Verification of Asynchronous Programs with Nested Locks

M. F. Atig[1], A. Bouajjani[2], K. Narayan Kumar[3], and P. 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, 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 threads running in parallel. Each of the thread is equipped with a multi-set. The threads can create tasks and post them onto multi-sets of other threads 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 of 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.

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.

Asynchronous execution by dynamically created concurrent tasks leads to extremely intricate and unpredictable behaviours, making very hard reasoning about asynchronous programs. Therefore, developing automated techniques for the verification of asynchronous programs is an important and challenging research topic. In the case of single-thread asynchronous programs, it has been shown that the reachability problem is decidable and EXPSPACE-complete (assuming that the data domain is finite), being as hard as the coverability problem in Petri nets [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 inter-leavings of concurrent actions. Therefore, assuming that locks are the only shared

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

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 its 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 decidable and EXPSPACE-complete. 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 NP-complete. 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 decidable and 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

Words. 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 $w = a_1 a_2 \ldots a_n \in \Sigma^*$, we let $|w| = n$ to denote the length of w, $w[i]$ to denote the i^{th} letter a_i and $w[i,j]$ to denote the subword $a_i \ldots a_j$. Given a word $w \in \Sigma$ and $\Sigma' \subseteq \Sigma$, we let $w \downarrow_{\Sigma'}$ to denote the projection of w onto Σ' .

Multi-sets. Let Σ be a finite alphabet. A multi-set over Σ is a function $M : \Sigma \mapsto \mathbb{N}$. We denote by $M[\Sigma]$ the collection 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.

Pushdown Automata. A *pushdown automata (PDA)* is a tuple $\mathbf{P} = (Q, \Gamma, \Sigma, \delta, s_0, \alpha_0)$ where Q is a 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 bottom of stack element $\perp$. 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) $\beta \in \Gamma_\epsilon \times \{\perp\}$ when $\alpha = \perp$. We use $\mathbf{Src}(\tau) = q$ to refer to the head state of the transition, $\mathbf{Dest}(\tau) = q'$ to refer the tail state, and $\lambda(\tau)$ to denote the label a.

A configuration of a PDA $\mathbf{P} = (Q, \Gamma, \Sigma, \delta, s_0, \alpha_0)$ is a pair (q, γ) with $q \in Q$ and $\gamma \in (\Gamma \setminus \{\perp\})^* \perp$. Given a configuration $c = (q, \gamma)$, we will use $\mathbf{Stt}(c)$ to refer to the state q and $\mathbf{Stk}(c)$ to refer to the stack component of the configuration 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 $\tau = (q, \alpha, a, \beta, q')$ with $\alpha, \beta \in \Gamma \setminus \{\perp\}$. This transition corresponds to the pop of the symbol α from the stack and the push of the word β into the stack.

We often omit the reference to $\mathbf{P}$ and write $\xrightarrow{\tau}$ when $\mathbf{P}$ is clear from the context. We will also sometimes omit τ and simply write $\rightarrow$ when the reference to τ is unimportant. We write $(q, \gamma) \xrightarrow{\sigma} (q', \gamma')$ for $\sigma = \tau_1 \ldots \tau_n \in \delta^*$ to mean that there is a sequence of transitions of the form $(q, \gamma) = (q_0, \gamma_0) \xrightarrow{\tau_1} (q_1, \gamma_1) \xrightarrow{\tau_2} \ldots \xrightarrow{\tau_{n-1}} (q_{n-1}, \gamma_{n-1}) \xrightarrow{\tau_n} (q_n, \gamma_n) = (q', \gamma')$. 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 $c_1 \xrightarrow{\tau_1 \tau_2 \cdots \tau_n} c_2$ and $w = \lambda(\tau_1)\lambda(\tau_2) \cdots \lambda(\tau_3)$. 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 one 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 $\mathcal{O}_i = \{i!j(a), i?a \mid a \in \Sigma, 1 \leq j \leq N\} \cup \{\text{lck}_i(l), \text{rel}_i(l) \mid l \in \mathcal{L}\}$. 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, $\text{lck}_i(l)/\text{rel}_i(l)$ 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) $\delta_i \cap (Q_i \times (\Gamma_i \setminus \{\perp\}) \times \{i?a \mid a \in \Sigma\} \times \Gamma_i^* \times Q_i) = \emptyset$ 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 $(q, \perp, b, \beta\perp, q')$ 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 a N-MPDS A is a triple of functions $(\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l})$ where, for each $1 \leq i \leq N$, $\boldsymbol{c}(i)$ 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 $\boldsymbol{c_0}(i) = (s_i, \perp), \boldsymbol{m_0}(i) = \lfloor \alpha_i \rfloor$ 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 an 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 $\boldsymbol{l}(k) = \boldsymbol{l'}(k)$ for all k. (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 $\boldsymbol{l}(k) = \boldsymbol{l'}(k)$ for all k. (The thread i picks a task of type a from its multiset and adds it to its stack for execution. Recall that removing a task from the multiset is only possible when the stack of the thread is empty.)
- $\lambda(\tau) = \text{lck}_i(l)$: $l \notin \bigcup_{1\leq k\leq N} \boldsymbol{l}(k)$, $\boldsymbol{l'}(i) = \boldsymbol{l}(i) \cup \{l\}$, $\boldsymbol{l}(i) = \boldsymbol{l'}(i)$, $\boldsymbol{l}(k) = \boldsymbol{l'}(k)$ 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 by another thread.)
- $\lambda(\tau) = \text{rel}_i(l)$: $l \in \boldsymbol{l}(i)$, $\boldsymbol{l'}(i) = \boldsymbol{l}(i) \setminus \{l\}$, $\boldsymbol{l}(i) = \boldsymbol{l'}(i)$, $\boldsymbol{l}(k) = \boldsymbol{l'}(k)$ 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 $\mathbf{Stt}(\boldsymbol{c}(i)) = \boldsymbol{r}(i)$.

