The Complexity of Bounded Context Switching with Dynamic Thread Creation

Pascal Baumann
Max Planck Institute for Software Systems, Kaiserslautern, Germany
pbaumann@mpi-sws.org

Rupak Majumdar
Max Planck Institute for Software Systems, Kaiserslautern, Germany
rupak@mpi-sws.org

Ramanathan S. Thinniyam
Max Planck Institute for Software Systems, Kaiserslautern, Germany
thinniyam@mpi-sws.org

Georg Zetzsche
Max Planck Institute for Software Systems, Kaiserslautern, Germany
georg@mpi-sws.org

Abstract

Dynamic networks of concurrent pushdown systems (DCPS) are a theoretical model for multi-threaded recursive programs with shared global state and dynamical creation of threads. The (global) state reachability problem for DCPS is undecidable in general, but Atig et al. (2009) showed that it becomes decidable, and is in 2EXPSPACE, when each thread is restricted to a fixed number of context switches. The best known lower bound for the problem is EXPSPACE-hard and this lower bound follows already when each thread is a finite-state machine and runs atomically to completion (i.e., does not switch contexts). In this paper, we close the gap by showing that state reachability is 2EXPSPACE-hard already with only one context switch. Interestingly, state reachability analysis is in EXPSPACE both for pushdown threads without context switches as well as for finite-state threads with arbitrary context switches. Thus, recursive threads together with a single context switch provide an exponential advantage.

Our proof techniques are of independent interest for 2EXPSPACE-hardness results. We introduce *transducer-defined* Petri nets, a succinct representation for Petri nets, and show coverability is 2EXPSPACE-hard for this model. To show 2EXPSPACE-hardness, we present a modified version of Lipton's simulation of counter machines by Petri nets, where the net programs can make explicit recursive procedure calls up to a bounded depth.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Theory of computation → Problems, reductions and completeness

Keywords and phrases Dynamic thread creation, Bounded context switching, Asynchronous Programs, Safety verification, State reachability, Petri nets, Complexity, Succinctness, Counter Programs

Digital Object Identifier 10.4230/LIPIcs.ICALP.2020.111

Category Track B: Automata, Logic, Semantics, and Theory of Programming

Funding This research was funded in part by the Deutsche Forschungsgemeinschaft project 389792660-TRR 248 (see `https://perspicuous-computing.science`) and by the European Research Council under the Grant Agreement 610150 (ERC Synergy Grant ImPACT).

Acknowledgements We would like to thank Ranko Lazić, who made us aware of "counter systems with chained counters" from [3] after submission of this work.

1 Introduction

There is a complexity gap between EXPSPACE and 2EXPSPACE that shows up in several problems in the safety verification of multithreaded programs.

Atig, Bouajjani, and Qadeer [1] study safety verification for *dynamic networks of concurrent pushdown systems* (DCPS), a theoretical model for multithreaded recursive programs with a finite shared global state, where threads can be recursive and can dynamically spawn additional threads. Unrestricted reachability is undecidable in this model. To ensure decidability, like many other works [16, 11, 14, 10], they assume a bound K that restricts each thread to have at most K context switches. For safety verification in this model, formulated as global state reachability, they show a lower bound of EXPSPACE and an upper bound of 2EXPSPACE, "closing the gap" is left open.

Kaiser, Kroening, and Wahl [8] study safety verification of multithreaded *non-recursive* programs with local and global Boolean variables. In this model, an arbitrary number of non-recursive threads execute over shared global state, but each thread can maintain local state in Boolean variables. Although their paper does not provide an explicit complexity bound, a lower bound of EXPSPACE and an upper bound of 2EXPSPACE can be derived from a reduction from Petri net coverability and their algorithm respectively.

Interestingly, when we restrict the models to disallow either context switches (i.e., each thread runs atomically to completion) or local state in the form of the pushdown stack or local variables (but allow arbitrary context switches), safety verification is in EXPSPACE [1, 7].

Thus, the complexity gap asks whether or not the combination of *local state* (maintained in local variables or in the stack) and bounded *context switching* provides additional power to computation. In this paper, we show that indeed it does. In fact, the combination of local state and just *one* context switch is sufficient to achieve 2EXPSPACE lower bounds for these problems. This closes the complexity gap.

We believe the constructions and models that we use along the way are of independent interest. We introduce *transducer-defined* Petri nets (TDPNs), a succinct representation for Petri nets. The places in a TDPN are encoded using words over a fixed alphabet, and the transitions are described by length-preserving transducers. We show that coverability for TDPNs is 2EXPSPACE-complete[1] and give a polynomial-time reduction from coverability for TDPNs to safety verification for DCPS with one context switch.

The idea of the latter reduction is to map a (compressed) place to the stack of a thread and a marking to the set of currently spawned threads. A key obstacle in the simulation is to "transfer" potentially exponential amount of information from before a transition to after it through a polynomial-sized global store. We present a "guess and verify" procedure, using non-determinism and the use of additional threads to verify a stack content letter-by-letter.

In order to show 2EXPSPACE-hardness for TDPNs, we introduce the model of *recursive net programs* (RNPs), which add the power of making possibly recursive procedure calls to the model of net programs (i.e., programs with access to Petri net counters). The addition of recursion enables us to replace the "copy and paste code" idea in Lipton's construction to show EXPSPACE-hardness of Petri net coverability [13] with a more succinct and cleaner program description where the copies are instead represented by different values of the local variables of the procedures. The net effect is to push the requirement for copies into the call stack of the RNP while maintaining a syntax which gives us a RNP which is polynomial in the

[1] After submitting this work, the authors were made aware of "(level 1) counter systems with chained counters" from [3], for which 2EXPSPACE-hardness of state reachability is shown in [3, Theorem 14]. The 2EXPSPACE-hardness of coverability in TDPN could also be deduced from that result.

size of a given counter program. When the stack size is bounded by an exponential function of the size of the program, we get a 2EXPSPACE-lower bound. We show that recursive net programs with exponentially large stacks can be simulated by TDPNs.

Finally, we note that the 2EXPSPACE lower bound holds for DCPS where each stack is bounded by a linear function of the size. Such stacks can be encoded by polynomially many local Boolean variables, giving us a 2EXPSPACE lower bound for the model of Kaiser et al.

In summary, we introduce a number of natural 2EXPSPACE-complete problems and, through a series of reductions, close an exponential gap in the complexity of safety verification for multithreaded recursive programs.

2 Dynamic Networks of Concurrent Pushdown Systems (DCPS)

In this section, we define the model of DCPS and then state our main result. Intuitively, a DCPS consists of a finite state control and several pushdown threads with local configurations, one of them being the active thread. A local configuration contains the number of context switches the thread has already performed, as well as the contents of its local stack. An action of a thread may specify a new thread with initially one symbol on the stack to be spawned as an inactive thread. The active thread can be switched out for one of the inactive threads at any time. When a thread is switched out, its context switch number increases by one. One can view this model as a collection of dynamically created recursive threads (with a call stack each), that communicate using some finite shared memory (the state control).

A *multiset* $\mathbf{m}\colon S \to \mathbb{N}$ over a set S maps each element of S to a natural number. Let $\mathbb{M}[S]$ be the set of all multisets over S. We treat sets as a special case of multisets where each element is mapped onto 0 or 1. We sometimes write $\mathbf{m} = [\![a_1, a_1, a_3]\!]$ for the multiset $\mathbf{m} \in \mathbb{M}[S]$ such that $\mathbf{m}(a_1) = 2$, $\mathbf{m}(a_3) = 1$, and $\mathbf{m}(a) = 0$ for each $a \in S \backslash \{a_1, a_3\}$. The empty multiset is denoted $\emptyset$. The *size* of a multiset $\mathbf{m}$, denoted $|\mathbf{m}|$, is given by $\sum_{a\in S} \mathbf{m}(a)$. Note that this definition applies to sets as well.

