Verification of Asynchronous Programs with Nested Locks

Mohamed Faouzi Atig[1], Ahmed Bouajjani[2], K. Narayan Kumar*[3], and Prakash Saivasan[4]

1 **Uppsala University, Sweden**
mohamed_faouzi.atig@it.uu.se
2 **IRIF, Université Paris Diderot, France**
abou@irif.fr
3 **Chennai Mathematical Institute and UMI RELAX, Chennai, India**
kumar@cmi.ac.in
4 **TU Braunschweig, Germany**
p.saivasan@tu-bs.de

Abstract

In this paper, we consider asynchronous programs consisting of multiple recursive threads running in parallel. Each of the threads is equipped with a multi-set. The threads can create tasks and post them onto the multi-sets or read a task from their own. In addition, they can synchronise through a finite set of locks. In this paper, we show that the reachability problem for such class of asynchronous programs is undecidable even under the nested locking policy. We then show that the reachability problem becomes decidable (EXP-SPACE-complete) when the locks are not allowed to be held across tasks. Finally, we show that the problem is NP-complete when in addition to previous restrictions, threads always read tasks from the same state.

1998 ACM Subject Classification D.2.4 Software, Program Verification

Keywords and phrases Asynchronous programs, Nested-locks, Reachability Problem

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2017.11

1 Introduction

Asynchronous programming is widely used in building efficient and responsive software. Jobs are decomposed in tasks that are delegated to different threads running in parallel. These threads share a global memory, and they also have their own local memory. In addition, each thread has an unbounded buffer where the tasks posted to the thread are stored. These tasks are handled by the thread in a serial manner, i.e., by running each task until completion before taking another one. Each task corresponds to the execution of a sequential program that can access both (thread-)local and global variables, call (potentially recursive) procedures, and create new tasks that are posted to designated threads.

The behaviours of asynchronous programs can be extremely intricate and unpredictable due to the dynamic creation of concurrent tasks. This makes the reasoning about asynchronous programs extremely hard.

Therefore, developing automated techniques for the verification of asynchronous programs is an important and challenging research topic. In the case of single-thread asynchronous

* Partially supported by Indo-French Project AVeCSO, Indo-Swedish DST-VR Project P-02/2014 and the Infosys Foundation

37th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2017).
Editors: Satya Lokam and R. Ramanujam; Article No. 11; pp. 11:1–11:14

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

programs, it has been shown that the reachability problem is decidable and EXPSPACE-complete (assuming that the data domain is finite) [2,5]. However, this problem is undecidable in general, and this is the case even for two threads handling only one task each [11]. Therefore, decidable instances of this problem can be obtained only by considering either under-approximations for bug detection using, e.g., bounded analyses [1, 4, 10], or by considering over-approximations for establishing absence of bugs, using abstract analyses that focus on relevant aspects.

In the context of abstract analyses of concurrent programs, one useful approach is abstracting away the content of the shared variables carrying data while keeping precise the content of the control variables, such as *locks*, that are used for synchronisation. Indeed, concurrency bugs are in general due to a misuse of synchronisation allowing unexpected interleavings of concurrent actions. Therefore, assuming that locks are the only shared variables between threads retains the relevant information, without being too coarse, when reasoning about the existence of, e.g., data races and deadlocks. This approach has been introduced in [8], where it has been shown that, for multi-thread programs with locks (and no task creation), the reachability problem is undecidable in general, and that this problem becomes decidable when locks are used (i.e., acquired and released) in a *well-nested* manner. This problem has been investigated further for larger classes of programs by other authors, e.g., in [6,9]. In particular, it has been shown that for dynamic networks of pushdown systems (modelling concurrent programs with thread creation), which is a class of models with a decidable reachability problem [3] that is incomparable with asynchronous programs, the extension with well-nested locking preserves the decidability of the reachability problem [6]. The goal of this paper is to investigate the decidability and the complexity of the reachability problem for asynchronous programs with nested locking.

We first prove that, surprisingly, the reachability of asynchronous programs with nested locks is undecidable as soon as four threads are considered (two with unbounded call stacks, and two being finite-state). However, we provide a condition on the use of locks by threads across task handling phases that leads to decidability. In fact, we found that the source of undecidability (even under nested locking) is the transfer of locks between tasks executed successively on a single thread. Therefore, we require that (1) a thread should not hold any lock when it starts handling a task, and (2) all locks acquired during the execution of a task must be released before completion of the task, i.e., when the handling of a task is completed, the set of locks held by the thread must be again empty. Technically, we define a task-locking policy that requires that every thread should not hold any lock when its stack is empty. We prove that under this policy, the reachability problem of asynchronous programs with nested locks is in EXPSPACE. The proof is by a polynomial reduction to the case of single-thread asynchronous programs using a nontrivial serialisability argument. Importantly, despite the high worst case complexity, there are existing work for solving efficiently the reachability problem of single-threaded asynchronous programs in practice [7].

Moreover, we consider an interesting and practically relevant case for which we establish a better complexity. In fact, we consider that while tasks should be allowed to communicate and synchronise through the shared global memory, they should not communicate through the thread-local shared memory. Indeed, this would assume that the tasks rely on the order they are scheduled, which is in general not under the control of the programmer. So, it is quite natural to assume that before termination, a task can put the result of its computation in the shared memory, and it can also create a new task that will be its continuation (when it will be scheduled later), but that the thread does not transfer any information to the next task it executes. (Actually, asynchronous programs have typically User Interface threads

(UI's) that consist of loops receiving jobs (or reacting to events), and creating for them handlers running in parallel.) Technically, we consider that a thread always starts handling tasks in the same state. Interestingly, we prove that under this assumption the reachability problem becomes in NP. The proof is by a reduction to the reachability problem in the case where the number of task handling phases is polynomially bounded.

To summarise, we prove that by forbidding transfers of locks between tasks executing on a same thread, the reachability problem of multithreaded asynchronous programs with nested locks is EXPSPACE-complete. Furthermore, we prove that by forbidding transfers of local states between tasks executing on a same thread, the reachability problem becomes NP-complete. Our results open the door to the development of efficient and complete methods for verifying asynchronous programs against concurrency bugs.

2 Preliminaries

Let Σ be a finite alphabet. We use Σ^* and Σ^+ to denote the set of all finite words and non-empty finite words, respectively. We use ϵ to denote the empty word. We write Σ_ϵ to denote $\Sigma \cup \{\epsilon\}$. For $w = a_1 a_2 \dots a_n \in \Sigma^*$, we let $|w| = n$, $w[i] = a_i$ and $w[i,j]$ to denote the length of w, the i^{th} letter a_i and the subword $a_i \dots a_j$, respectively. Given a word $w \in \Sigma$ and $\Sigma' \subseteq \Sigma$, we let $w \downarrow_{\Sigma'}$ to denote the projection of w onto Σ' .