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 an 2-MPDS, without any locks, by using the multi-sets as a synchronizing mechanism. Given two pushdown automata over an alphabet Σ, we construct an 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-MPS 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. In our setting 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 $\lambda(\tau_i) = \mathrm{lck}_k(l)$, $\lambda(\tau_j) = \mathrm{rel}_k(l)$ and $\forall r \in \{i+1, \ldots, j-1\}, \lambda(\tau_r) \notin \bigcup_{1 \leq m \leq N} \{\mathrm{lck}_m(l), \mathrm{rel}_m(l)\}$. 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, a result of Kahlon et al. [8] shows that the reachability problem restricted nested locking is decidable for networks of pushdown automata with locks. However, as we shall see in the next section, 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 assuming that the locks are acquired and released following the 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 some alphabet Σ. We construct a 4-MPDS A that simulates joint executions of the two pushdown automata. WLOG, we will assume that there are no ϵ moves in P^1 and P^2 (note that 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. As a matter of 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.

The details of the construction and its correctness proof can be found in Appendix A. ◀

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 require 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 $(\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l}) \xrightarrow{\tau_1}_A (\boldsymbol{c_1}, \boldsymbol{m_1}, \boldsymbol{l_1}) \xrightarrow{\tau_2}_A (\boldsymbol{c_2}, \boldsymbol{m_2}, \boldsymbol{l_2}) \ldots (\boldsymbol{c_{n-1}}, \boldsymbol{m_{n-1}}, \boldsymbol{l_{n-1}}) \xrightarrow{\tau_n}_A (\boldsymbol{c'}, \boldsymbol{m'}, \boldsymbol{l'})$ 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. (Recall again that $(\boldsymbol{c_0}, \boldsymbol{m_0}, \boldsymbol{l_0})$ is the initial configuration.)

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 transformed into this form by introducing a new set of states and new initial task symbols. Each thread from this new initial state will begin its execution by removing a task from its multi-set and then moving to its original starting configuration.

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: $\tau_1\alpha_2\tau_2 \ldots \alpha_j\tau_j\beta$ where, $\tau_1, \ldots, \tau_j$ 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 $\beta_1\tau'_1\beta_2\tau'_2 \ldots \beta_k\tau'_k\gamma$, 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 $\sigma_i = (\tau_1\alpha_1)(\tau_2\alpha_2)\ldots(\tau_j\beta_1)(\tau'_1\beta_2)\ldots(\tau'_k\gamma)$. We refer to this as the phase decomposition of σ w.r.t thread i. We further refer to $(\tau_1\alpha_1),(\tau_2\alpha_2),\ldots(\tau_{j-1}\alpha_{j-1})$ as *task* phases of thread i, $(\tau_j\beta_1)$ as the *boundary* phase and $(\tau'_1\beta_2),(\tau'_2\beta_3)\ldots(\tau'_k\gamma)$ as *lock-holding* phases of thread i. Note that there may be no lock-holding phases or even no task phases for some threads. A phase of σ is a phase of some thread in σ (and similarly with task phases, boundary phases 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 $\gamma_1, \gamma_2, \ldots, \gamma_m$ 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 $\gamma_1, \gamma_2, \ldots \gamma_m$ be the listing of the all phases of σ in the occurrence order. Then, there is a run $\rho' = (\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l}) \xrightarrow{\gamma_1\gamma_2\cdots\gamma_m}_A (\boldsymbol{c'}, \boldsymbol{m'}, \boldsymbol{l'})$. 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, where 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 $\Sigma \times \{1, 2, \ldots, N\} \times Y$ 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 codes 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 task 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 begining 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. (Observe that we treat the initial phase as a task phases.)

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 and the boundary segment precedes the lock-taking segment. Formally, we call a sequence $w = (l_1, i_1) \ldots (l_k, i_k) \in ((\mathcal{L} \cup \{\emptyset\}) \times [1..N])^*$ 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 sequence and hence, here onwards we will refer to them simply as guiding sequence. We start by guessing a valid sequence and then construct its corresponding MPDS. (Observe that the number of valid sequences is at most exponetial in the size of the N-MPDS). As part of the component z of that MPDS, we store the segment number (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 phase 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 $1 < j_1 < j_2 \ldots < j_m$. Let the entry in the guiding sequence at position j_r is (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 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 $(\emptyset, i).(l_1, i)\ldots(l_m, i)$, 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 temporary 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 $\bigcup_{1 \leq i \leq N} Q_i \times \{i\}$. 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. A detailed construction with proof is in Appendix B.

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

Typically asynchronous concurrent software have threads that wait for a task in a while loop. On receipt of a task, they execute the task 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 can be decided in NP. Furthermore, this upper-bound is tight since we also prove that this problem is NP-hard.

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

The rest of this section is devoted to the proof of the above theorem.

▶ **Lemma 6.** *stateless well-nested MPDSs under the task-locking policy is* NP-hard.

Upper-bound: 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 7.** *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 introduce the notion of 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 $(s_1, w_1, \boldsymbol{M}_1) \xrightarrow{\sigma}_A (q, w, \boldsymbol{M})$, with $(s_1, w_1, \boldsymbol{M}_1)$ 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 the above lemma and the construction in previous section, we get the following corollary:

▶ **Corollary 8.** *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 task in the constructed MPDS. By Lemma 7, 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 task bounds the total number of the tasks that need to be create 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 of the stateless well-nested N-MPDSs under the task locking policy.

▶ **Lemma 9.** *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 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). Observe that the validity of a sequence 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 the state q can be reduced to checking if the guessed sequence can be accepted by the pushdown automaton (which again can be performed in polynomial time). ◀

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. URL: `https://doi.org/10.2168/LMCS-7(4:4)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. URL: `https://doi.org/10.4230/LIPIcs.FSTTCS.2008.1739`, `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. URL: `https://doi.org/10.1007/11539452_36`, `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. URL: `http://doi.acm.org/10.1145/1926385.1926432`, `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. URL: `http://doi.acm.org/10.1145/2160910.2160915`, `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. URL: `https://doi.org/10.1007/978-3-642-18275-4_15`, `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. URL: `http://doi.acm.org/10.1145/1190216.1190266`, `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. URL: `https://doi.org/10.1007/11513988_49`, `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. URL: `https://doi.org/10.1007/978-3-540-69166-2_14`, `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. URL: `https://doi.org/10.1007/978-3-540-31980-1_7`, `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. URL: `http://doi.acm.org/10.1145/349214.349241`, `doi:10.1145/349214.349241`.