Given two multisets $\mathbf{m}, \mathbf{m}' \in \mathbb{M}[S]$ we define $\mathbf{m} \oplus \mathbf{m}' \in \mathbb{M}[S]$ to be a multiset such that for all $a \in S$, we have $(\mathbf{m} \oplus \mathbf{m}')(a) = \mathbf{m}(a) + \mathbf{m}'(a)$. We also define the natural order $\preceq$ on $\mathbb{M}[S]$ as follows: $\mathbf{m} \preceq \mathbf{m}'$ iff there exists $\mathbf{m}^\Delta \in \mathbb{M}[S]$ such that $\mathbf{m} \oplus \mathbf{m}^\Delta = \mathbf{m}'$. We also define $\mathbf{m} \ominus \mathbf{m}'$ for $\mathbf{m}' \preceq \mathbf{m}$ analogously: for all $a \in S$, we have $(\mathbf{m} \ominus \mathbf{m}')(a) = \mathbf{m}(a) - \mathbf{m}'(a)$.

A *Dynamic Network of Concurrent Pushdown Systems (*DCPS*)* $\mathcal{A} = (G, \Gamma, \Delta, g_0, \gamma_0)$ consists of a finite set of *(global) states* G, a finite alphabet of *stack symbols* Γ, an *initial state* $g_0 \in G$, an *initial stack symbol* $\gamma_0 \in \Gamma$, and a finite set of *transition rules* Δ. Elements of Δ have one of the two forms (1) $g|\gamma \hookrightarrow g'|w'$, or (2) $g|\gamma \hookrightarrow g'|w' \triangleright \gamma'$, where $g, g' \in G$, $\gamma, \gamma' \in \Gamma$, $w' \in \Gamma^*$, and $|w'| \leq 2$. Rules of the first kind allow the DCPS to take a single step in one of the pushdown threads while the second additionally spawn a new thread with top of stack symbol γ'. The *size* of $\mathcal{A}$ is defined as $|\mathcal{A}| = |G| + |\Gamma| + |\Delta|$.

The set of *configurations* of $\mathcal{A}$ is $G \times (\Gamma^* \times \mathbb{N}) \times \mathbb{M}[\Gamma^* \times \mathbb{N}]$. Given a configuration $\langle g, (w, i), \mathbf{m}\rangle$, we call g the *(global) state*, (w, i) the *local configuration* of the *active thread*, and $\mathbf{m}$ the multiset of the *local configurations* of the *inactive threads*. The initial configuration of $\mathcal{A}$ is $\langle g_0, (\gamma_0, 0), \emptyset\rangle$. For a configuration c of $\mathcal{A}$, we will sometimes write $c.g$ for the state of c and $c.\mathbf{m}$ for the multiset of threads of c (both active and inactive). The *size* of a configuration $c = \langle g, (w, i), \mathbf{m}\rangle$ is defined as $|c| = |w| + \sum_{(w',j)\in\mathbf{m}} |w'|$.

For $i \in \mathbb{N}$ we define the relation $\Rightarrow_i = \rightarrow_i \cup \mapsto_i$ on configurations of $\mathcal{A}$, where $\rightarrow_i$ and $\mapsto_i$ are defined as follows:

- $\langle g, (\gamma.w, i), \mathbf{m}\rangle \rightarrow_i \langle g', (w'.w, i), \mathbf{m}'\rangle$ for all $w \in \Gamma^*$ iff (1) there is a rule $g|\gamma \hookrightarrow g'|w' \in \Delta$ and $\mathbf{m}' = \mathbf{m}$ or (2) there is a rule $g|\gamma \hookrightarrow g'|w' \triangleright \gamma' \in \Delta$ and $\mathbf{m}' = \mathbf{m} \oplus [\![(\gamma', 0)]\!]$.

- $\langle g, (w,i), \mathbf{m} \oplus [\![(w',j)]\!]\rangle \mapsto_i \langle g, (w',j), \mathbf{m} \oplus [\![(w,i+1)]\!]\rangle$ for all $j \in \mathbb{N}$, $g \in G$, $\mathbf{m} \in \mathbb{M}[\Gamma^* \times \mathbb{N}]$, and $w, w' \in \Gamma^*$.

For $b \in \mathbb{N}$ we define the relation $\Rightarrow_{\leq b} := \bigcup_{i=0}^{b} \Rightarrow_i$. We use $\Rightarrow_i^*$ and $\Rightarrow_{\leq b}^*$ to denote the reflexive, transitive closure of $\Rightarrow_i$ and $\Rightarrow_{\leq b}$, respectively.

Given $K \in \mathbb{N}$, a state g of $\mathcal{A}$ is *K-bounded reachable* iff $\langle g_0, (\gamma_0, 0), \emptyset\rangle \Rightarrow_{\leq K}^* \langle g, (w,i), \mathbf{m}\rangle$ for some $(w,i) \in \Gamma^* \times \{0, \dots, K\}$ and $\mathbf{m} \in \mathbb{M}[\Gamma^* \times \{0, \dots, K+1\}]$.

Intuitively, a local configuration (w,i) describes a pushdown thread with stack content w that has already performed i context switches. The relation $\rightarrow_i$ corresponds to applying the two kinds of transition rules at i context switches. Both of them define pushdown transitions, which the active thread can perform. Type (2) also spawns a new inactive pushdown thread with 0 context switches, whose initial stack content consists of a single specified symbol. For each $i \in \mathbb{N}$, the relation $\mapsto_i$ corresponds to switching out the active thread and raising its number of context switches from i to $i+1$, while also switching in a previously inactive thread. For a fixed K, the *K-bounded state reachability problem* (SRP[K]) for a DCPS is :
Input A DCPS $\mathcal{A}$ and a global state g
Question Is g K-bounded reachable in $\mathcal{A}$?
This corresponds to asking whether the global state g is reachable if each thread can perform at most K context switches.

▶ **Theorem 1** (Main Result)**.** *For each $K \geq 1$, the problem* SRP[K] *is* 2EXPSPACE*-complete.*

The fact that SRP[K] is in 2EXPSPACE for any fixed K follows from the results of Atig et al. [1]. They use a slightly different variant of DCPS. However, it is possible to show a reduction from SRP[K] for our variant to SRP[$K+2$] for theirs.

Our main result is to show 2EXPSPACE-hardness for SRP[1]. One may also adapt the results of Atig et al. to the problem where K is part of the input (encoded in unary), to derive an EXPSPACE lower bound and a 2EXPSPACE upper bound. Our result immediately implies 2EXPSPACE-hardness for this problem as well.

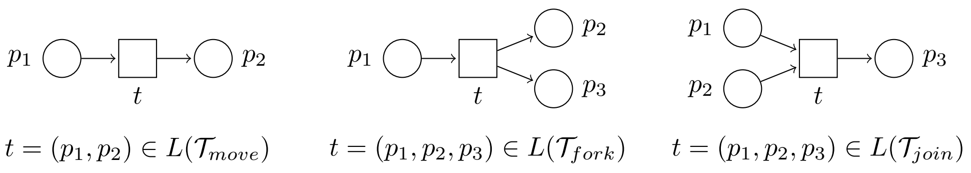
In the remaining sections we prove the lower bound in Theorem 1. In Section 3, we introduce *transducer-defined Petri nets* (TDPN), a succinct representation for Petri nets for which we prove the coverability problem is 2EXPSPACE-complete. Then, we show a reduction from the coverability problem for TDPNs to the SRP[1] problem. In Section 4, we prove hardness for coverability of TDPNs, completing the proof.

