurs in the same process.

Therefore, we may concentrate on control-set reachability of a set Q_0 by means of executions satisfying properties $\mathbf{S}_1$ and $\mathbf{S}_2$. Let us call such executions **S**-executions. By guessing whether a process eventually executes a join-operation or not and maintaining a corresponding bit in the process-local state, we may enforce that the DFN only performs prefixes of S-executions and reaches the set Q_0 of dedicated control states *only* by means of an **S**-execution. Let us call such a DFN an **S**-DFN for control-state reachability of Q_0.

For an **S**-DFN, we now abandon irrelevant nesting of processes and only keep nesting of processes which is required for simulating join-operations. This means that spawn'-transitions add new processes as leaves to the configuration, while spawn-transitions add new processes on toplevel as new roots. For that, we consider finite multi-sets m of unordered finite trees t. Each such tree t is of the form $t = (m', (q, X), \#)$ where q is a state of the **S**-DFN, X is a set of currently held locks and m' is a multiset of trees — each corresponding to a process spawned by a spawn'-transition. We write $\oplus$ for the union of multisets.

For such multisets abstracting configurations of a **S**-DFN, we define the following abstract transitions. A join-transition $r : (q, \#) \xrightarrow{\mathsf{join}} (q', \#)$ is applicable at $t = (m', (q, X), \#)$ within an abstract configuration m if all subtrees $t' \in m'$ are terminated. In this case, it replaces the subtree t within m by the subtree $t' = (m', (q', X), \#)$. Applying the spawn'-transition $r' : (q, \#) \xrightarrow{\langle q_1,\#\rangle} (q_2, \#)$, at $t = (m', (q, X), \#)$ within an abstract configuration m, replaces t with $t' = (m' \oplus \{(\emptyset, (q_1, \emptyset), \#)\}, (q_2, X), \#)$. A spawn-transition $r : (q, \#) \xrightarrow{(q_1,\#)} (q_2, \#)$ applied to a subtree $t = (m', (q, X), \#)$ within an abstract configuration m, replaces t with $t' = (m', (q_2, X), \#)$ and adds the tree $(\emptyset, (q_1, \emptyset), \#)$ to the multiset on the toplevel. The abstract execution steps corresponding to the remaining transitions are defined in a straight forward way. We remark that the notion of an S-execution is also applicable to the abstracted DFNs with joins. Let $P^\sharp$ denote the abstract DFN constructed from a **S**-DFN P in this way. Since every

execution of P is a prefix of an S-execution, the same also holds for abstract executions of $P^\sharp$. Moreover, we obtain that a configuration t where all states are contained in the control set Q_0, can be reached by P from the initial configuration $t_0 = (\varepsilon, (q_0, \emptyset), \#)$ by means of an S-execution iff a configuration m, where all states are from Q_0, can be reached in $P^\sharp$ by means of an abstract S-execution from $m_0 = (\emptyset, (q_0, \emptyset), \#)$. Let V_0 be the set of all multiset configurations m such that all states in m are from Q_0.

On unordered trees t and multisets m, the embedded subtree ordering is the least reflexive and transitive ordering $\preceq$ with the following properties:

- Assume that $t = (m, (q, X), \#)$. Then $t' \in m'$ implies $t' \preceq t$; and also $m' \preceq m$ implies $(m', (q, X), \#) \preceq t$.
- Assume that $m = m_1 \oplus \{t\}$ for some t. Then $m_1 \preceq m$; and $t' \preceq t$ implies $m_1 \oplus \{t'\} \preceq m$.

By Kruskal's Theorem [11], the ordering $\preceq$ on multisets of unordered finite trees is a well-quasi-ordering.

As in the case without join-transitions, we find that if $m \preceq m'$ and w' is reachable in $P^\sharp$ from m' in one step, then either $m \preceq w'$ or there is a multiset configuration w which is reachable from m in a corresponding abstract step such that $w \preceq w'$. From this monotonicity, we conclude that whenever a configuration in V_0 is reachable from m' and $m \preceq m'$, then a configuration in V_0 is also reachable from m.