12 Koushik Sen and Mahesh Viswanathan. Model checking multithreaded programs with asynchronous atomic methods. In *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, pages 300–314, 2006. URL: `https://doi.org/10.1007/11817963_29`, `doi:10.1007/11817963_29`.

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

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

Proof. In the following, we will give the formal construction and the correctness proof of Theorem 2. To simplify the presentation, we will use $\text{Test}(l)$ for checking if the lock l is free. Such an operation can be easily performed by holding and releasing the lock immediately. We first fix the pushdown automata $P^1 = (Q_1, \Gamma_1, \Sigma, \delta_1, s_1, \alpha_1)$ and $P^2 = (Q_2, \Gamma_2, \Sigma, \delta_2, s_2, \alpha_2)$ over the alphabet Σ. We assume w.l.o.g. that the pushdown automata P^1 and P^2 do not contain any transition rule of the form $(q, \perp, c, \beta, q')$, we will also assume that f_1, f_2 are the accepting states of the respective pushdown automata. We will further WLOG assume that these states can only be reached on stack being empty. The required 4-MPDS $A = (\Sigma \cup \{a, b, r, l\}, \{P_1, P_2, P_3, P_4\}, \{l_1, l_2\})$ is constructed as follows.

1. $P_1 = (S_1, \Gamma_1 \cup \{b\}, \mathcal{O}_1, \Delta_1, t^1, \alpha_1)$ where: (1) $S_1 = Q_1 \cup (\{t_1, t_2, t_3\} \times \delta_1) \cup \{t^1\}$ such that t_1, t_2, t_3 and t^1 are fresh states (i.e., $t^1, t_1, t_2, t_3 \notin Q_1$), and (2) the transition relation Δ_1 is defined as the smallest transition relation satisfying the following conditions:
   - **a.** We have $(t^1, \perp, 1?b, b \cdot \perp, t^1), (t^1, b, \epsilon, \alpha_1, s_1) \in \Delta_1$. The thread awaits for the task b and once this task is received, P_1 moves to the initial configuration of P^1 and the simulation can start.
   - **b.** For each $\tau = (q, \alpha, c, \beta, q') \in \delta_1$, we have $(\mathbf{Src}(\tau), \alpha, \text{lck}_1(l_2), \alpha, (t_1, \tau))$, $((t_1, \tau), \alpha, 1!3(c), \alpha, (t_2, \tau))$, $((t_2, \tau), \alpha, \text{rel}_1(l_2), \alpha, (t_3, \tau))$, $((t_3, \tau), \alpha, \text{Test}(l_1), \beta, \mathbf{Dest}(\tau))$ are in Δ_1. This corresponds to the fact that P_1 takes the lock l_2 while choosing to simulate the transition τ. Then, P_1 posts the task c to the thread P_3 and release the lock l_2. Finally, P_1 waits to take the lock l_1 while simulating the effect of the transition τ on the stack and released this lock immediately.
2. $P_2 = (S_2, \Gamma_2 \cup \{b\}, \mathcal{O}_2, \Delta_2, t^2, \alpha_2)$ where: (1) $S_2 = Q_2 \cup (\{t_1, t_2, t_3\} \times \delta_2) \cup \{t^2\}$ such that t_1, t_2, t_3 and t^2 are fresh states (i.e., $t^2, t_1, t_2, t_3 \notin Q_2$), and (2) the transition relation Δ_2 is defined as the smallest transition relation satisfying the following conditions:
   - **a.** We have $(t^2, \perp, 2?b, b \cdot \perp, t^2), (t^2, b, \epsilon, \alpha_2, s_2) \in \Delta_2$. The second threads awaits for the task b and once this task b is received, P_2 moves to the initial configuration of P^2 and the simulation can start.
   - **b.** For each $\tau = (q, \alpha, c, \beta, q') \in \delta_2$, we have $(\mathbf{Src}(\tau), \alpha, \text{lck}_2(l_2), \alpha, (t_1, \tau))$, $((t_1, \tau), \alpha, 2!3(c), \alpha, (t_2, \tau))$, $((t_2, \tau), \alpha, \text{rel}_2(l_2), \alpha, (t_3, \tau))$, $((t_3, \tau), \alpha, \text{Test}(l_1), \beta, \mathbf{Dest}(\tau))$ are in Δ_2. This corresponds to the fact that P_2 takes the lock l_2 while choosing to simulate the transition τ. Then, P_2 posts the task c to the thread P_3 and release the lock l_2. Finally, P_2 waits to take the lock l_1 while simulating the effect of the transition τ on the stack and released this lock immediately.
3. $P_3 = (S_3, \{\alpha\}, \mathcal{O}_3, \Delta_3, p_0, \alpha)$ is defined as follows:
   - **a.** $S_3 = \{p_0, \ldots, p_9\} \cup (\{p\} \times \Sigma)$ is the set of states.
   - **b.** The transition relation Δ_3 is defined as the smallest relation satisfying the following conditions:
     - **i.** $(p_0, \perp, \text{lck}_3(l_1), \perp, p_1), (p_1, \perp, 3!1(b), \perp, p_2), (p_2, \perp, 3!2(b), \perp, p_3)$ are in Δ_3. This corresponds to the initialisation phase where P_1 takes the lock l_1 and posts the task b to P_1 and P_2.
     - **ii.** For every $c \in \Sigma$, we have $(p_3, \perp, 3?c, \perp, (p, c))$ and $((p, c), \perp, 3?c, \perp, p_4) \in \Delta_3$. These transitions corresponds to checking if P_1, P_2 read the same letter.

iii. $(p_4, \perp, 3!4(l), \perp, p_5)$, $(p_5, \perp, 3?a, \perp, p_6)$ and $(p_6, \perp, \mathrm{rel}_3(l_1), \perp, p_7)$ are in Δ_3. This corresponds to asking P_4 to hold the lock l_2. On receiving an acknowledgement (i.e., the task a), P_3 releases the lock l_1.

iv. $(p_7, \perp, \mathrm{lck}_3(l_1), \perp, p_8)$, $(p_8, \perp, 3!4(r), \perp, p_9)$ and $(p_9, \perp, 3?a, \perp, p_3)$ are in Δ_3. This corresponds to finally locking l_1, asking P_4 to release l_2 and waiting for an acknowledgement from P_4.