3 Transducer Defined Petri Nets (TDPN)

In this section, we prove the lower bound in Theorem 1 by reducing coverability for a succinct representation of Petri nets, namely TDPN, to SRP[1] for DCPS. We first recall some definitions about Petri nets, transducers and problems related to them.

▶ **Definition 2.** *A **Petri net** is a tuple $N = (P, T, F, p_0, p_f)$ where P is a finite set of* places*, T is a finite set of* transitions *with $T \cap P = \varnothing$, $F \subseteq (P \times T) \cup (T \times P)$ is its* flow relation*, and $p_0 \in P$ (resp. $p_f \in P$) its* initial place *(resp.* final place*). A* marking *of N is a multiset $\mathbf{m} \in \mathbb{M}[P]$. For a marking $\mathbf{m}$ and a place p we say that there are $\mathbf{m}(p)$* tokens *on p. Corresponding to the initial (resp. final) place we have the initial marking $\mathbf{m}_0 = [\![p_0]\!]$ (resp. final marking $\mathbf{m}_f = [\![p_f]\!]$). The* size *of N is defined as $|N| = |P| + |T|$.*

A transition $t \in T$ is enabled at a marking $\mathbf{m}$ if $\{p \mid (p,t) \in F\} \preceq \mathbf{m}$. If t is enabled in $\mathbf{m}$, t can be fired, which leads to a marking $\mathbf{m}'$ with $\mathbf{m}' = \mathbf{m} \oplus \{p \mid (t,p) \in F\} \ominus \{p \mid (p,t) \in F\}$. In this case we write $\mathbf{m} \xrightarrow{t} \mathbf{m}'$. A marking $\mathbf{m}$ is coverable *in N if there is a sequence $\mathbf{m}_0 \xrightarrow{t_1} \mathbf{m}_1 \xrightarrow{t_2} \dots \xrightarrow{t_l} \mathbf{m}_l$ such that $\mathbf{m} \preceq \mathbf{m}_l$. We call such a sequence a* run *of N.*



p_1
t
p_2
$t = (p_1, p_2) \in L(\mathcal{T}_{move})$
p_1
t
p_2
p_3
$t = (p_1, p_2, p_3) \in L(\mathcal{T}_{fork})$
p_1
p_2
t
p_3
$t = (p_1, p_2, p_3) \in L(\mathcal{T}_{join})$

Figure 1 The types of transitions defined by the three transducers.

The *coverability problem* for Petri nets is defined as:
Input A Petri net N.
Question Is $\mathbf{m}_f$ coverable in N?

▶ **Definition 3.** *For $n \in \mathbb{N}$, a* (length preserving) ***n-ary transducer*** $\mathcal{T} = (\Sigma, Q, q_0, Q_f, \Delta)$ *consists of an alphabet Σ, a finite set of* states Q, *an* initial state $q_0 \in Q$, *a set of* final states $Q_f \subseteq Q$, *and a* transition relation $\Delta \subseteq Q \times \Sigma^n \times Q$. *For a* transition $(q, a_1, \ldots, a_n, q') \in \Delta$ *we also write $q \xrightarrow{(a_1,\ldots,a_n)} q'$. The size of $\mathcal{T}$ is defined as $|\mathcal{T}| = n \cdot |\Delta|$.*

The language *of $\mathcal{T}$ is the n-ary relation $L(\mathcal{T}) \subseteq (\Sigma^*)^n$ containing precisely those n-tuples $(w_1, \ldots, w_n)$, for which there is a transition sequence $q_0 \xrightarrow{(a_{1,1},\ldots,a_{n,1})} q_1 \xrightarrow{(a_{1,2},\ldots,a_{n,2})} \ldots \xrightarrow{(a_{1,m},\ldots,a_{n,m})} q_m$ with $q_m \in Q_f$ and $w_i = a_{i,1}a_{i,2}\cdots a_{i,m}$ for all $i \in \{1, \ldots, n\}$. Such a transition sequence is called an* accepting run *of $\mathcal{T}$.*

We note that in the more general (i.e. non-length-preserving) definition of a transducer, the transition relation Δ is a subset of $Q \times (\Sigma \cup \varepsilon)^n \times Q$. All transducers we consider in this paper are length-preserving.

▶ **Definition 4.** *A **transducer-defined Petri net** $\mathcal{N} = (w_{init}, w_{final}, \mathcal{T}_{move}, \mathcal{T}_{fork}, \mathcal{T}_{join})$ consists of two words $w_{init}, w_{final} \in \Sigma^l$ for some $l \in \mathbb{N}$, a binary transducer $\mathcal{T}_{move}$ and two ternary transducers $\mathcal{T}_{fork}$ and $\mathcal{T}_{join}$. Additionally, all three transducers share Σ as their alphabet. This defines an* explicit *Petri net $N(\mathcal{N}) = (P, T, F, p_0, p_f)$:*

- *$P = \Sigma^l$.*
- *T is the disjoint union of T_{move}, T_{join} and T_{fork}[2] where*
  - *$T_{move} = \{(w, w') \in \Sigma^l \times \Sigma^l \mid (w, w') \in L(\mathcal{T}_{move})\}$,*
  - *$T_{fork} = \{(w, w', w'') \in \Sigma^l \times \Sigma^l \times \Sigma^l \mid (w, w', w'') \in L(\mathcal{T}_{fork})\}$, and*
  - *$T_{join} = \{(w, w', w'') \in \Sigma^l \times \Sigma^l \times \Sigma^l \mid (w, w', w'') \in L(\mathcal{T}_{join})\}$.*
- *$p_0 = w_{init}$ and $p_f = w_{final}$.*
- *$\forall t \in T$:*
  - *If $t = (p_1, p_2) \in T_{move}$ then $(p_1, t), (t, p_2) \in F$.*
  - *If $t = (p_1, p_2, p_3) \in T_{fork}$ then $(p_1, t), (t, p_2), (t, p_3) \in F$.*
  - *If $t = (p_1, p_2, p_3) \in T_{join}$ then $(p_1, t), (p_2, t), (t, p_3) \in F$.*

An accepting run of one of the transducers, which corresponds to a single transition of N, is called a transducer-move. *The size of $\mathcal{N}$ is defined as $|\mathcal{N}| = l + |\mathcal{T}_{move}| + |\mathcal{T}_{fork}| + |\mathcal{T}_{join}|$.*

A Petri net defined by transducers in this way can only contain three different types of transitions, each type corresponding to one of the three transducers. These transition types are depicted in Figure 1. The *coverability problem* for TDPN is given by:
Input A TDPN $\mathcal{N}$.
Question Is $\mathbf{m}_f = [\![w_{final}]\!]$ coverable in the corresponding explicit Petri net $N(\mathcal{N})$?

[2] Note that a tuple $(w, w', w'') \in T_{join}$ is different from the same tuple in T_{fork}. In the interest of readability, we have chosen not to introduce a 4th coordinate to distinguish the two.

Observe that the exlicit Petri net $N(\mathcal{N})$ has $|\Sigma|^l$ places, which is exponential in the size of $\mathcal{N}$. This means that TDPN are exponentially succinct representations of Petri nets.

It is a common theme in complexity theory to consider succinct versions of decision problems [12, 6, 15]. The resulting complexity is usually one exponent higher than the original version. In fact, certain types of hardness proofs can be lifted generically [15] (but such a simple argument does not seem to apply in our case). The hardness proof in the following is deferred to Section 4.