A multi-set over Σ is a function $M : \Sigma \mapsto \mathbb{N}$. We denote by $M[\Sigma]$ the set of all multi-sets over Σ and by $\emptyset$ the empty multi-set. Given two multi-sets M and M', we write $M' \leq M$ iff $M'(a) \leq M(a)$ for all $a \in \Sigma$. We denote by $M + M'$ the multi-set formed by $(M + M')(a) = M(a) + M'(a)$ for all $a \in \Sigma$. For $M \geq M'$, $M - M'$ is defined in a similar manner. For any word $w \in \Sigma^*$, we denote by $\lfloor w \rfloor$ the multi-set formed by counting the number of occurrences of each letter from Σ in w.

A *pushdown automaton (PDA)* is a tuple $\mathbf{P} = (Q, \Gamma, \Sigma, \delta, s_0, \alpha_0)$ where Q is the finite set of states, Γ is the finite stack alphabet, Σ is the finite input alphabet, $s_0 \in Q$ is the initial state, $\alpha_0 \in \Gamma$ is the initial stack symbol, and δ is the transition relation. We assume that Γ contains the special stack bottom symbol $\perp$. The transition set δ is a subset of $Q \times \Gamma \times \Sigma_\epsilon \times \Gamma^* \times Q$ with the restrictions that: (1) if $\tau = (q, \alpha, a, \beta, q') \in \delta$ then $|\beta| \leq 2$ and (2) $\beta \in \{b\perp \mid b \in \Gamma_\epsilon\}$ when $\alpha = \perp$. We use $\mathbf{Src}(\tau) = q$ (resp. $\mathbf{Dest}(\tau) = q'$) to refer to the head (resp. tail) state q (resp. q') of the transition τ, and $\lambda(\tau)$ to denote the label a.

A configuration of $\mathbf{P}$ is a pair (q, γ) with $q \in Q$ and $\gamma \in ((\Gamma \setminus \{\perp\})^* \cdot \{\perp\})$. Given a configuration $c = (q, \gamma)$, we use $\mathbf{Stt}(c)$ (resp. $\mathbf{Stk}(c)$) to refer to the state q (resp. the stack component γ). The initial configuration of $\mathbf{P}$ is defined by the pair $(s_0, \alpha_0\perp)$. The transition relation $\xrightarrow{\tau}_{\mathbf{P}}$, with $\tau \in \delta$, relating pairs of configurations, is defined as the smallest relation satisfying the following condition: $(q, \alpha\gamma) \xrightarrow{\tau}_{\mathbf{P}} (q', \beta\gamma)$ if τ is of the form $(q, \alpha, a, \beta, q')$. This transition corresponds to the pop of the symbol α and the push of the word β.

We often omit the reference to $\mathbf{P}$ and write $\xrightarrow{\tau}$ when $\mathbf{P}$ is clear from the context. Sometimes, we omit τ and simply write $\rightarrow$ when the reference to τ is not important. We write $(q, \gamma) \xrightarrow{\sigma} (q', \gamma')$ for $\sigma = \tau_1 \dots \tau_n \in \delta^*$ to mean that there is a sequence of transitions of the form $(q, \gamma) = (q_0, \gamma_0) \xrightarrow{\tau_1} (q_1, \gamma_1) \xrightarrow{\tau_2} \dots \xrightarrow{\tau_{n-1}} (q_{n-1}, \gamma_{n-1}) \xrightarrow{\tau_n} (q_n, \gamma_n) = (q', \gamma')$. Given two configurations c_1, c_2, we use $L(\mathbf{P}, c_1, c_2)$ to denote the set of words w such that there is $\tau_1, \dots, \tau_n \in \delta$ with $c_1 \xrightarrow{\tau_1\tau_2\cdots\tau_n} c_2$ and $w = \lambda(\tau_1)\lambda(\tau_2)\cdots\lambda(\tau_n)$. We use $L(\mathbf{P}, c_2)$ to denote $L(\mathbf{P}, c_1, c_2)$ where $c_1 = (s_0, \alpha_0\perp)$ is the initial configuration.

3 Model

Multiset PushDown Systems (MPDS) have been introduced by Sen and Viswanathan as a formal model for asynchronous programs [12]. An MPDS model consists of a pushdown automaton equipped with a multiset. The multiset is used to store pending tasks (i.e., stack symbols). When the stack is empty, a pending task is taken, in non-deterministic manner, from the multiset and put into the stack. Then, the system starts the execution of the chosen task following the pushdown transition rules with the ability to create new tasks (that will be added to the multiset). In this paper, we consider a generalization of multi-set pushdown systems (MPDS) called N-Multi-set Pushdown Systems (N-MPDS) where $N \in \mathbb{N}$ denotes the (fixed) number of threads executing in parallel. An N-MPDS consists of a collection of pushdown automata (each one comes with its own multi-set). When the stack of a pushdown automaton is empty, a task is taken from its associated multi-set and executed. During the execution of a task, newly created tasks are added to a pre-determined multi-set. Furthermore, the pushdown automaton can communicate through a finite set of locks (i.e., each pushdown automaton can acquire and release a given particular lock). Thus, an MPDS (resp. pushdown automata communicating via locks [8]) corresponds to the particular case where we have one pushdown automaton (i.e., $N = 1$) (resp. there is no task creation).

▶ **Definition 1.** An N-MPDS over the (finite) set of locks $\mathcal{L}$ is a tuple $A = (\Sigma, \mathcal{P}, \mathcal{L})$, where Σ is a finite set of tasks and $\mathcal{P} = \{\mathbf{P}_i \mid 1 \leq i \leq N\}$ is a collection of pushdown automata (or threads) $\mathbf{P}_i = (Q_i, \Gamma_i, \mathcal{O}_i, \delta_i, s_i, \alpha_i)$, where $\mathcal{O}_i = \{i!j(a), i?a \mid a \in \Sigma, 1 \leq j \leq N\} \cup \{\mathrm{lck}_i(l), \mathrm{rel}_i(l) \mid l \in \mathcal{L}\}$. Here $i!j(a)$ means that the thread i creates a task labeled by a and adds it to the multi-set of thread j. While $i?a$ means that the thread i picks a pending task labeled by a from its multi-set. Furthermore, $\mathrm{lck}_i(l)/\mathrm{rel}_i(l)$ corresponds to acquiring / releasing of the lock l by the thread i. We assume that the sets δ_i, $1 \leq N$, are disjoint and let $\delta = \bigcup_{1 \leq i \leq N} \delta_i$. We also *require* that (1) $\delta_i \cap (Q_i \times (\Gamma_i \setminus \{\bot\}) \times \{i?a \mid a \in \Sigma\} \times \Gamma_i^* \times Q_i) = \emptyset$ for all $i \in \{1, \ldots, N\}$ (i.e., the execution of pending tasks can only be performed when the stack is empty), and (2) if a transition is of the form $(q, \bot, b, \beta\bot, q')$ is in δ, with $\beta \in \Sigma$, then b is of the form $i?\beta$ for some $i \in \{1, \ldots, N\}$ (i.e., the thread i picks a pending task β and adds it to its empty stack).