4. $P_4 = (S_4, \alpha, \mathcal{O}_4, \Delta_4, p'_0, \alpha)$ is constructed as follows

a. $S_4 = \{p'_0, \dots, p'_4\}$ is the set of states.

b. The transition relation Δ_3 is defined as the smallest relation containing the following transitions: $(p'_0, \perp, 4?l, \perp, p'_1), (p'_1, \perp, \mathrm{lck}_4(l_2), \perp, p'_2), (p'_2, \perp, 4!3(a), \perp, p'_3), (p'_3, \perp, 4?r, \perp, p'_4), (p'_4, \perp, \mathrm{rel}_4(l_2), \perp, p'_5), (p'_5, \perp, 4!3(a), \perp, p'_0)$.

The required destination function $\boldsymbol{r}$ is given by $\boldsymbol{r}(1) = f_1, \boldsymbol{r}(2) = f_2, \boldsymbol{r}(3) = p_3, \boldsymbol{r}(4) = p'_0$. The correctness of the above construction follows from the following lemma. Given any function f, we will use the notation $f' = f[i \leftarrow k]$ to denote $f'(i) = k$ and f' retains the values of f otherwise. In the lemma below, we will also slightly abuse the notation of a move of a pushdown system. We will use $c \xrightarrow{a} c'$ to mean that there is a transition τ with $\lambda(\tau) = a$ and $c \xrightarrow{\tau} c'$.

▶ **Lemma 10.** *$(s_1, \perp) \xrightarrow{w}_{P_u^1} (q, \gamma_1)$ for some $q \in Q_1, \gamma_1 \in \Gamma_1^*$ and $(s_2, \perp) \xrightarrow{w}_{P_u^2} (p, \gamma_2)$ for some $p \in Q_2, \gamma_2 \in \Gamma_2^*$ iff $(\boldsymbol{c}, \boldsymbol{m_0}, \boldsymbol{l}) \rightarrow_A (\boldsymbol{c'}, \boldsymbol{m_0}, \boldsymbol{l})$, where $\boldsymbol{l} = \boldsymbol{l_0}[3 \leftarrow l_1]$, $\boldsymbol{c'}(1) = (q, \gamma_1)$, $\boldsymbol{c'}(2) = (p, \gamma_2)$, $\boldsymbol{c}(3) = \boldsymbol{c'}(3) = (p_3, \perp), \boldsymbol{c}(4) = \boldsymbol{c'}(4) = (p'_0, \perp), \boldsymbol{c}(1) = (s_1, \perp)$ and $\boldsymbol{c}(2) = (s_2, \perp)$.*

Proof of lemma-10. ($\Rightarrow$) We will prove this direction by inducting on the size of w. Suppose $w = \epsilon$, we have nothing to prove, hence we will assume $w = w'.a$ for some $a \in \Sigma$. Assume a joint run in P_u^1 and P_u^2 on w, of the form $(s_1, \perp) \xrightarrow{w'}_{P_u^1} (q', \gamma'_1) \xrightarrow{a}_{P_u^1} (q, \gamma_1)$ and $(s_2, \perp) \xrightarrow{w'}_{P_u^2} (p', \gamma'_2) \xrightarrow{a} (p, \gamma_2)$. Firstly by induction, we have a run of the form $(\boldsymbol{c}, \boldsymbol{m_0}, \boldsymbol{l}) \rightarrow_A (\boldsymbol{c''}, \boldsymbol{m_0}, \boldsymbol{l})$, where $\boldsymbol{c''}(1) = (q', \gamma'_1), \boldsymbol{c''}(2) = (p', \gamma'_2), \boldsymbol{c''}(3) = \boldsymbol{c'}(3), \boldsymbol{c''}(4) = \boldsymbol{c'}(4)$. There are also transitions $\tau'_1 \in \delta_1$ and $\tau'_2 \in \delta_2$ such that $\lambda(\tau'_1) = \lambda(\tau'_2) = a$ that was used in the moves $(q', \gamma'_1) \xrightarrow{a}_{P_u^1} (q, \gamma_1)$ and $(p', \gamma'_2) \xrightarrow{a} (p, \gamma_2)$. Now using these τ'_1, τ'_2 in lemma-11, we get $\tau_1 \cdots \tau_{22}$. It is easy to see that such a transition is executable from $(\boldsymbol{c''}, \boldsymbol{m_0}, \boldsymbol{l})$ primarily because $\boldsymbol{c''}(1) = (q', \gamma'_1), \boldsymbol{c''}(2) = (p', \gamma'_2)$. From this, we get the required run of the form $(\boldsymbol{c}, \boldsymbol{m_0}, \boldsymbol{l}) \rightarrow_A (\boldsymbol{c'}, \boldsymbol{m_0}, \boldsymbol{l})$

($\Leftarrow$)