▶ **Theorem 5.** *The coverability problem for* TDPN *is* 2EXPSPACE*-complete.*

Traditionally, succinct versions of graphs and automata feature a compression using circuits [6, 15] or formulas [12]. One could also compress Petri nets by using circuits to accept binary encodings of elements (p, t) or (t, p) of the flow relation. It is relatively easy to reduce coverability for TDPN to this model by encoding transitions t as the pair or triple of places that they correspond to, yielding 2EXPSPACE-hardness. We consider transducers because they make the reduction to DCPS more natural. 2EXPSPACE-membership for any such representation follows by first unravelling the Petri net and then checking coverability [17].

We now show that coverability for TDPN can be reduced in polynomial time to SRP[1] for DCPS. The goal of the reduction is, given a TDPN $\mathcal{N} = (w_{init}, w_{final}, \mathcal{T}_{move}, \mathcal{T}_{fork}, \mathcal{T}_{join})$, to produce a DCPS $\mathcal{A}(\mathcal{N})$ with a global state *halt* such that w_{final} is coverable in $\mathcal{N}$ iff *halt* is 1-bounded reachable in $\mathcal{A}(\mathcal{N})$. We outline the main ideas and informally explain the solution to some technical issues that arise.

Representation of Markings. The main idea behind the simulation of a TDPN $\mathcal{N}$ by a DCPS $\mathcal{A}(\mathcal{N})$ is that a token on a place w of $\mathcal{N}$ is represented by a thread with stack content w. Extending this idea, a marking is represented by a multiset of threads, one for each token.

Initialization. The initial marking of $\mathcal{N}$ is $[\![w_{init}]\!]$ and $\mathcal{A}(\mathcal{N})$ starts by going into a special state where it always fills its stack with w_{init} and then moving to a global state *main*. We need $O(l)$ states in the global memory for the initialization.

Simulation of one Transducer-move. In the sequel, we explain the simulation of a single transducer-move from $\mathcal{T}_{move}$; the changes required to be made in the case of $\mathcal{T}_{join}$ and $\mathcal{T}_{fork}$ are explained at the end. Remembering the choice of transducer incurs a multiplicative cost of 3 in the global memory. The transducer-move requires us to do two things: Read the stack contents of a particular *input* thread which corresponds to a place w from which a token is removed; after which we need to create an *output* thread which corresponds to a place w' to which a token is added. This results in the following issue regarding input threads:

Issue 1: How can an input thread communicate its stack content w which comes from an exponentially large space of possibilities (since this space is Σ^l) given the requirement for the global state space to be polynomial in $|\mathcal{N}|$?

Solution 1: We pop the contents of the thread while simultaneously spawning *bit-threads*, each of which contains one letter of w along with the index $i \in \{1, \ldots, l\}$ of the letter and the information that w is a place from which a token is being removed; all of which is coded into a single *bit-symbol*.

Note that we have two types of threads: bit-threads and token-threads (i.e., those whose stack contents encode a token's position). Moreover, these two types of threads have disjoint sets of stack symbols: bit-symbols and *token-symbols*. The idea used to solve Issue 1 and read the stack contents, cannot be used in reverse to create an output token-thread since it is not possible to populate a stack with information from bit-threads.

Issue 2: How do we ensure the creation of appropriate output threads?

Solution 2: We implement a "guess-and-verify" procedure whereby we first guess the contents of an output token-thread while simultaneously producing bit-threads corresponding to w'; this is followed by a verification of the transition by comparing bit-threads produced corresponding to w and w', in a bit-by-bit fashion.

In particular, our simulation of a single transducer-move corresponds to a loop on the global state *main* which is broken up into three stages: Read, guess and verify. The implementation of this loop ensures that a configuration c of $\mathcal{A}(\mathcal{N})$ where $c.g = \mathit{main}$ has a multiset $c.\mathbf{m}$ of threads faithfully representing a marking $\tilde{\mathbf{m}}$ of $\mathcal{N}$ in that $c.\mathbf{m}$ contains exactly $\tilde{\mathbf{m}}(w'')$ token-threads with stack content w'' for each place w'' of $\mathcal{N}$ and no other threads.

We note that the discussion so far shows how the run of a $\mathcal{N}$ can be simulated when the schedule switches contexts at appropriate times. We must also ensure that new behaviors cannot arise due to context switches at arbitrary other points. We accomplish this by using global *locks* that ensure unwanted context switches get stuck.

Issue 3: How do we control the effect of arbitrary context switches?

Solution 3: The global state is partitioned in such a way as to only enable operations on bit-symbols while in some states and token-symbols in others. We ensure that for every bit-symbol γ, there is at most one thread with top of stack γ at any given time. Thus with the help of global control, we make sure unwanted context switches to bit-threads get the system stuck. The problem reduces to avoiding unwanted context switches between token-threads.

We use a *locking mechanism.* We add an extra $\top$ symbol at the top of every token-thread when it is first created. A read-stage always begins in a special state used for unlocking a thread (i.e. removing $\top$). While reading a particular thread, the global state disallows any transition on $\top$ or bit-symbols. Since all inactive token-threads have $\top$ as the top of stack symbol, this implies that the system cannot proceed until it switches back to the unlocked token-thread. Similarly, during the guess-stage where we are creating a new token-thread, transitions are disallowed on $\top$ and bit-symbols. The verify-stage only operates on bit-threads and switching to a token-thread is similarly pointless.

We now describe the three stages. Recall that the global state keeps the information that the current step is a transducer-move from $\mathcal{T}_{move}$.

Read-stage. We non-deterministically switch to a token-thread t_0 containing w as stack content, which we need to read. As explained earlier, we produce bit-threads decorated appropriately and at the end of this stage, we have popped all of t_0 and created l bit-threads; t_0 ceases to exist. The number of global states required in the stage is $O(l)$.

Guess-stage. Next, we create a new token-thread with w' as its stack contents by non-deterministic guessing, simultaneously spawning bit-threads for each letter of w'. At the end of this stage l more bit-threads have been added to the task buffer (for a total of $2l$ bit-threads) along with a token-thread containing w'. As in the read-stage, the number of global states used in this stage is $O(l)$.

Verify-stage. We guess a sequence of transitions $\delta_1 \dots \delta_l$ of $\mathcal{T}_{move}$ on-the-fly; we guess δ_i which must be of the form $q_{i-1} \xrightarrow{(w_i, w'_i)} q_i$ where w_i (resp. w'_i) the i^{th} letter of w (resp. w'). We verify our guess by comparing each δ_i with the corresponding bit-threads b_i, b'_i with index i produced in the read-stage from w, w' respectively, before moving on to δ_{i+1}. During the verification, the bit-threads are *killed.* We enforce the condition that the target state of δ_i matches the source state of δ_{i+1}.

Claim. Killing a bit-thread t' with a single stack symbol γ' can be simulated by a DCPS. Consider the following sequence of operations starting from global state g with an active thread t which contains only one symbol γ on the stack:

1. Spawn a thread t'' with a special symbol γ_{spawn} and move to a special kill-state *kill* which contains information regarding the state g prior to the kill operation and stack symbols of t and t'.
2. Switch to a thread with symbol γ' and pop its contents while moving to a special state *return* which is forwarded the information contained in *kill*.
3. Switch to the thread with γ_{spawn} as top of stack and replace it with γ and at the same time go to global state g.

This concludes our proof sketch of the claim. Adding a kill operation to a DCPS only incurs a polynomial increase in the size of the DCPS.