Proof (Theorem 14). Let W denote the set of abstract multiset configurations reachable from $m_0 = \{(\emptyset, (q_0, \emptyset), \#)$ (w.r.t. $P^\sharp$) and $\mathsf{min}(W)$ the set of minimal elements in W w.r.t. the ordering $\preceq$ on multisets. Then $V_0 \cap W \neq \emptyset$ iff $V_0 \cap \mathsf{min}(W) \neq \emptyset$. Thus, it suffices to determine the set of *minimal* configurations which are abstractly reachable from m_0. The set $\mathsf{min}(W)$ can be determined by iteratively accumulating the set of abstractly reachable configurations where during every step, only those configurations are maintained which are currently minimal. Since in a set of minimal configurations, multisets are pairwise incomparable, we conclude, now no longer by Dickson's lemma, but by Kruskal's Theorem that the algorithm terminates.

6 Conclusion

We have analyzed the complexity of simultaneous reachability for multiple recursive processes running in parallel which may use contextual locking. While this problem has been shown to be PTIME solvable for two processes in [4], we have shown that this problem becomes PSPACE-complete already for $k > 2$ processes where PSPACE is still sufficient if dynamic thread creation is allowed.

The situation seems to be more complicated if reachability of a regular set of configurations is considered. Such regular sets allow to formalize more intricate properties of configurations. We succeeded to prove decidability by means of Dickson's lemma. The precise complexity of this problem, though, remains open.

Interestingly, decidability is preserved even if a join operation is added. Note that fork/join parallelism through parallel procedure calls as considered, e.g., in [15, 18], can be expressed by means of DPNs with join. Accordingly, reachability for this model of concurrency remains decidable if contextual locking is allowed. Also there, however, the precise complexity remains open.

References

1. Bonnet, R., Chadha, R.: Bounded context-switching and reentrant locking. In: FOSSACS. LNCS, vol. 7794, pp. 65–80. Springer (2013)
2. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. Int. J. Found. Comput. Sci. 14(4), 551– (2003)
3. Bouajjani, A., Müller-Olm, M., Touili, T.: Regular symbolic analysis of dynamic networks of pushdown systems. In: CONCUR. LNCS, vol. 3653, pp. 473–487. Springer (2005)
4. Chadha, R., Madhusudan, P., Viswanathan, M.: Reachability under contextual locking. In: TACAS. LNCS, vol. 7214, pp. 437–450. Springer (2012)
5. Esparza, J., Ganty, P.: Complexity of pattern-based verification for multithreaded programs. In: POPL. pp. 499–510. ACM (2011)
6. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theor. Comput. Sci. 256(1-2), 63–92 (2001)
7. Gawlitza, T.M., Lammich, P., Müller-Olm, M., Seidl, H., Wenner, A.: Join-lock-sensitive forward reachability analysis for concurrent programs with dynamic process creation. In: VMCAI. LNCS, vol. 6538, pp. 199–213. Springer (2011)
8. Kahlon, V.: Boundedness vs. unboundedness of lock chains: Characterizing decidability of pairwise cfl-reachability for threads communicating via locks. In: LICS. pp. 27–36. IEEE Computer Society (2009)
9. Kahlon, V.: Reasoning about threads with bounded lock chains. In: CONCUR. LNCS, vol. 6901, pp. 450–465. Springer (2011)
10. Kahlon, V., Ivancic, F., Gupta, A.: Reasoning about threads communicating via locks. In: CAV. LNCS, vol. 3576, pp. 505–518. Springer (2005)
11. Kruskal, J.B.: Well-quasi-ordering, the tree theorem, and vazsonyi's conjecture. Trans. of the American Math. Society 95(2), 210–225 (1960)
12. Lammich, P.: Lock-Sensitive Analysis of Parallel Programs. Ph.D. thesis, WWU Münster (June 2011)
13. Lammich, P., Müller-Olm, M.: Conflict analysis of programs with procedures, dynamic thread creation, and monitors. In: SAS. LNCS, vol. 5079, pp. 205–220. Springer (2008)
14. Lammich, P., Müller-Olm, M., Wenner, A.: Predecessor sets of dynamic pushdown networks with tree-regular constraints. In: CAV. LNCS, vol. 5643, pp. 525–539. Springer (2009)
15. Mayr, R.: Decidability and Complexity of Model Checking Problems for Infinite-State Systems. Ph.D. thesis, TU München (April 1998)
16. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: TACAS. LNCS, vol. 3440, pp. 93–107. Springer (2005)
17. Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. ACM Trans. Program. Lang. Syst. 22(2), 416–430 (2000)
18. Seidl, H., Steffen, B.: Constraint-based inter-procedural analysis of parallel programs. Nord. J. Comput. 7(4), 375–400 (2000)