Suppose there is a run of the form $(\boldsymbol{c}, \boldsymbol{m_0}, \boldsymbol{l}) \xrightarrow{\sigma}_A (\boldsymbol{c'}, \boldsymbol{m_0}, \boldsymbol{l})$, then by lemma-11, we have $(\boldsymbol{c}, \boldsymbol{m_0}, \boldsymbol{l}) \xrightarrow{\sigma'}_A (\boldsymbol{c'}, \boldsymbol{m_0}, \boldsymbol{l})$ with $\sigma' = \sigma_1 \dots \sigma_n$. Let the run be of the form $(\boldsymbol{c}, \boldsymbol{m_0}, \boldsymbol{l}) \xrightarrow{\sigma_1 \dots \sigma_{n-1}}_A (\boldsymbol{c''}, \boldsymbol{m_0}, \boldsymbol{l}) \xrightarrow{\sigma_n} (\boldsymbol{c'}, \boldsymbol{m_0}, \boldsymbol{l})$. By Induction, we have a run of the form $(s_1, \perp) \xrightarrow{w'}_{P_u^1} (q', \gamma'_1)$ and $(s_2, \perp) \xrightarrow{w'}_{P_u^2} (p', \gamma'_2)$, where $(q', \gamma'_1) = \boldsymbol{c''}(1)$ and $\boldsymbol{c''}(2) = (p', \gamma'_2)$. By lemma-11, σ_n is of the form $\sigma_n = \tau_1 \cdots \tau_{22}$. Further there are $\tau \in \delta_1$ and $\tau' \in \delta_2$ that appear in τ_1, τ_2 such that $\lambda(\tau_1) = \lambda(\tau_2) = a$ for some $a \in \Sigma$. It is easy to see that τ can be executed from (q', γ'_1) to obtain $\boldsymbol{c'}(1)$ and similarly τ' from (p', γ'_2). From this we obtain the required joint run $(s_1, \perp) \xrightarrow{w'}_{P_u^1} (q', \gamma'_1) \xrightarrow{a}_{P_u^1} (q, \gamma_1)$ and $(s_2, \perp) \xrightarrow{w'}_{P_u^2} (p', \gamma'_2) \xrightarrow{a} (p, \gamma_2)$, where $\boldsymbol{c'}(1) = (q, \gamma_1)$ and $\boldsymbol{c'}(2) = (p, \gamma_2)$.

◀

The following lemma states that if there is a run from a configuration of the form $(\boldsymbol{c}, \boldsymbol{m_0}, \boldsymbol{l})$ to a configuration of the form $(\boldsymbol{c'}, \boldsymbol{m_0}, \boldsymbol{l})$ where states of $\boldsymbol{c'}(1, 2)$ are from the given pushdown system i.e. not any intermediary states, then such a run can be split into many parts (depending on number of times process-1 and process-2 visits the states of PDS). Further the run in each such split can be replaced by a run of a particular form.

▶ **Lemma 11.** $(\boldsymbol{c}, \boldsymbol{m_0}, \boldsymbol{l}) \xrightarrow{\sigma} (\boldsymbol{c'}, \boldsymbol{m_0}, \boldsymbol{l})$ *(where the configurations are as in lemma-10) iff there is* σ' *such that* $(\boldsymbol{c}, \boldsymbol{m_0}, \boldsymbol{l}) \xrightarrow{\sigma'} (\boldsymbol{c'}, \boldsymbol{m_0}, \boldsymbol{l})$ *and* $\sigma' = \sigma_1 \ldots \sigma_n$ *(for some* $n \in \mathbb{N}$*), where for each* $i \in [1..n]$*,* $\sigma_i = \tau_1 \cdots \tau_{22}$ *where*

1. $\tau_1 = (q, \alpha, lck_1(l_2), \alpha, (t_1, \tau))$ *for some compatible* $\tau = (q, \alpha, c, \beta, q') \in \delta_1$*,* $\tau_2 = ((t_1, \tau), \alpha, 1!3(c), \alpha, (t_2, \tau))$*,* $\tau_3 = ((t_2, \tau), \alpha, rel_1(l_2), \alpha, (t_3, \tau))$*.*
2. $\tau_4 = (q, \alpha', lck_1(l_2), \alpha', (t_1, \tau'))$ *for some compatible* $\tau' = (q, \alpha', c, \beta', q') \in \delta_2$*,* $\tau_5 = ((t_1, \tau'), \alpha', 1!3(c), \alpha', (t_2, \tau'))$*,* $\tau_6 = ((t_2, \tau'), \alpha', rel_1(l_2), \alpha', (t_3, \tau'))$ *.*
3. $\tau_7 \cdot \tau_8 \cdot \tau_9 = (p_3, \perp, 3?c, \perp, (p, c)) \cdot ((p, c), \perp, 3?c, \perp, p_4) \cdot (p_4, \perp, 3!4(l), \perp, p_5)$*.*
4. $\tau_{10} \cdot \tau_{11} \cdot \tau_{12} = (p'_0, \perp, 4?l, \perp, p'_1) \cdot (p'_1, \perp, lck_4(l_2), \perp, p'_2) \cdot (p'_2, \perp, 4!3(a), \perp, p'_3)$*.*
5. $\tau_{13} \cdot \tau_{14} \cdots \tau_{22} = (p_5, \perp, 3?a, \perp, p_6)(p_6, \perp, rel_3(l_1), \perp, p_7).((t_3, \tau), \alpha, Test(l_1), \beta, q') \cdot ((t_2, \tau'), \epsilon, ((t_3, \tau'), \alpha', Test(l_1), \beta', q') \cdot (p_7, \perp, lck_3(l_1), \perp, p_8) \cdot (p_8, \perp, 3!4(r), \perp, p_9) \cdot (p'_3, \perp, 4?r, \perp, p'_4) \cdot (p'_4, \perp, rel_4(l_2), \perp, p'_5) \cdot (p'_5, \perp, 4!3(a), \perp, p'_0) \cdot (p_9, \perp, 3?a, \perp, p_3)$