In our setting, the net result of the sequence of operations simulating a kill-move is to remove the two bit-threads b_i, b'_i from the multiset of threads without changing the global state or the top of stack symbol γ. The special states *kill* (resp. *return*) ensure that if one switches to a thread whose top of stack is different from γ' in Step 2 (resp. γ_{spawn} in Step 3), no transition can be made. We return to our discussion regarding the sequence of transitions δ_i.

Since this process of checking the transducer-move occurs bit-by-bit, we require $O(l|\mathcal{T}_{move}|)$ many global states in this stage. At the end of the verification process, $\mathcal{A}(\mathcal{N})$ is once again in state *main* and the new multiset is the result of the addition of a w' thread and removal of the w thread from the old multiset of threads. We can now simulate the next transducer-move.

Checking for Coverability. At any point when $\mathcal{A}(\mathcal{N})$ is in the state *main*, it makes a non-deterministic choice between simulating the next transducer-move or checking for coverability. In the latter case, it goes into a special *check* state where the active thread is compared letter by letter with w_{final} in a process similar to initialization. At the end of the checking process, $\mathcal{A}(\mathcal{N})$ reaches the state *halt*. If the check fails at any intermediate point, $\mathcal{A}(\mathcal{N})$ terminates without reaching the *halt* state. We require a further $O(l)$ states for checking coverability.

Fork and Join. We have shown above how a single transducer-move is simulated assuming that it is a transducer-move from $\mathcal{T}_{move}$. In general, the transducer-move could be from $\mathcal{T}_{join}$ or $\mathcal{T}_{fork}$ as well. In these two cases, we have triples of the form (w, w', w'') accepted by the transducer. However, in the former, we read w, w' and guess w'' while in the latter, we read w and guess w', w''. In the case of $\mathcal{T}_{join}$, once we have read w, we non-deterministically switch to a thread containing w' as its contents. Whenever the threads picked during the read-stage and the threads created during the guess-stage do not agree with the guessed transitions of the transducer-move, we encounter a problem during the verify-stage and $\mathcal{A}(\mathcal{N})$ terminates without reaching the *halt* state.

Context Switches. Every thread (other than the initial one for w_{init}) is created during the guess-stage and then switched out once. The next time it is switched in, it is read and ceases to exist. This implies that there exists a run of $\mathcal{A}(\mathcal{N})$ simulating a run of $\mathcal{N}$ where every thread undergoes at most one context switch. Conversely, we show that a run of $\mathcal{A}(\mathcal{N})$ reaching *halt* where every thread is bounded by at most 1 context switch implies the existence of a run in $\mathcal{N}$ which covers the final marking as desired.

This concludes our overview of the construction of $\mathcal{A}(\mathcal{N})$ and completes the reduction of coverability for TDPN to SRP[1] for DCPS. The global memory is polynomial in the size of $\mathcal{N}$. Similarly, the stack alphabet is expanded to include $O(l \cdot |\Sigma|)$ bit symbols, hence the alphabet of $\mathcal{A}(\mathcal{N})$ is polynomial as well. In summary, $\mathcal{A}(\mathcal{N})$ can be produced in time polynomial in the size of the input.

▶ Remark 6. Our lower bound holds already for DCPS where the stack of each thread is bounded by a linear function of the size of the DCPS. Thus, as a corollary, we get 2EXPSPACE-hardness for a related model in which each thread is a *Boolean program*, i.e., where each thread has its stack bounded by a constant but has a polynomial number (in the size of $|G| + |\Gamma| + |\Delta|$) of local Boolean variables. This closes the gap from [8] as well as other similar models studied in the literature [2, 9, 4].

4 Recursive Net Programs (RNP)

We prove Theorem 5 by adapting the Lipton construction [13], as it is explained in [5], to our succinct representation of Petri nets. Our construction requires two steps. First we reduce termination for bounded counter programs to termination for *Petri net* programs which do not allow zero tests. Second, we reduce termination of net programs with to coverability for TDPN.

For the first step, we have to show how we can simulate the operation of a bounded counter program with one without zero tests. In the Lipton construction, this is achieved by constructing a gadget that performs zero tests for counters bounded by some bound B. These gadgets are obtained by transforming a gadget for bound B into a gadget for B^2. Starting with $B = 2$ and applying this transformation n times leads to a gadget for $B = 2^{2^n}$. One then has to argue that the resulting net program still has linear size in the parameter n. For a 2EXPSPACE lower bound, one would need to simulate a program where the bound is triply exponential in n. A naive implementation of the gadget would then lead to a program with triply exponential counter values, but exponential program size in n.

In order to argue later that the resulting program can be encoded in a small TDPN, we will present the Lipton construction in a different way. Instead of growing the program with every gadget transformation, we implement the gadgets recursively using a stack. We call these programs *recursive net programs* (RNP). This way, when we instantiate the model for a triply exponential bound on the counters (to get 2EXPSPACE-hardness instead of EXPSPACE-hardness), the resulting programs still have polynomial size control flow. Note that at run time, such programs can have an exponentially deep stack; however, this very large stack does not form part of the program description. We shall show that RNP have a natural encoding as TDPN.

For the second step, we reduce termination for RNP to coverability for TDPN. To this end, we borrow some techniques from the original construction to translate an RNP into an exponential sized Petri net. We then assign binary addresses to its places and construct transducers for those pairs and triples that correspond to transitions. This results in a TDPN of polynomial size. Finally, we argue that we do not need the whole exponential sized Petri net to reason about the transducers, and that just a polynomial size part suffices. This then gives us a polynomial time procedure.

4.1 From Bounded Counter Programs to RNP

Bounded Counter Programs. A *counter program* is a finite sequence of *labelled commands* separated by semicolons. Let l, l_1, l_2 be *labels* and x be a *variable* (also called a *counter*). The labelled commands have one of the following five forms:

(1) l : **inc** x; // increment
(2) l : **dec** x; // decrement
(3) l : **halt**
(4) l : **goto** l_1; // unconditional jump
(5) l : **if** $x = 0$ **then goto** l_1 **else goto** l_2; // conditional jump

Variables can hold values over the natural numbers, labels have to be pairwise distinct, but can otherwise come from some arbitrary set. For convenience, we require each program to contain exactly one **halt** command at the very end. The *size* $|C|$ of a counter program C is the number of its labelled commands.

During execution, all variables start with initial value 0. The semantics of programs follows from the syntax, except for the case of decrementing a variable whose value is already 0. In this case, the program aborts, which is different from proper termination, i.e., the execution of the **halt** command. It is easy to see that each counter program has only one execution, meaning it is deterministic. This execution is *k-bounded* if none of the variables ever reaches a value greater than k during it.

Let $\exp^{m+1}(x) := \exp(\exp^m(x))$ and $\exp^1(x) = \exp(x) := 2^x$. The N-fold exponentially bounded *halting problem* (also called *termination*) for counter programs ($\mathsf{HP}[N]$) is given by:
Input A unary number $n \in \mathbb{N}$ and a counter program C.
Question Does C have an $\exp^N(n)$-bounded execution that reaches the **halt** command?
We make use of the following well-known result regarding this problem:

▶ **Theorem 7.** *For each $N > 0$, the problem $\mathsf{HP}[N+1]$ is N-EXPSPACE-complete.*

The proof for arbitrary N matches the proof for $N = 1$, which the Lipton construction used.