A configuration of an N-MPDS A is a triple of functions $(\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l})$ where, for each $1 \leq i \leq N$, $\boldsymbol{c}(i)$ is a configuration of $\mathbf{P}_i$, $\boldsymbol{m}(i)$ is a multi-set over Σ (representing the set of tasks waiting to be executed by the thread i) and $\boldsymbol{l}(i) \subseteq \mathcal{L}$ is the set of locks held by thread i. We require that $\boldsymbol{l}(i) \cap \boldsymbol{l}(j) = \emptyset$ for all $i \neq j$. The initial configuration is defined $(\boldsymbol{c_0}, \boldsymbol{m_0}, \boldsymbol{l_0})$, where $\boldsymbol{c_0}(i) = (s_i, \bot), \boldsymbol{m_0}(i) = \lfloor \alpha_i \rfloor$ and $\boldsymbol{l_0}(i) = \emptyset$ for all $i, 1 \leq i \leq N$ (i.e., the stack of each thread is empty and there is only one pending task per thread and all locks are free). Observe that the definition of the initial configuration is equivalent to the one where each pushdown automaton is in its initial configuration, there is no pending task and all locks are free. We only use the former for the sake of simplicity.

Observe that an MPDS [12] can be defined as a 1-MPDS $A = (\Sigma, \mathcal{P}, \mathcal{L})$ whose set of locks is empty (i.e., $\mathcal{L} = \emptyset$) and set of threads $\mathcal{P}$ consists of only one pushdown automaton $\mathbf{P}_1 = (Q_1, \Gamma_1, \mathcal{O}_1, \delta_1, s_1, \alpha_1)$. To simplify the presentation, we use $(\Sigma, \mathbf{P}_1)$ to denote the MPDS A. Further, we use $(\boldsymbol{c}(1), \boldsymbol{m}(1))$ to denote a configuration of A instead of $(\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l})$.

Given two configurations $(\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l})$ and $(\boldsymbol{c'}, \boldsymbol{m'}, \boldsymbol{l'})$ and a transition $\tau \in \delta$, we use $(\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l}) \xrightarrow{\tau}_A (\boldsymbol{c'}, \boldsymbol{m'}, \boldsymbol{l'})$ to denote that there is an index $i \in \{1, \ldots, N\}$ such that $\tau \in \delta_i$, $\boldsymbol{c}(i) \xrightarrow{\tau}_{\mathbf{P}_i} \boldsymbol{c'}(i)$, $\forall j \neq i$ we have $\boldsymbol{c}(j) = \boldsymbol{c'}(j)$ and further one of the following holds:

- $\lambda(\tau) = i!j(a)$: $\boldsymbol{m'}(j) = \boldsymbol{m}(j) + \lfloor a \rfloor$, $\boldsymbol{m}(k) = \boldsymbol{m'}(k)$ for all $k \neq j$ and $\boldsymbol{l} = \boldsymbol{l'}$. (The thread i creates a task of type a and adds it to the multiset of the thread j.)
- $\lambda(\tau) = i?a$: $\boldsymbol{m}(i)(a) > 1$, $\boldsymbol{m'}(i) = \boldsymbol{m}(i) - \lfloor a \rfloor$, $\boldsymbol{m}(j) = \boldsymbol{m'}(j)$ for all $k \neq i$, and $\boldsymbol{l} = \boldsymbol{l'}$.

(The thread i picks a task a from its multiset and adds it to its stack. Recall that removing a task from the multiset is possible only if the stack is empty.)

- $\lambda(\tau) = \text{lck}_i(l)$: $l \notin \bigcup_{1 \leq k \leq N} \boldsymbol{l}(k)$, $\boldsymbol{l'}(i) = \boldsymbol{l}(i) \cup \{l\}$, $\boldsymbol{m}(i) = \boldsymbol{m'}(i)$, $\boldsymbol{l}(k) = \boldsymbol{l'}(k)$ and $\boldsymbol{m}(k) = \boldsymbol{m'}(k)$ for all $k \neq i$. (The thread i acquires the lock l if it is not already held.)
- $\lambda(\tau) = \text{rel}_i(l)$: $l \in \boldsymbol{l}(i)$, $\boldsymbol{l'}(i) = \boldsymbol{l}(i) \setminus \{l\}$, $\boldsymbol{m}(i) = \boldsymbol{m'}(i)$, $\boldsymbol{l}(k) = \boldsymbol{l'}(k)$ and $\boldsymbol{m}(k) = \boldsymbol{m'}(k)$ for all $k \neq i$. (The thread i releases the lock l.)

An execution π of A is an alternating sequence $(\boldsymbol{c_1}, \boldsymbol{m_1}, \boldsymbol{l_1}) \cdot \tau_1 \cdot (\boldsymbol{c_2}, \boldsymbol{m_2}, \boldsymbol{l_2}) \cdot \tau_2 \cdots (\boldsymbol{c_{n-1}}, \boldsymbol{m_{n-1}}, \boldsymbol{l_{n-1}}) \cdot \tau_{n-1} \cdot (\boldsymbol{c_n}, \boldsymbol{m_n}, \boldsymbol{l_n})$ of configurations and transitions such that $(\boldsymbol{c_i}, \boldsymbol{m_i}, \boldsymbol{l_i}) \xrightarrow{\tau_i}_A (\boldsymbol{c_{i+1}}, \boldsymbol{m_{i+1}}, \boldsymbol{l_{i+1}})$, $\forall i \in \{1, \ldots, n-1\}$. For configurations $(\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l})$ and $(\boldsymbol{c'}, \boldsymbol{m'}, \boldsymbol{l'})$, we write $(\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l}) \xrightarrow{\sigma}_A (\boldsymbol{c'}, \boldsymbol{m'}, \boldsymbol{l'})$ to denote that there is an execution $\pi = (\boldsymbol{c_1}, \boldsymbol{m_1}, \boldsymbol{l_1}) \cdot \tau_1 \cdot (\boldsymbol{c_2}, \boldsymbol{m_2}, \boldsymbol{l_2}) \cdot \tau_2 \cdots (\boldsymbol{c_{n-1}}, \boldsymbol{m_{n-1}}, \boldsymbol{l_{n-1}}) \cdot \tau_{n-1} \cdot (\boldsymbol{c_n}, \boldsymbol{m_n}, \boldsymbol{l_n})$ such that $\sigma = \tau_1 \tau_2 \cdots \tau_{n-1}$, $(\boldsymbol{c_1}, \boldsymbol{m_1}, \boldsymbol{l_1}) = (\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l})$ and $(\boldsymbol{c_n}, \boldsymbol{m_n}, \boldsymbol{l_n}) = (\boldsymbol{c'}, \boldsymbol{m'}, \boldsymbol{l'})$. Sometimes we write the execution π as $(\boldsymbol{c_1}, \boldsymbol{m_1}, \boldsymbol{l_1}) \xrightarrow{\tau_1}_A (\boldsymbol{c_2}, \boldsymbol{m_2}, \boldsymbol{l_2}) \ldots (\boldsymbol{c_{n-1}}, \boldsymbol{m_{n-1}}, \boldsymbol{l_{n-1}}) \xrightarrow{\tau_{n-1}}_A (\boldsymbol{c_n}, \boldsymbol{m_n}, \boldsymbol{l_n})$.