Proof. ($\Rightarrow$) Assume a run of the form $(\boldsymbol{c}, \boldsymbol{m_0}, \boldsymbol{l}) \xrightarrow{\sigma} (\boldsymbol{c'}, \boldsymbol{m_0}, \boldsymbol{l})$, such a run can be split as $(\boldsymbol{c}, \boldsymbol{m_0}, \boldsymbol{l}) \xrightarrow{\alpha} (\boldsymbol{c''}, \boldsymbol{m_0}, \boldsymbol{l}) \xrightarrow{\beta} (\boldsymbol{c'}, \boldsymbol{m_0}, \boldsymbol{l})$ such that $\boldsymbol{c''}(1) \in Q_1 \times \Gamma_1$ and $\boldsymbol{c''}(2) \in Q_2 \times \Gamma_2$. Further there are no intermediate configurations between $(\boldsymbol{c''}, \boldsymbol{m_0}, \boldsymbol{l}) \xrightarrow{\beta} (\boldsymbol{c'}, \boldsymbol{m_0}, \boldsymbol{l})$ that has this property (i.e. the intermediate configurations corresponding to process $1, 2$ are not of the form $Q_i \times \Gamma_i$). Inductively for α, there is an α' in the required form. We will now argue that for β, there is a β' in the required form. Notice that in the configuration $(\boldsymbol{c'}, \boldsymbol{m_0}, \boldsymbol{l})$, processes $3, 4$ are waiting for messages and the only transitions enabled for processes-$1, 2$ are of the form mentioned in item-1, 2 (i.e. $\tau_1.\tau_2.\tau_3, \tau_4.\tau_5.\tau_6$). Execution of these transitions enable the transitions τ_7, τ_8 of item-3. Hence the first few transitions w that can be executed from $(\boldsymbol{c'}, \boldsymbol{m_0}, \boldsymbol{l})$ is of the form $w \in \bowtie(\text{"}\tau_1\tau_2\tau_3\text{"} \cdot \tau_7, \text{"}\tau_4\tau_5\tau_6\text{"} \cdot \tau_8) \cup \bowtie(\text{"}\tau_1\tau_2\tau_3\text{"} \cdot \tau_8, \text{"}\tau_4\tau_5\tau_6\text{"} \cdot \tau_7)$ (where $\text{"}\tau_1\tau_2\tau_3\text{"}$ and $\text{"}\tau_4\tau_5\tau_6\text{"}$ are treated as a single letter) and $\bowtie(,)$ is the shuffle operator. It is also easy to see that if there is one such w, there is also an execution of the form $\tau_1 \cdot \tau_2 \cdot \tau_3 \cdot \tau_4 \cdot \tau_5 \cdot \tau_6 \cdot \tau_7 \cdot \tau_8$. After execution of w, processes $1, 2$ are waiting on lock l_1 and process 4 on the task a. The only transition enabled is τ_9. After execution of τ_9, it is easy to see that the only possible execution that can happen is $\tau_{10} \cdot \tau_{11} \cdot \tau_{12}$ of process 4. Now process 4 goes back to waiting for task r and only possible execution at this point is $\tau_{13} \cdot \tau_{14}$. At this point process 3 can go ahead and lock l_1 but this would lead to an dead-lock situation between processes $1, 2, 3$. Hence an execution without deadlock is due to $\tau_{15} \cdot \tau_{16}$ (or $\tau_{16} \cdot \tau_{15}$ since these are independent transitions). Notice that the processes $1, 2$ cannot go ahead and start the simulation of the next move (i.e. $\tau_1.\tau_2 \ldots$), since these transitions need lock l_2. Hence the required sequence is process $3, 4$ executes $\tau_{13} \ldots \tau_{22}$.

($\Leftarrow$) The other direction is simple since $\tau_1 \ldots \tau_{22}$ is the required sequence of transitions that enables a move of the form $(\boldsymbol{c}, \boldsymbol{m_0}, \boldsymbol{l}) \xrightarrow{\tau_1 \ldots \tau_{22}} (\boldsymbol{c'}, \boldsymbol{m_0}, \boldsymbol{l})$. ◀

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

B.1 Proof of Lemma 4

Proof. We prove, by induction on j, $0 \leq j \leq m$, that there is a run of the form $\rho_j = (\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l}) \xrightarrow{\gamma_1 \gamma_2 \cdots \gamma_j}_A (\boldsymbol{c_j}, \boldsymbol{m_j}, \boldsymbol{l_j})$. This suffices to prove the lemma, each thread executes the same sequence of transitions and in the same order and hence the final configuration has to be identical. For the base case, with $j = 0$, there is nothing to prove. For the inductive case, suppose $\gamma_j = \tau_1\tau_2 \ldots \tau_k$ is a phase of thread i. For r, $1 \leq r \leq k$, let $(\boldsymbol{c'_r}, \boldsymbol{m'_r}, \boldsymbol{l'_r})$ the configuration in ρ from which the transition τ_r is executed.

We prove, by another induction, now on r, $0 \leq r \leq k$, that we can extend the run ρ_{j-1} on $\gamma_1 \ldots \gamma_{j-1}$ to a run on $\gamma_1 \ldots \gamma_{j-1}\tau_1\tau_2 \ldots \tau_r$ reaching a configuration $(\boldsymbol{c''_r}, \boldsymbol{m''_r}, \boldsymbol{l''_r})$ such that if $r < k$ then $\boldsymbol{c''_r}(i) = \boldsymbol{c'_{r+1}}(i)$ and $\boldsymbol{l''_r}(i) = \boldsymbol{l'_{r+1}}(i)$.

First of all observe that $\boldsymbol{c'_1}(i) = \boldsymbol{c_{j-1}}(i)$ and $\boldsymbol{l'_1}(i) = \boldsymbol{l_{j-1}}(i)$. These equalities follow from the fact these two configurations are reached along two runs where the thread i has executed exactly the same sequence of transitions. The same cannot be said of the multi-set component, whose contents depend on the tasks posted to i from other threads as well. Since $(\boldsymbol{c_{j-1}}, \boldsymbol{m_{j-1}}, \boldsymbol{l_{j-1}}) = (\boldsymbol{c''_0}, \boldsymbol{m''_0}, \boldsymbol{l''_0})$ this establishes the base case.