Recursive Net Programs. The definition of *recursive net programs* (RNP) also involves sequences of labelled commands separated by semicolons. Let l, l_1, l_2 be labels, x be a variable, and `proc` be a procedure name. Then the labelled commands can still have one of the previous forms (1) to (4). However, form (5) changes from a conditional to a nondeterministic jump, and there are two new forms for procedure calls:

(1) l : **inc** x; // increment
(2) l : **dec** x; // decrement
(3) l : **halt**
(4) l : **goto** l_1; // unconditional jump
(5) l : **goto** l_1 **or goto** l_2; // nondeterministic jump
(6) l : **call** `proc`; // procedure call
(7) l : **return**; // end of procedure

In addition to labelled commands, these programs consist of a finite set PROC of procedure names and also a *maximum recursion depth* $k \in \mathbb{N}$. Furthermore, they not only contain one sequence of labelled commands to serve as the main program, but also include two additional sequences of labelled commands for each procedure name `proc` $\in$ PROC. The second sequence for each `proc` is not allowed to contain any **call** commands and serves as a sort of "base case" only to be called at the maximum recursion depth. Each label has to be unique among all sequences and each jump is only allowed to target labels of the sequence it belongs to. Each RNP contains exactly one **halt** command at the end of the main program.

For $\texttt{proc} \in \mathsf{PROC}$ let $\#c(\texttt{proc})$ be the number of commands in both of its sequences added together and let $\#c(\texttt{main})$ be the number of commands in the main program. Then the *size* of an RNP R is defined as $|R| = \lceil \log k \rceil + \#c(\texttt{main}) + \sum_{\texttt{proc} \in \mathsf{PROC}} \#c(\texttt{proc})$.

The semantics here is quite different compared to counter programs: If the command "l : **call** $\texttt{proc}$" is executed, the label l gets pushed onto the call stack. Then if the stack contains less than k labels, the first command sequence pertaining to $\texttt{proc}$, which we now call $\texttt{proc}_{<\max}$, is executed. If the stack already contains k labels, the second command sequence, $\texttt{proc}_{=\max}$, is executed instead. Since $\texttt{proc}_{=\max}$ cannot call any procedures by definition, the call stack's height (i.e. the *recursion depth*) is bounded by k. On a return command, the last label gets popped from the stack and we continue the execution at the label occurring right after the popped one.

How increments and decrements are executed depends on the current recursion depth d as well. For each variable x appearing in a command, $k + 1$ copies x_0 to x_k are maintained during execution. The commands **inc** x resp. **dec** x are then interpreted as increments resp. decrements on x_d (and not x or any other copy). As before, all these copies start with value 0 and decrements fail at value 0, which is different from proper termination.

Instead of a conditional jump, we now have a nondeterministic one, that allows the program execution to continue at either label. Regarding termination we thus only require there to be at least one execution that reaches the **halt** command. This gives us the following halting problem for RNP:

Input An RNP R

Question Is there an execution of R that reaches the **halt** command?

We now adapt the Lipton construction to recursive net programs. We start with a $\exp^2(n)$-bounded counter program C with a set of counters X and construct an RNP $R(C)$ with maximum recursion depth $n + 1$ that terminates iff C terminates. The number of commands in $R(C)$ will be linear in $|C|$.

Auxiliary Variables. The construction of $R(C)$ involves simulating the zero test. To this end, we introduce for each counter $x \in X$ a complementary counter $\bar{x}$ and ensure that the invariant $x_0 + \bar{x}_0 = \exp^2(n)$ always holds. We can then simulate a zero test on x by checking that $\bar{x}$ can be decremented $\exp^2(n)$ times. This requires us to implement a decrement by $\exp^2(n)$ in linearly many commands and also a similar increment to reach a value of $\exp^2(n)$ for $\bar{x}$ from its initial value 0 at the start of the program. Furthermore, we need helper variables s, $\bar{s}$, y, $\bar{y}$, z, and $\bar{z}$. We also sometimes need to increment or decrement the $(d+1)$th copy of one of these six variables at recursion level d. As an example, for incrementing s_{d+1} in this way, we define the procedure $\texttt{s\_inc}$:

$$\texttt{s\_inc}_{<\max}: \ \textbf{inc}\ s; \textbf{return} \qquad \texttt{s\_inc}_{=\max}: \ \textbf{inc}\ s; \textbf{return}$$

The analogous procedures for $\bar{s}$, y, $\bar{y}$, z, and $\bar{z}$ are defined similarly.

Program Structure. The program $R(C)$ consists of two parts: The initial part $R_{init}(C)$, which initializes all the complementary counters as mentioned above, followed by $R_{sim}(C)$, the part that simulates C. We construct $R_{sim}(C)$ from C by replacing some of its commands. Increments of the form **inc** x are replaced by **dec** $\bar{x}$;**inc** x, decrements **dec** x are replaced by **dec** x;**inc** $\bar{x}$. Unconditional jumps and the **halt** command stay the same. Each conditional jump (form (5) for counter programs) is replaced by

$$\begin{aligned} l &: \mathrm{Test}(x, l_{\text{continue}}, l_2); \\ l_{\text{continue}} &: \mathrm{Test}(\bar{x}, l_1, l_2) \end{aligned}$$

```
Test($x, l_{\mathrm{zero}}, l_{\mathrm{nonzero}}$) :

               goto $l_{\mathrm{nztest}}$ or goto $l_{\mathrm{loop}}$;
  $l_{\mathrm{nztest}}$ : dec $x$; inc $x$; goto $l_{\mathrm{nonzero}}$;
    $l_{\mathrm{loop}}$ : dec $\bar{x}$; inc $x$; call s̄_dec; call s_inc;
               goto $l_{\mathrm{exit}}$ or goto $l_{\mathrm{loop}}$;
    $l_{\mathrm{exit}}$ : call dec; goto $l_{\mathrm{zero}}$

Test$_{+1}$($v, l_{\mathrm{zero}}, l_{\mathrm{nonzero}}$) :

               goto $l_{\mathrm{nztest}}$ or goto $l_{\mathrm{loop}}$;
  $l_{\mathrm{nztest}}$ : call v_dec; call v_inc; goto $l_{\mathrm{nonzero}}$;
    $l_{\mathrm{loop}}$ : call v̄_dec; call v_inc;
               call s̄_dec; call s_inc;
               goto $l_{\mathrm{exit}}$ or goto $l_{\mathrm{loop}}$;
    $l_{\mathrm{exit}}$ : call dec; goto $l_{\mathrm{zero}}$
```

```
dec$_{<\mathrm{max}}$ :

   $l_{\mathrm{outer}}$ : call y_dec; call ȳ_inc;
   $l_{\mathrm{inner}}$ : call z_dec; call z̄_inc;
             dec $s$; inc $\bar{s}$;
             Test$_{+1}$($z, l_{\mathrm{next}}, l_{\mathrm{inner}}$);
    $l_{\mathrm{next}}$ : Test$_{+1}$($y, l_{\mathrm{exit}}, l_{\mathrm{outer}}$);
    $l_{\mathrm{exit}}$ : return

dec$_{=\mathrm{max}}$ :

             dec $s$; inc $\bar{s}$; dec $s$; inc $\bar{s}$;
             return
```

Figure 2 Definitions of the macros Test and Test$_{+1}$ as well as the procedure dec. Regarding the second macro we require $v \in \{y, z\}$.

where Test$(x, l_{zero}, l_{nonzero})$ is what we call a *macro*. We use it as syntactic sugar to be replaced by its specification for the actual construction of $R(C)$. This is in contrast to procedures, which refer to specific parts of the program that can be called to increase the recursion depth.

Test Macros and Decrement Procedure. The macro Test is specified in the left part of Figure 2. It involves a call to the procedure dec, which is defined in the right part of the same figure. Below Test we have also specified the variant Test$_{+1}$, which is used in dec. The main difference is that Test$_{+1}$ can only be invoked on variables y or z and acts on their $(d+1)$th copy at recursion depth d.