Reachability problem. Given a N-MPDS $A = (\Sigma, \mathcal{P}, \mathcal{L}, \Delta)$ (as defined above) and a function $\boldsymbol{r}$ (referred to as the destination) that assigns to each $i : 1 \leq i \leq N$ a state from $Q_i \setminus \{s_i\}$, the *reachability problem* asks if there is an execution of the form $(\boldsymbol{c_0}, \boldsymbol{m_0}, \boldsymbol{l_0}) \xrightarrow{\sigma}_A (\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l})$, with $(\boldsymbol{c_0}, \boldsymbol{m_0}, \boldsymbol{l_0})$ is the initial configuration, for some $\boldsymbol{c}, \boldsymbol{m}$ and $\boldsymbol{l}$ such that for all $i : 1 \leq i \leq N$, we have $\textbf{Stt}(\boldsymbol{c}(i)) = \boldsymbol{r}(i)$.

First note that, without the assumption of removing a pending task from the multi-set when the stack of the thread is empty, we can simulate the intersection of two pushdown automata by a 2-MPDS, without any locks, by using the multi-sets as a synchronizing mechanism. Given two pushdown automata over an alphabet Σ, we construct a 2-MPDS with task alphabet $\Sigma \cup \{\#\}$. The simulation proceeds as follows: The first thread guesses a letter a, simulates a step of the first PDS, posts this letter a as a task to the second thread and waits for a task of type $\#$. The second thread nondeterministically guesses the next letter a, picks up a task of this type from its multi-set, simulates a step and then posts the task $\#$ to the first thread. Thus, we may simulate both the pushdowns on the same input word and the reachability problem for 2-MPDS without locks is rendered undecidable. Observe that, in this simulation, values are removed from the multi-sets at will and this goes against the spirit of our model: the multi-sets were introduced to hold the tasks that await execution. A thread should execute these (recursive) tasks one after another. In particular a task should be removed from the multi-set for execution only when the previous task has completed. When a task is completed, the call stack of the thread should be empty.

Second, the reachability problem for 2-MPDS without multi-sets is undecidable in the presence of locks. This follows from the undecidability of the reachability problem for pushdown automata synchronizing using locks [11]. Thus, we need restrictions on the usage of locks as well. One well-known restriction that yields decidability for networks of pushdown automata is that of *nested locking*. Nested locking, introduced by Kahlon et al. [8], requires that locks be released in the same order as they are acquired. This is formalized as follows.

Nested Locking. Given an execution of the form $(\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l}) \xrightarrow{\tau_1}_A (\boldsymbol{c_1}, \boldsymbol{m_1}, \boldsymbol{l_1}) \xrightarrow{\tau_2}_A (\boldsymbol{c_2}, \boldsymbol{m_2}, \boldsymbol{l_2}) \ldots (\boldsymbol{c_{n-1}}, \boldsymbol{m_{n-1}}, \boldsymbol{l_{n-1}}) \xrightarrow{\tau_n}_A (\boldsymbol{c'}, \boldsymbol{m'}, \boldsymbol{l'})$, we say positions $i, j \in \{1, \ldots, n\}$ form a *acquire-release pair* if some lock l is acquired in the ith transition and released by the jth transition and this lock is not acquired or released in between i.e. there is a $k : 1 \leq k \leq N$, and $l \in \mathcal{L}$ such that $\lambda(\tau_i) = \text{lck}_k(l)$, $\lambda(\tau_j) = \text{rel}_k(l)$ and $\forall r \in \{i+1, \ldots, j-1\}, \lambda(\tau_r) \notin \bigcup_{1 \leq m \leq N} \{\text{lck}_m(l), \text{rel}_m(l)\}$. We write $i \curvearrowright_k j$ to indicate this. An execution of the form $\pi = (\boldsymbol{c_0}, \boldsymbol{m_0}, \boldsymbol{l_0}) \xrightarrow{\sigma} (\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l})$ is

said to be well-nested iff there are no positions i, j, i', j' such that $i < i' < j < j'$ and $i \curvearrowright_k j$ and $i' \curvearrowright_k j'$ for some $k, 1 \leq k \leq N$.

An N-MPDS A is said to be *well-nested* if all its executions of the form $\pi = (\boldsymbol{c_0}, \boldsymbol{m_0}, \boldsymbol{l_0}) \xrightarrow{\sigma} (\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l})$ are well-nested. Recall that $(\boldsymbol{c_0}, \boldsymbol{m_0}, \boldsymbol{l_0})$ is the initial configuration. As indicated earlier, Kahlon et al. [8] show that the reachability problem restricted to nested locking is decidable for networks of pushdown automata with locks. However, in the presence of multi-sets and locks, the nested locking assumption is still insufficient to obtain decidability.

4 Undecidability of the Reachability Problem for Well-Nested N-MPDSs

In this section we show that the reachability problem for N-MPDSs remains undecidable even when assuming the well nested locking policy. In particular we prove:

▶ **Theorem 2.** *The reachability problem for well-nested 4-MPDSs is undecidable.*

Proof sketch. Let P^1 and P^2 be two pushdown automata over an alphabet Σ. We construct a 4-MPDS A that simulates joint executions of P^1 and P^2. In the following, we assume w.l.o.g. that there are no ϵ moves in P^1 and P^2 (the undecidability result holds even with such an assumption). The MPDS A has four components P_1, P_2, P_3 and P_4 where P_1 and P_2 simulate P^1 and P^2 respectively, using the agents P_3 and P_4 to ensure that the simulations follow the same input word. In fact, P_3 and P_4 will not use their respective stacks.

The system A uses two locks l_1 and l_2 and the set of tasks is given by $\Sigma \cup \{a, b, r, l\}$. The simulation begins with an initialization step and this is followed by a sequence of steps, where in each step the threads P_1 and P_2 simulate a run of P^1 and P^2 on one letter.

In the initialization step, the thread P_3 acquires the lock l_1 and sends the task b to both P_1 and P_2 instructing them to begin the simulation. Both threads P_1 and P_2 await for the task b and begin their simulation on receiving this task. At the end of this initialization step (and at the beginning of each of the subsequent steps) the lock l_1 is with P_3 and l_2 is free and all the task multi-sets are empty.

In each step, the thread P_i, $1 \leq i \leq 2$, does the following: takes lock l_2, continues the simulation of P^i by reading a letter $c \in \Sigma$, posts the task c to the thread P_3, releases lock l_2 and then waits to take lock l_1. When available it takes l_1 and releases it immediately to complete its execution of the step.

In each step, the thread P_3 does the following: guesses a task type $c \in \Sigma$, removes two copies of c from its multi-set (ensuring that P_1 and P_2 have carried out simulations on the same letter), sends the task l to the thread P_4 (instructing it to take the lock l_2), waits for the task a (an acknowledgment from P_4 that it has indeed taken the lock l_2), releases the lock l_1 (to enable P_1 and P_2 to complete the concluding part of their execution of this step), retakes lock l_1, sends the task r to P_4 (instructing it to release the lock l_2) and waits for the task a (an acknowledgment from P_4 that it has indeed released the lock l_2).

In each step, the thread P_4 awaits the task l, then takes the lock l_2, sends the task a to P_3 in acknowledgment, awaits the task r, then releases lock l_2 and sends the task a to P_3.