For the inductive step, we observe that since τ_r is a transition of the thread i and it was possible to execute it at $(\boldsymbol{c'_r}, \boldsymbol{m'_r}, \boldsymbol{l'_r})$, and $\boldsymbol{c'_r}(i) = \boldsymbol{c''_{r-1}}(i)$ and $\boldsymbol{l'_r}(i) = \boldsymbol{l''_{r-1}}(i)$, by the induction hypothesis, all transitions other than task removing or locking transitions are enabled, and result in the same change to the state, stack and locks, as required. The only interesting case is when τ_r is either a task removing transition or a locking transition. We consider these two cases below.

Suppose, τ_r is a task removing transition. Then, necessarily, $r = 1$ (from the definition of phases). Suppose τ_r removes a task a from the multi-set of thread i. We have to establish that $a \in \boldsymbol{m''_0}(i)$. Since $\boldsymbol{m''_0} = \boldsymbol{m_{j-1}}$, we will show that $a \in \boldsymbol{m_{j-1}}(i)$ by a simple counting argument.

Consider any transition τ in ρ that posts the task a to the thread i in the prefix of ρ leading to the configuration $(\boldsymbol{c'_1}, \boldsymbol{m'_1}, \boldsymbol{l'_1})$ where τ_1 was executed. Each of these transitions belongs to a phase whose index is less than the index of the phase of τ_1 (which is the position number of τ_1). So, they all appear among the phases $\gamma_1, \gamma_2, \ldots, \gamma_{j-1}$. Thus, every one of those post transitions have been executed in the run on $\gamma_1 \ldots \gamma_{j-1}$ and consequently the number of posts of task a to thread i in the run on $\gamma_1\gamma_2 \ldots \gamma_{j-1}$ is at least as the number of such posts in the run upto the execution of τ_1 in ρ.

Now, we count the number of times the task a has removed from the multi-set of thread i along $\gamma_1, \gamma_2, \ldots \gamma_{j-1}$. Each such transition begins a new phase in ρ. Since they occur among $\gamma_1, \ldots, \gamma_{j-1}$, their position (the index of the phase they initiate) is necessarily less (earlier) than the position where τ_1 is executed. Thus, the number of times the task a is removed from the multi-set of thread i in the run on $\gamma_1, \ldots, \gamma_{j-1}$ is at most the number of such transitions in the prefix of ρ upto the execution of τ_1.

Thus, $\boldsymbol{m_{j-1}}(i)(a) > 0$ and τ_1 can be executed as required. Since the same transition is fired we also have the equality in state, stack and lock components as required.

Finally, suppose τ_r is a locking transition on a lock l. Since $\boldsymbol{l''_{r-1}}(i) = \boldsymbol{l'_r}(i)$ and $\boldsymbol{c''_{r-1}}(i) = \boldsymbol{c'_r}(i)$, the only reason this transition is not enabled at $(\boldsymbol{c''_{r-1}}, \boldsymbol{m''_{r-1}}, \boldsymbol{l''_{r-1}})$ is because l belongs to $\boldsymbol{l''_{r-1}}(j)$ for some $j \neq i$. By the definition of the task-locking policy, this lock l should have been taken in a lock-holding phase γ_z with $z \leq j - 1$, as a matter fact, in the first transition of such a phase. This means that this transition must necessarily occur before the first transition of phase j in the run ρ. And by the definition of phase decompositions, this lock is never released in ρ after this point. This contradicts the fact that the transition τ_r was executed in ρ. Thus, the lock transition must be enabled at $(\boldsymbol{c''_{r-1}}, \boldsymbol{m''_{r-1}}, \boldsymbol{l''_{r-1}})$ and again the invariant is seen to hold after its execution.

This completes the inner inductive proof and thus also the proof of the lemma.

◀

B.2 Construction of MPDS from N-MPDS

In this section, we will formalise and prove correctness of the construction presented in Section-5.2. Before we describe the construction, we fix the N-MPDS to be $A = (\Sigma, \mathcal{P}, \mathcal{L})$ and the destination function to be $\boldsymbol{r}$. Further we will let $\mathcal{P} = \{P_1 \dots P_n\}$, where $P_i = (Q_i, \Gamma_i, \mathcal{O}_i, \delta_i, s_i, \alpha_i)$. We will also assume WLOG that the states Q_i of P_i are disjoint. We will show how to construct an MPDS $M(w) = (\Sigma', P)$ for any fixed valid guiding sequence $w = (l_1, i_1) \dots (l_k, i_k)$.

The alphabet Σ' of the MPDS $M(w)$ is given by $\Sigma' = (\Sigma \times [1..N] \times [0..k]) \cup \bigcup_{i \in [1..N]} (Q_i \times \{i\})$. The symbols $\Sigma \times [1..N] \times [0..k]$ will be used to store the tasks in multi-set. Such tasks are additionally tagged with process and phase number to which they belong. Additionally we also have the states tagged with the process-id. This will be used to store the current state of each process.

The pushdown system $P = (Q, \Gamma, \mathcal{O}, \delta, s_1, \alpha_i)$ is described as follows.

- The set of states Q is given by $Q = \{s_1 \cdots s_N\} \cup \{s\} \times [0..k] \cup \bigcup_{i \in [1..N]} Q_i \times [0..k] \times [0..k] \times (\mathcal{L} \cup \{\emptyset\}) \cup \bigcup_{i \in [1..N]} Q_i \times [0..k] \times (\mathcal{L} \cup \{\emptyset\})$. The set of states include a start state and some intermediary states, set of states that non-deterministically chooses to schedule the task phase or the boundary + locking phase and the states that will be used in simulating the task phase as well as the locking phase. Both the states corresponding to locking phase and the task phase have the ability to indicate whether locks are held (by storing the first lock in the sequence) or no locks are taken. We only need to hold the first taken lock since the locks are guaranteed to be taken and released in a well-nested manner.
- The set of stack symbols is collection of all the stack symbols of each of the processes $\Gamma = \bigcup_{i \in [1..N]} \Gamma_i$. WLOG we will assume that the stack symbols other than $\perp$ in each of the processes in $\mathcal{P}$ are disjoint.
- The operations that can be performed on MPDS are simply the read and write operations to the multi-set, $\mathcal{O} = \{?a, !(a) \mid a \in \Sigma'\}$
- The transition relation δ is as described below.