Semantically, dec at recursion depth d decrements s_d by $\exp^2(n+1-d)$ (and increments $\bar{s}_d$ by the same amount). Both variants of Test simulate a conditional jump and have the side effect of switching the values x_d and $\bar{x}_d$ if the tested variable x_d was 0. Because of this, every conditional jump of C gets replaced by two instances of the Test macro, where the second one reverses the potential side effect.

The decrements of procedure dec are performed via two nested loops that each run $\exp^2(n-d)$-times. Each of these loops uses a helper variable y_{d+1} or z_{d+1} that has to be tested for zero at the end, using the Test$_{+1}$ macro. This involves transferring the helper variable's value to s_{d+1} and then calling dec at the next recursion depth. Essentially, any decrement by $\exp^2(j)$ for some j is implemented using $\exp^2(j-1)$ many decrements by $\exp^2(j-1)$ via the nested loops. This iterative squaring of the value by which we decrement continues down to the base case of $\exp^2(0) = 2$.

Semantics. Our construction is semantically very similar to the Lipton construction, barring two main differences: Firstly, instead of having $n+1$ different procedure definitions of dec (one per level d), we only need two because of recursion. The case for the Test macros is similar, as is the case of the helper variables s, y, z and their complements. Secondly, our variable copies start with index 0 counting upwards, whereas in the Lipton construction the variables start with index n and count downwards. This means that for some index d we have the invariant $s_d + \bar{s}_d = \exp^2(n+1-d)$ in our construction, where it is $s_d + \bar{s}_d = \exp^2(d)$ for Lipton. While the invariant of the Lipton construction is simpler, ours allows us to define the recursion depth starting at 0 and going upwards, which seemed more natural for recursion.

Let us give a more precise analysis regarding the effect of the Test macros and `dec` procedure. During the execution of `dec` at recursion depth d, we begin with $s_d = \exp^2(n+1-d)$, $y_{d+1} = z_{d+1} = \exp^2(n-d)$, and $\bar{s}_d = \bar{y}_{d+1} = \bar{z}_{d+1} = 0$. The invariants $s_d + \bar{s}_d = \exp^2(n+1-d)$, $y_{d+1} + \bar{y}_{d+1} = \exp^2(n-d)$, and $z_{d+1} + \bar{z}_{d+1} = \exp^2(n-d)$ are upheld throughout. At the end we have $s_d = \bar{y}_{d+1} = \bar{z}_{d+1} = 0$, $y_{d+1} = z_{d+1} = \exp^2(n-d)$, and $\bar{s}_d = \exp^2(n+1-d)$, meaning the decrements were performed correctly and all helper variables retain their initial values. The situation is quite similar for Test and Test_{+1}, if the variable to be tested was initially 0. In the non-zero case, the tested variable is just decremented and incremented once, whereas no other variables are touched. All executions that differ from the described behavior are guaranteed to get stuck.

Correctness of these semantics can be proven by induction on the recursion depth. It requires the assumptions $x_0 + \bar{x}_0 = \exp^2(n)$, $v_d + \bar{v}_d = \exp^2(n+1-d)$, and $\bar{v}_d = 0$ for all $x \in X$, $v \in \{\bar{s}, y, z\}$, and $d > 0$.

Initialization. We now have to construct $R_{init}(C)$ in such a way, that it performs all the necessary increments for these assumptions to hold at the start of $R_{sim}(C)$. Since this is again achieved using iterative squaring, we omit the precise construction. It involves calling a procedure `inc` to perform the increments on copies of variables at lower recursion depths, while x_0 is incremented in the main program for each $x \in X$.

Size Analysis. To give a brief size analysis of $R(C)$, PROC contains 14 procedure names, whose corresponding definitions have constant size. For each command in C, $R_{sim}(C)$ contains constantly many commands, and $R_{init}(C)$ has linearly many commands in the size of the variable set X. Since wlog. each variable of C is involved in at least one of its commands, the amount of commands in $R(C)$ is linear in $|C|$. Here, for doubly exponential counter values, we would not even need n to be given in unary since only $\lceil \log(n+1) \rceil$ factors into the size of $R(C)$.

Handling Triply Exponential Counter Values. The exact same construction with a maximum recursion depth of $2^n + 1$ can be used to simulate a counter program with counters bounded by $\exp^3(n)$: Starting with 2 and squaring n-times yields $\exp^2(n)$, therefore squaring 2^n times instead yields $\exp^3(n)$. The correctness follows from the same inductive proofs as before. For this changed maximum recursion depth, configurations contain exponentially in n many counter values and also maintain a call stack of size up to 2^n. However, since the maximum recursion depth can be encoded in binary, its size is still polynomial in the unary encoding of n. Thus, the halting problem for recursive net programs is 2EXPSPACE-hard.

4.2 From RNP to TDPN

Figure 3 and Figure 4 show how the commands of recursive net programs can be simulated by Petri net transitions. This is again done in similar fashion to Esparza's description [5] of the Lipton construction [13]. As we can see, this involves only the three types of transitions defined by our transducers.

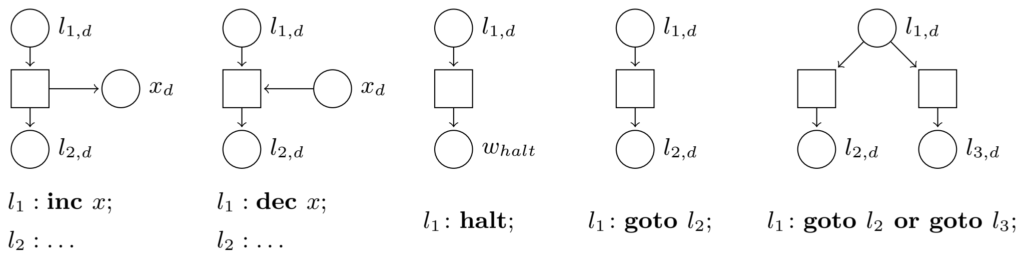


Figure 3 Petri net transitions for five of the seven command types found in recursive net programs. Here, $d \in \{0, \dots, k\}$, where k is the maximum recursion depth.

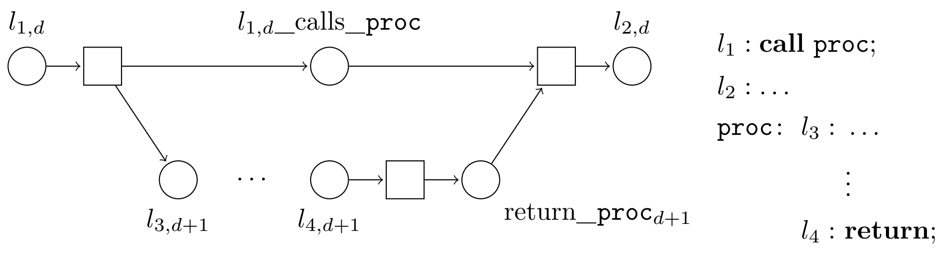


Figure 4 Petri net transitions for procedure calls found in recursive net programs. Here, $d \in \{0, \dots, k-1\}$, where k is the maximum recursion depth.

Let us give more detail regarding the Petri net construction: Given an RNP R with maximum recursion depth k we construct a transducer-defined Petri net $\mathcal{N} = (w_{init}, w_{final}, \mathcal{T}_{move}, \mathcal{T}_{fork}, \mathcal{T}_{join})$, which defines the Petri net $N(\mathcal{N}) = (P, T, F, p_0, p_f)$, such that $[\![p_f]\!]$ is coverable in $N(\mathcal{N})$ iff there is a terminating execution of R. We begin by arguing about the shape of $N(\mathcal{N})$ and then construct our transducers afterwards.