It is clear that in each step, the simulation of both pushdowns is extended by a run on the same letter from Σ. We still have to argue that this protocol ensures that the threads proceed step by step (i.e., some of them cannot go ahead before the others are ready to participate in the next step). In each step, after simulating a run of the pushdowns both threads P_1 and P_2 have to wait for lock l_1 to be released. This is possible only after P_3 has verified that they have both used identical letters in their simulation. When the lock

l_1 is available for them to complete their executions of this step, the lock l_2 is guaranteed to be held by P_4 (since P_3 releases l_1 only after confirmation from P_4 that the lock l_2 has been taken). Thus after completing the current step P_1 and P_2 cannot proceed to the next step of the simulation (until l_2 is free). The thread P_3 takes back the lock l_1 before l_2 is released by P_4 and thus the locks are returned to the required state before the next step in the simulation begins.

It is possible that the lock l_1 is taken back by P_3 before P_1 or P_2 (or both) complete their final operations in the step. In this case, the system deadlocks since the thread that failed to complete will wait for l_1 while P_3 will wait for a task from Σ to be posted by this waiting thread. Thus the simulating threads can neither get ahead nor fall behind in each step. ◀

5 Well-Nested N-MPDS under the Task Locking Policy

As we have seen, allowing the transfer of locks, even in a nested manner, from a task to another task leads to the undecidability of the reachability problem for N-multi-set pushdown systems. Therefore, we consider an additional constraint on the locking policy. The new constraint consists in requiring that threads do not hold any locks when their stack is empty. This restriction can be understood as follows: the threads can be thought of as *schedulers* that pick and execute tasks. As tasks are executed to completion, a new task is picked only when the stack is empty, this amounts to requiring that it is mainly the tasks that use locks and not threads. However, since the thread and the tasks share the local state space, the thread is still allowed to pass a finite amount of local state information to the tasks, but it is not allowed to pass or receive locks. We shall return to this point in the next section.

Task Locking Policy. Let $A = (\Sigma, \mathcal{P}, \mathcal{L})$ be an N-MPDS. We say that an execution $(\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l}) \xrightarrow{\tau_1}_A (\boldsymbol{c_1}, \boldsymbol{m_1}, \boldsymbol{l_1}) \xrightarrow{\tau_2}_A (\boldsymbol{c_2}, \boldsymbol{m_2}, \boldsymbol{l_2}) \ldots (\boldsymbol{c_{n-1}}, \boldsymbol{m_{n-1}}, \boldsymbol{l_{n-1}}) \xrightarrow{\tau_n}_A (\boldsymbol{c}_n, \boldsymbol{m}_n, \boldsymbol{l}_n)$ is a *task-locking* execution if for each $i : 1 \leq i \leq n$ and for each $j : 1 \leq j \leq N$, if $\mathbf{Stk}(\boldsymbol{c_i}(j)) = \perp$ then $\boldsymbol{l_i}(j) = \emptyset$. The N-MPDS A is under the *task-locking* policy if all its executions of the form $\pi = (\boldsymbol{c_0}, \boldsymbol{m_0}, \boldsymbol{l_0}) \xrightarrow{\sigma}_A (\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l})$ are *task-locking* executions.

Our main result is that the reachability problem for well-nested N-MPDSs under the task-locking policy is decidable and is Expspace-Complete.

▶ **Theorem 3.** *The reachability problem for well-nested MPDSs under the task-locking policy is* Expspace-Complete.

The lower bound follows immediately from the Expspace-hardness of MPDS [12]. The rest of this section is dedicated to prove the upper-bound (namely that the reachability problem for well-nested N-MPDSs under the task-locking policy is in Expspace). As a first step, we show that one may *serialize* each execution of the system in such a way that (i) completed tasks can be executed atomically (ii) incomplete tasks (which are at most one per thread) can broken up in a bounded number of segments such that each segment can be executed atomically. This will be used in the next step to polynomially reduce our problem to the reachability in a (single) pushdown system with multi-sets.

For the rest of the section, let us fix an N-MPDS $A = (\Sigma, \mathcal{P}, \mathcal{L}, \Delta)$ where $\mathcal{P} = \{\mathbf{P}_i \mid 1 \leq i \leq N\}$ with $\mathbf{P}_i = (Q_i, \Gamma_i, \mathcal{O}_i, \delta_i, s_i, \alpha_i)$. We will further assume w.l.o.g. that in the N-MPDS A, every thread starts its execution by removing a task from the multi-set. Note that any N-MPDS can be easy transformed into an equivalent one of that form.

5.1 Serialized Executions

Consider an execution $\rho = (\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l}) \xrightarrow{\sigma} (\boldsymbol{c'}, \boldsymbol{m'}, \boldsymbol{l'})$ of A starting at the initial configuration (i.e., $(\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l}) = (\boldsymbol{c}_0, \boldsymbol{m}_0, \boldsymbol{l}_0)$). Let σ_i be the projection of σ on the set δ_i, that is, it is the sequence of transitions executed by thread i along the execution ρ. Then, σ_i can be decomposed uniquely into a sequence of the form: $\tau_1\alpha_2\tau_2 \dots \alpha_j\tau_j\beta$ where, $\tau_1, \dots, \tau_j$ are the transitions in σ_i that remove a task from the multi-set (i.e., $\lambda(\tau_m)$, with $m \in [1..j]$, is of the form $i?a$). Observe that at the configurations of ρ where the transitions τ_m, are executed, the stack of thread i is empty and furthermore the thread i does not hold any locks.

We also decompose β uniquely as $\beta_1\tau'_1\beta_2\tau'_2 \dots \beta_k\tau'_k\gamma$, where τ'_m, $1 \leq m \leq k \in \delta_i$ are the transitions in β which acquire a lock that is not subsequently released. That is, τ'_m is a lock transition on some lock l_m and it is the last transition involving lock l_m in β. We observe that in the configurations of ρ where the transitions τ'_m, $1 \leq m \leq k$ are executed, the set of locks held by thread i must be exactly the set $\{l_r \mid r < m\}$. (Clearly these locks are held, by the definition of τ'_r's. No other lock is held as it would violate the nested locking assumption.)

Thus $\sigma_i = (\tau_1\alpha_1)(\tau_2\alpha_2) \dots (\tau_j\beta_1)(\tau'_1\beta_2) \dots (\tau'_k\gamma)$. We refer to this as the phase decomposition of σ w.r.t. thread i. We further refer to $(\tau_1\alpha_1), (\tau_2\alpha_2), \dots (\tau_{j-1}\alpha_{j-1})$ as *task* phases, $(\tau_j\beta_1)$ as the *boundary* phase and $(\tau'_1\beta_2), (\tau'_2\beta_3) \dots (\tau'_k\gamma)$ as *lock-holding* phases of thread i. Note that there may be no lock-holding phases or even task phases for some threads. A phase of σ is a phase of some thread in σ (and similarly with task, boundary and lock phases).

Given a phase γ in σ we define its index in σ to be the position of the first transition of γ in the sequence σ and write $\mathbf{i}(\gamma)$ to denote this number. Clearly $\mathbf{i}$ defines a linear order on the phases of γ and this allows us to list these phases as $\gamma_1, \gamma_2, \dots, \gamma_m$ where $\mathbf{i}(\gamma_i) < \mathbf{i}(\gamma_j)$ whenever $i < j$. We call this the occurrence ordering of phases. The following Lemma establishes the serialization result we desire.