The idea is for $N(\mathcal{N})$ to start with one place per variable and one place per label, as well as one auxiliary place for each **call** command and each proc $\in$ PROC, which can be seen in Figure 4. Additionally, there is also a single auxiliary place w_{halt} for the **halt** command. Let the number of all these places be h. Then each such place gets copied $k+1$ times, so that a copy exists for each possible recursion depth. Transitions get added at each recursion depth d according to Figure 3 and Figure 4, whereas some transitions in the latter also connect to places of recursion depth $d+1$.

Regarding the transducers, we use the alphabet $\{0, 1\}$. Every place address $w = u.v$ has a prefix u of length $\lceil \log h \rceil$ and a postfix v of length $\lceil \log k \rceil$. We assign each of the h places that $N(\mathcal{N})$ started with a number from 0 to $h-1$. The binary representation of this number (with leading zeros) is used for the u-part of its address. For the v-part, we use the binary representation of the recursion depth d (also with leading zeros), that a particular copy of this place corresponds to. The address of the place corresponding to the first label in the main program at recursion depth 0 is used for w_{init}, whereas the one corresponding to w_{halt} at recursion depth 0 is used for w_{final}.

To accept a particular pair or triple of addresses as a transition, each of the three transducers distinguishes between all possibilities regarding the u-parts. Any pair or triple of $\lceil \log h \rceil$-length words that matches a particular transition of the right type (move, fork, join) has a unique path in the transducer, while all non-matching pairs or triples do not. Then for the v-parts, the transducer needs to either check for equality, if all places correspond to the same recursion depth, or for one binary represented number to be one higher. Since it is clear from the u-parts, whether the recursion depths should all match or not, we can just connect the unique paths to the correct part of the transducer at the end.

The transducer parts for the first $\lceil \log h \rceil$ bits require to distinguish between up to $2^{3\log h} = 8h$ possibilities, meaning they require polynomially in h many states. The parts for the last $\lceil \log k \rceil$ bits can easily be constructed using polynomially many states in $\log k$. Since h is linear in the number of commands in R and $\lceil \log k \rceil$ is the size of the binary encoding of the maximum recursion depth, $\mathcal{N}$ is of polynomial size compared to R. Because we can construct $\mathcal{N}$ by first constructing $N(\mathcal{N})$ without the copies for each recursion depth, this is feasible in polynomial time. Thus, the coverability problem for transducer-defined Petri nets is 2EXPSPACE-hard.

5 Discussion

The chain of reductions in Sections 3 and 4 complete the 2EXPSPACE lower bound for 1-bounded reachability for DCPS. In fact, an inspection of the reductions show a technical strengthening: the 2EXPSPACE lower bound already holds for SRP[1] of DCPS which satisfy two additional properties, *boundedness* and *local termination*.

▶ **Definition 8.** *A* DCPS $\mathcal{A}$ *is said to be* bounded *if there is a global bound $B \in \mathbb{N}$ on the size of every configuration of every run of $\mathcal{A}$. It is* locally terminating *if every infinite run of $\mathcal{A}$ contains infinitely many context switches.*

Consider the chain of reductions from the halting problem for bounded counter programs to RNP to TDPN to SRP[1]. The configurations of the counter programs, by definition, are bounded by a triply-exponential bound on the parameter n. This bound translates to bounds on the RNP and TDPN instances. In particular, the number of places in the TDPN produced in the reduction is exponentially bounded in n and the number of tokens on these places is triple-exponentially bounded in n. The DCPS constructed from the TDPN uses the stack of a thread to store an address of a place; thus, the height of a stack is bounded by a polynomial in n. In addition, since the number of tokens in the TDPN correspond to the number of in-progress threads in the DCPS, this implies a triple exponential (in n) bound on the number of threads in any execution of the DCPS. Thus, the size of every configuration in every run of the DCPS is bounded.

Second, the rules of the constructed DCPS do not allow any one thread to run indefinitely. In other words, any non-terminating run of the DCPS must involve infinitely many threads and the run contains infinitely many context switches.

▶ **Theorem 9.** *The* SRP[1] *problem for bounded, locally terminating* DCPS *is* 2EXPSPACE-*hard.*

References

1 Mohamed Faouzi Atig, Ahmed Bouajjani, and Shaz Qadeer. Context-bounded analysis for concurrent programs with dynamic creation of threads. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pages 107–123, 2009.

2 Byron Cook, Daniel Kroening, and Natasha Sharygina. Verification of Boolean programs with unbounded thread creation. *Theoretical Computer Science*, 388(1-3):227–242, 2007. doi:10.1016/j.tcs.2007.07.050.

3 Stéphane Demri, Diego Figueira, and M. Praveen. Reasoning about data repetitions with counter systems. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 33–42, 2013.

4 Emanuele D'Osualdo, Jonathan Kochems, and C.-H. Luke Ong. Automatic verification of Erlang-style concurrency. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013, Proceedings*, volume 7935 of *Lecture Notes in Computer Science*, pages 454–476. Springer, 2013.

5 Javier Esparza. Decidability and complexity of Petri net problems – an introduction. In G. Rozenberg and W. Reisig, editors, *Lectures on Petri Nets I: Basic Models. Advances in Petri Nets*, number 1491 in Lecture Notes in Computer Science, pages 374–428, 1998.

6 Hana Galperin and Avi Wigderson. Succinct representations of graphs. *Information and Control*, 56(3):183–198, 1983.

7 Pierre Ganty and Rupak Majumdar. Algorithmic verification of asynchronous programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 34(1):6, 2012.

8 Alexander Kaiser, Daniel Kroening, and Thomas Wahl. Dynamic cutoff detection in parameterized concurrent programs. In *22nd International Conference on Computer Aided Verification, CAV 2010, Edinburgh, UK, July 15-19, 2010, Proceedings*, pages 645–659. Springer, 2010.

9 Jonathan Kochems. *Verification of asynchronous concurrency and the shaped stack constraint*. PhD thesis, University of Oxford, UK, 2014. URL: `http://ora.ox.ac.uk/objects/uuid:cd487639-0e7f-4248-9405-e05e8a8383d5`.

10 Salvatore La Torre, Parthasarathy Madhusudan, and Gennaro Parlato. The language theory of bounded context-switching. In *LATIN 2010: Theoretical Informatics, 9th Latin American Symposium, Oaxaca, Mexico, April 19-23, 2010, Proceedings*, volume 6034 of *Lecture Notes in Computer Science*, pages 96–107. Springer, 2010.

11 Akash Lal and Thomas W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1):73–97, 2009. `doi:10.1007/s10703-009-0078-9`.

12 Ernst Leiss. Succinct representation of regular languages by Boolean automata. *Theoretical Computer Science*, 13(3):323–330, 1981. `doi:10.1016/S0304-3975(81)80005-9`.

13 Richard Lipton. The reachability problem is exponential-space hard. *Yale University, Department of Computer Science, Report*, 62, 1976.

14 Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, PLDI 2007, San Diego, CA, USA, June 10-13, 2007*, pages 446–455. ACM, 2007.

15 Christos H. Papadimitriou and Mihalis Yannakakis. A note on succinct representations of graphs. *Information and Control*, 71(3):181–185, 1986.

16 Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3440 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2005.

17 Charles Rackoff. The covering and boundedness problems for vector addition systems. *Theoretical Computer Science*, 6(2):223–231, 1978.