▶ **Lemma 4.** *Let $\rho = (\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l}) \xrightarrow{\sigma}_A (\boldsymbol{c'}, \boldsymbol{m'}, \boldsymbol{l'})$ be a run from the initial configuration. Let $\gamma_1, \gamma_2, \dots \gamma_m$ be the listing of the all phases of σ in the occurrence order. Then, there is a run $\rho' = (\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l}) \xrightarrow{\gamma_1\gamma_2\dots\gamma_m}_A (\boldsymbol{c'}, \boldsymbol{m'}, \boldsymbol{l'})$. That is, we can execute the phases of σ atomically (without interleaving) and in their occurrence order.*

5.2 From N-MPDS to 1-MPDS

We are now ready to use the serialization lemma to show that reachability for N-MPDS can be reduced to reachability in an 1-MPDS (or simply MPDS). We will outline the main ideas behind our construction, refining them as we go along, before proceeding to the details.

The multi-set pushdown system that we construct has to maintain the local state of each of the threads. Similarly, it also needs information on the set of locks held by each thread. However notice that by assumption, all valid executions of the system only admits well-nested locks. Hence it is enough to store the first taken lock (that will be released) to know if there are pending locks or if no locks are taken. Thus, each state of the MPDS is of the form $(\boldsymbol{q}, \boldsymbol{l}, z)$, where $\boldsymbol{q}(i)$ denotes the state of thread i and $\boldsymbol{l}(i)$ stores the first lock that was taken (and will be released) by the thread i or is empty if none is taken. The component z holds additional information, such as the identity of the thread being executed (more details will be provided as we go along). We observe that while the N-MPDS has a multi-set per thread, we are only allowed to use a single multi-set in the constructed MPDS. We resolve this by the indexation of each task in the multi-set of the MPDS with the id of the thread it belongs to (i.e., the multi-set alphabet of the MPDS is set to be $\Sigma \times \{1, 2, \dots, N\} \times Y$ where a value

(a, i, y) stands for a value a in the multi-set belonging to thread i, with some additional information y that will be explained later).

A configuration of the MPDS is of the form $(((\boldsymbol{q}, \boldsymbol{l}, z), \alpha), \boldsymbol{M})$ and as explained above it encodes the local states and lock state of a configuration of the N-MPDS. Further, the value $\sum_{y \in Y} \boldsymbol{M}(a, i, y)$ represents $\boldsymbol{m}(i)(a)$, the number of tasks of type a in the multi-set of thread i. Thus, the only unrepresented part of the configuration of the N-MPDS is its collection of stacks, one per thread. Since we have only one stack in the MPDS, we have to manage the simulation of this collection of stacks using this single stack. We shall return to this soon.

Omitting, for the moment, the effect of transitions on the stack (and unexplained parts z and y), the definition of the transitions is easy to see: Simulating a transition of thread i that takes the lock l, involves storing it in the state if it is the very first lock taken by the process (i.e. change the local state component). Unlocking of that transition is handled similarly. Observe that there is no need to keep track of all the taken locks per thread due to the serialization (since the set of available locks at the beginning and end of each phase is already known). To simulate a move posting task a on to thread j, the MPDS posts the value (a, j, y) to its multi-set. Scheduling a task a on thread j corresponds to removing a message of the form (a, j, y) from the multi-set and so on.

We now turn our attention to dealing with the multiple stacks of the N-MPDS. As an immediate consequence of our serialization Lemma, it suffices to simulate the executions of the N-MPDS along executions where each phase is executed atomically. Observe that task phases of any thread begins and ends with the empty stack (in that thread) and empty set of locks. Thus, our MPDS can use its single stack to simulate any sequence of task phases (one after the other). However, the boundary phase does not necessarily end with an empty stack and the lock-holding phases need not begin or end with an empty stack. Consider a sequence of phases of the form $\beta_1\beta_2\beta_3\beta_4\beta_5\beta_6$ where β_1 is a boundary phase of thread i, β_3 and β_6 are lock-holding phases of thread i, β_2 is the boundary phase of task j, β_5 is a lock-holding phase of thread j and β_4 is a phase (of any type) of thread k. The contents of the stack of the thread i must be preserved from the completion of β_1 to the beginning of β_3 and then on to the beginning of β_6, while that of thread j is to be preserved from the completion β_2 to the beginning of β_5. We also need to execute the phase β_4 in between. There is no direct way to achieve this using a single stack. However, we can exploit the fact that there are only a bounded number of boundary and lock-holding phases (actually their number is bounded by the number of threads and the number of locks in the system respectively) to show that we may execute them out of order in a consistent manner using a single stack.

We illustrate the issues involved using the example from the previous paragraph. Suppose β_3, β_5 and β_6 take the locks l_3, l_5 and l_6 respectively and these are never released subsequently. In order to avoid storing the stack contents at the end of a phase (which we cannot), we would like to execute all the lock-holding phases of a task atomically. While it may seem tempting to run β_4 (say if it is a task phase) first, this may not be possible as this task may be created during the execution of β_1 or β_2 or β_3. This kind of causality does not pose a problem as it is handled by the multi-sets. However, we cannot postpone β_4 to completion ahead of the other phases, as β_4 may require the use of lock l_3 which is no longer available after the execution of β_3.

The key idea, is to divide the entire global execution into segments. An initial segment where only task phases are executed (where all locks are available for any phase that is executed) followed by the segment that involves a boundary phase together with task phases, the segment that involves another boundary phase or from the first lock-taking phase to the second (where exactly one lock is unavailable for any phase) together with task phases, the

segment that corresponds to a boundary phase or from the second locking taking phase to the third (where exactly two locks are unavailable for any phase) together with task phases and so on. The number of segments into which any execution breaks up is bounded by the number of locks.

Our MPDS guesses a priori a sequence $w \in ((\mathcal{L} \cup \{\emptyset\}) \times [1..N])^*$ called the guiding sequence (which represents the partition of the global execution into segments). A tuple of the form $(\emptyset, i)$ in the guiding sequence indicates the position of the boundary segment of the thread i. Similarly a tuple of the form (l, i) indicates the positions of the lock-taking segment where the thread i acquires the lock l and never releases it. We call a guiding sequence *valid-guiding-sequence* if all locks occur at-most once in the sequence and if for each thread, the boundary segment precedes the lock-taking segment. Formally, we call a sequence $w = (l_1, i_1) \dots (l_k, i_k) \in ((\mathcal{L} \cup \{\emptyset\}) \times [1..N])^*$ a valid guiding sequence if it has the following properties: 1) For all $i \in [1..N]$, we have $w \downarrow_{(\mathcal{L} \cup \{\emptyset\}) \times \{i\}} \in (\emptyset, i).(\mathcal{L} \times \{i\})^*$, 2) Let $w \downarrow_{\mathcal{L} \times [1..N]} = (s_1, j_1) \cdots (s_m, j_m)$. We have $s_u \neq s_v$ for all $u \neq v$. (i.e. Locks occur only once)

We are only interested in valid guiding sequences and hence, here onwards we will refer to them simply as guiding sequences. We start by guessing a valid sequence and then construct its corresponding MPDS. (Observe that the number of valid sequences is at most exponential in the size of the N-MPDS). As part of the component z of that MPDS, we store the segment number (i.e., the position into the guiding sequence). Initially, the sequence number is initialized to 0 indicating that no part of the guiding sequence is processed. The MPDS begins the simulation with a sequence of task phases that constitute segment 0. When chooses to initiate segment 1. If segment 1 is of the form $(\emptyset, i_1)$ then it picks the thread i_1, executes its boundary phase followed by the execution of all the subsequent phases of thread i_1, each of which must necessarily initiate a segment.

Suppose these segments belonging to the thread i_1 are $1 < j_1 < j_2 \dots < j_m$. Let the entry in the guiding sequence at position j_r be (l_{j_r}, i_1). Then, the MPDS verifies that, while executing the phase of thread i_1 initiating segments j_r, $1 \leq r \leq m$, that the lock l_{j_r} is taken by the first transition and never released, no lock from $\{l | \exists j < j_r, p \in [1..N] : w[j] = (l, p)\}$ is taken during that phase and that any other lock taken is released within the phase. Thus, we execute not just the boundary phase of i_1 but all the subsequent phases of this thread as well. Further, while doing this we ensure that the phases executed out of turn use only the appropriate set of locks available to them if they were executed in the right order. We also ensure that this thread is never scheduled again and this can be taken care of by looking at the current segment number stored in the state. We do not schedule a thread $p \in [1..N]$ if the current segment in the state is j and $w[k] = (\emptyset, p)$ for some $k < j$. We also ensure that the desired final state is reached during such an execution. After this simulation of thread i_1 we increase the segment number to 2 and proceed (if it does not belong to the thread i_1). We do a similar contiguous execution, of the boundary phase and all the subsequent phases of a thread, every time we decide to switch segments and encounter a segment of the form $(\emptyset, i)$. We skip any segment that has been already processed. Such executions ensure the correct usage of the locks using the lock-thread pair sequence. However, we have lost the causal relationship between task creation and execution. For instance, while completing the full execution of i_1 we may create tasks during the execution of segment j_4, but then we can schedule such a task in an earlier segment (with a incorrect lock environment to make it worse). But this problem is solved as follows: we tag the tasks inserted into the multi-set not only with the thread identity but also the segment in which the task is to be scheduled (explaining the third component y in the alphabet of the multi-set alphabet). This target segment of each task is chosen non-deterministically, with a number greater than or equal to

that of the posting transition, tagged with this value and then inserted into the multi-set. Then, a task is picked up from the multi-set only if its segment number tag matches the current segment number. This ensures that tasks are scheduled after their creation (and executed with the right set of locks).

Thus overall the execution of the MPDS may be summarized as the following:

The initialization part of our MPDS initializes the multi-set to hold the initial multi-set symbol for each process. The segment value in the initial state is set to 0. At the beginning of each step the stack is empty. The MPDS can increment the stored segment number in non-deterministic manner. We will refer to the current segment number stored in the state by j. At the beginning of each simulation of a segment (except when $j = 0$), the MPDS removed a task of the form (a, i, j) and executed it as a boundary phase while making sure that the j-th entry in the guiding sequence is of the form $(\emptyset, i)$. Suppose that the sequence of segments corresponding to thread j in the guiding sequence is $(\emptyset, i).(l_1, i) \dots (l_m, i)$, then the boundary phase is first simulated. Subsequently the first locking phase is executed. Then, the MPDS continues the process till the simulation of the last locking phase while making sure that the execution reaches the desired final state of the thread i. On completion, the stack is emptied and we are now allowed to simulate task phases of the from (a, k, j). In this case the task (a, k, j) is executed till the stack is empty again.

During the simulation of a boundary/task phase, the state is tagged with the information on whether the set of locks temporary taken (i.e., will be released) is empty or not. Initially, the state is marked with $\emptyset$ to indicate that there are no locks temporarily held. As soon as a lock is taken (and will be released), the identity of this lock is recorded in the state in place of the empty set $\emptyset$. If the state is already tagged with a lock, subsequent held locks are not stored. When the lock l is released from a state which is tagged with a lock l, the value is reset to $\emptyset$ indicating no more locks are temporarily held at that point.

Observe, that we construct one such automaton per guiding sequence. We run them one after the other to solve the reachability problem for the N-MPDS.

This construction is exponentially large even for a given guiding sequence. We now refine this construction further, making it polynomial — the key idea is to transfer large sections of the control states to the multi-set. This will lead to an automaton whose state space and multi-set alphabet are both polynomial in the size of the original system.

The exponential blow in the state space arises due to the product of the local state spaces and locks that are maintained in the state. During start of a task or a boundary phase, the set of locks held can easily be determined from the guiding sequence. Hence we only need record for the currently active thread, whether the locks held are empty or the identity of the first lock taken for that thread. The key step to eliminate blow up due to product of local state space is to expand the multi-set alphabet to include $\bigcup_{1 \le i \le N} Q_i \times \{i\}$. That is, we keep the current state of each thread, other than the currently executing thread, in the multi-set (transferring the state to the multi-set while switching threads).

With this change we obtain a polynomial sized MPDS, that verifies reachability via runs obeying the given guiding sequence. This results in a Expspace decision procedure per guiding sequence. Since the number of guiding sequences is only exponential, we can run through all candidate sequences and obtain a Expspace procedure overall.

6 Stateless Well-Nested N-MPDS under the Task Locking Policy

Typically asynchronous concurrent software has threads that wait for a task in a while loop. On receipt of a task, they execute it and go back to a waiting state. Motivated by this, we propose a model in which each thread can remove tasks from its multiset only in a particular

state. Formally, let $A = (\Sigma, \mathcal{P}, \mathcal{L})$ be an N-MPDS (as defined in Section 3) and $\boldsymbol{s}$ be a state function that assigns to each index $i \in [1..N]$ a state from Q_i (i.e. $\boldsymbol{s}(i) = q_i$ where q_i is a state of the thread i). We say that A is $\boldsymbol{s}$-stateless if $\delta_i \cap \bigcup_{a\in\Sigma}(Q_i \setminus \{\boldsymbol{s}[i]\} \times \Gamma_i^* \times \{i?a\} \times \Gamma_i^* \times Q_i) = \emptyset$. We say that an N-MPDS A is *stateless* if there is a state function $\boldsymbol{s}$ such that A is $\boldsymbol{s}$-stateless.

In this section, we show that the reachability problem for stateless well-nested N-MPDS under the task locking policy is NP-COMPLETE.

▶ **Theorem 5.** *The reachability problem for stateless well-nested MPDSs under the task-locking policy is* NP-COMPLETE.

The lower bound is proved by reduction from the satisfiability problem of a given 3-SAT formula. In the following, we sketch the proof that establishes the upper-bound. To that aim, let us fix a well-nested N-MPDS $A = (\Sigma, \mathcal{P}, \mathcal{L})$ (as defined in Section 3) under the task-locking policy. Let us assume that the N-MPDS is $\boldsymbol{s}$-stateless for some state function $\boldsymbol{s}$. Further, we assume w.l.o.g. that the only enabled move from the initial state of any thread is a transition that removes a task from the multi-set. In the following, we will show that there is an NP algorithm to solve the reachability problem for A. Towards this, we will show that for any execution of the 1-MPDS, there is a shorter execution that involves only a polynomial number of tasks and reaches the same stack and lock configurations.

▶ **Lemma 6.** *Let $\rho = (\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l}) \xrightarrow{\sigma}_A (\boldsymbol{c'}, \boldsymbol{m'}, \boldsymbol{l'})$ be an execution from the initial configuration. Then, there is another execution $\rho' = (\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l}) \xrightarrow{\sigma'} (\boldsymbol{c'}, \boldsymbol{m''}, \boldsymbol{l'})$ such that σ' can be decomposed into phases as $\sigma' = \gamma_1 \cdots \gamma_m$ where $m = \mathcal{O}(N \times ((N + |\mathcal{L}|) \times |\Sigma|) + N + |\mathcal{L}|)$.*

We now define K-bounded reachability for an MPDS. Given an MPDS $A = (\Sigma, (Q, \Gamma, \mathcal{O}, \delta, s_1, \alpha_1))$ and a state $q \in Q$, the K-bounded reachability asks if there is an execution $(s_1, w_1, \boldsymbol{M}_1) \xrightarrow{\sigma}_A (q, w, \boldsymbol{M})$, where $(s_1, w_1, \boldsymbol{M}_1)$ is the initial configuration, for some w and $\boldsymbol{M}$ such that $|\sigma \downarrow_{\{1!1(a), 1?a | a\in\Sigma\}}| \leq K$ (i.e., the total number of created and removed tasks is bounded by K). From lemma 6 and the construction in previous section, we get:

▶ **Corollary 7.** *The reachability problem for stateless well-nested N-MPDSs under the task locking policy can be polynomially reduced to the K-bounded reachability for MPDSs, where K is polynomial in the size of the given N-MPDS.*

Proof. The proof of the above corollary uses the same reduction as the one used in Section 5.2. In that construction, the number of operations that remove tasks from the multi-sets of the N-MPDS is the same as the number of operations that remove tasks in the constructed MPDS. By Lemma 6, this number of removed tasks is bounded by a constant K' which is polynomial in the size of the N-MPDS. Furthermore, for any transition (say $(p, \alpha, 1!1(a), \beta, p')$) of the constructed MPDS that creates a task we add to the MPDS another *copy* of the transition that omits the creation of the task (namely a transition of the form $(p, \alpha, \epsilon, \beta, p')$). Since the total number of removed tasks bounds the total number of the tasks that need to be created we get our bound K which can be set to $2K'$. ◀

We will now prove that the K-bounded reachability for MPDSs is in NP. This automatically gives us an NP algorithm for the reachability problem.

▶ **Lemma 8.** *Given an MPDS A and a state q, there is an non-deterministic polynomial time algorithm that decides whether q is K-bounded reachable in A.*

Proof. The NP algorithm starts by guessing the sequence of multi-set operations that create or remove tasks. Observe that the size of such a sequence is bounded by K. Then, the algorithm checks if the guessed sequence is *valid* (i.e., for every prefix of the guessed sequence,

the number of created tasks of a type a is larger that the number of removed tasks of type a). This can be easily checked in polynomial time. Now, if the guessed sequence is valid, the algorithm checks if this sequence can lead to an execution of the MPDS that reaches the state q. This is done by seeing the MPDS $A = (\Sigma, (Q, \Gamma, \mathcal{O}, \delta, s_1, \alpha_1))$ as a pushdown automaton $P = (Q, \Gamma, \mathcal{O}, \delta, s_1, \alpha_1)$ over the alphabet $\{1!1(a), 1?a \mid a \in \Sigma\}$. Finally, checking whether the guessed sequence can lead to an execution of the MPDS that reaches q can be reduced to checking if the guessed sequence can be accepted by the pushdown automaton. ◀

References

1. Mohamed Faouzi Atig, Ahmed Bouajjani, and Shaz Qadeer. Context-bounded analysis for concurrent programs with dynamic creation of threads. *Logical Methods in Computer Science*, 7(4), 2011. doi:10.2168/LMCS-7(4:4)2011.
2. Mohamed Faouzi Atig, Ahmed Bouajjani, and Tayssir Touili. Analyzing asynchronous programs with preemption. In Ramesh Hariharan, Madhavan Mukund, and V. Vinay, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2008, December 9-11, 2008, Bangalore, India*, volume 2 of *LIPIcs*, pages 37–48. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2008. doi:10.4230/LIPIcs.FSTTCS.2008.1739.
3. Ahmed Bouajjani, Markus Müller-Olm, and Tayssir Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In Martín Abadi and Luca de Alfaro, editors, *CONCUR 2005 - Concurrency Theory, 16th International Conference, CONCUR 2005, San Francisco, CA, USA, August 23-26, 2005, Proceedings*, volume 3653 of *Lecture Notes in Computer Science*, pages 473–487. Springer, 2005. doi:10.1007/11539452_36.
4. Michael Emmi, Shaz Qadeer, and Zvonimir Rakamaric. Delay-bounded scheduling. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 411–422. ACM, 2011. doi:10.1145/1926385.1926432.
5. Pierre Ganty and Rupak Majumdar. Algorithmic verification of asynchronous programs. *ACM Trans. Program. Lang. Syst.*, 34(1):6:1–6:48, 2012. doi:10.1145/2160910.2160915.
6. Thomas Martin Gawlitza, Peter Lammich, Markus Müller-Olm, Helmut Seidl, and Alexander Wenner. Join-lock-sensitive forward reachability analysis for concurrent programs with dynamic process creation. In Ranjit Jhala and David A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, volume 6538 of *Lecture Notes in Computer Science*, pages 199–213. Springer, 2011. doi:10.1007/978-3-642-18275-4_15.
7. Ranjit Jhala and Rupak Majumdar. Interprocedural analysis of asynchronous programs. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 339–350. ACM, 2007. doi:10.1145/1190216.1190266.
8. Vineet Kahlon, Franjo Ivancic, and Aarti Gupta. Reasoning about threads communicating via locks. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 505–518. Springer, 2005. doi:10.1007/11513988_49.
9. Peter Lammich and Markus Müller-Olm. Conflict analysis of programs with procedures, dynamic thread creation, and monitors. In María Alpuente and Germán Vidal, editors, *Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 16-18, 2008. Proceedings*, volume 5079 of *Lecture Notes in Computer Science*, pages 205–220. Springer, 2008. doi:10.1007/978-3-540-69166-2_14.

10 Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In Nicolas Halbwachs and Lenore D. Zuck, editors, *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. doi:10.1007/978-3-540-31980-1_7.

11 G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.*, 22(2):416–430, 2000. doi:10.1145/349214.349241.

12 Koushik Sen and Mahesh Viswanathan. Model checking multithreaded programs with asynchronous atomic methods. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 300–314. Springer, 2006. doi:10.1007/11817963_29.