  1. We have a sequence of initialization phase in which the multi-set is populated with the initial state (s_i, i) of each process and also the intial multi-set (α_i, i) symbol. Final state of the initialisation phase is $(s, 0)$ from where the simulation starts. For this, we have the following set of transitions. $\forall i \in [1..N-1], j \in [1..k], (s_i, \perp, !((s_i, i) \cdot (\alpha_i, i, j)), \perp, s_{i+1})$ and $(s_N, \perp, \epsilon, \perp, (s, 0))$.
  2. From any state of the form $(s, j), j \in [0..k-1]$, we can non-deterministically choose to schedule either a task phase or a boundary phase. We can schedule a task-phase of process i only if the boundary phase of process-i was not already scheduled. The j in the state indicates the index in the guiding sequence. Using this it can be determined if the boundary phase of the process was already scheduled or not. The boundary phase is scheduled from a state (s, j) for a process i iff $w[j] = (\emptyset, i)$. If $w[j] = (l, i)$ for some $l \in \mathcal{L}$, then we have a transition that can skip such an index. We detail formally the sequence of transitions below.
     - We have for $0 \leq j \leq k$ the set of transitions $((s, j), \perp, ?(q, i), \perp, (q, j, \emptyset))$ which non-deterministically schedule a task phase of process i.
     - We also have for $0 \leq j < k$ such that $w[j+1] = (\emptyset, i)$, a transition of the form $((s, j), \perp, ?(q, i), \perp, (q, j, j, \emptyset))$ which non-deterministically schedules a boundary phase of process i.

- We also have for j such that $w[j] = (l, i)$ $((s, j), \perp, \epsilon, \perp, (s, j + 1))$, we non-deterministically can skip to next index for lock sequence.

3. We now enumerate the transitions involved in simulating the task phase. Notice that the starting state of such a simulation would be of the form $(q, j, \emptyset)$. The task phase simulation starts with empty stack and empty set of locks and ends in empty stack and empty set of locks. The current state of the process at the end of a task phase is always stored in the multi-set. At the end of such a simulation, we have a transition that updates the multi-set with the current state of the process. To ensure that the task phase always ends in empty set of locks, we always store the first taken lock till it is released. Since the locks are guaranteed to be taken in well-nested manner, once the first taken lock is released, we are sure that the set of locks taken is empty. When in index j, we should ensure that set of locks appearing in indices $[1..j]$ in the guiding sequence should not be taken. We let $L_j = \{l \mid \exists t \leq j : w[t] = (l, i)\}$ to be set of all locks appearing in indices less than j. Lastly we only have a single multi-set in our MPDS. So any messages posted to process j must be simulated by tagging the message with the process-id. Further we also tag the message with an index from $[0..k]$ indicating the stage in which the message will be consumed, this stage is guessed non-deterministically and is always greater equal to the current index. We formalise and list these set of transitions below. In the description below, we will assume that the state $q, q' \in Q_i$ belongs to process i.
   - For any transition of the form $(q, \alpha, \mathrm{lck}_i(l), \beta, q') \in \delta_i$, for all j such that $l \notin L_j$, we add $((q, j, \emptyset), \alpha, \epsilon, \beta, (q', j, l))$ and for $l' \neq l$, $((q, j, l'), \alpha, \epsilon, \beta, (q', j, l'))$
   - For any transition of the form $(q, \alpha, \mathrm{rel}_i(l), \beta, q') \in \delta_i$, for all j such that $l \notin L_j$, we add $((q, j, l), \alpha, \epsilon, \beta, (q', j, \emptyset))$ and for $l' \neq l$, $((q, j, l'), \alpha, \epsilon, \beta, (q', j, l'))$
   - For any transition of the form $(q, \perp, i?a, \beta, q') \in \delta_i$, for all j, we add $((q, j, \emptyset), \perp, ?(a, i, j), \beta, (q', j, \emptyset))$.
   - Similarly for any transition of the form $(q, \alpha, i!m(a), \beta, q') \in \delta_i$, for all j, we add for all $t \geq j$, $((q, j, s), \alpha, !(a, i, t), \beta, (q', j, s))$ for all $s \in \mathcal{L} \cup \{\emptyset\}$.
   - For any transition of the form $(q, \alpha, \epsilon, \beta, q') \in \delta_i$, for all j, we add $((q, j, s), \alpha, \epsilon, \beta, (q', j, s))$ for all $s \in \mathcal{L} \cup \{\emptyset\}$.
   - Finally, we add for all $j \in [1..k-1], i \in [1..N], q \in Q_i$, $((q, j, \emptyset), \perp, !(q, i), \perp, (s, j))$.

4. We now enumerate the transitions involved in simulating the boundary and the lock taking phase. The state (s, j) can non-deterministically guess to schedule a thread in its boundary phase. The starting state of the boundary phase will be of the form $(q, j, j, \emptyset)$. The idea here is to execute the boundary phase along with the rest of the lock-taking phase of the current process. Let sequence $u = w \downarrow_{\mathcal{L} \times \{i\}}$ containts the part of lock taking sequence from the guiding sequence for this process. When simulating the boundary phase, we ensure that no locks from L_j are taken. Let $u = (l_1, i) \dots (l_m, i)$. On reaching a state where transition that takes lock l_1 is enabled and if the current set of locks taken are empty, we execute the lock transition and goto a state of the form $(q', j, j_1, \emptyset)$ where j_1 s the position of (l_1, i) in the original guiding sequence . We continue this till we have executed lock transitions corresponding to the sequence $l_1 \dots l_m$. Finally we also check if the required state for this process is reached. The formal transitions are listed below. In the construction below, we will assume that $q, q' \in Q_i$.
   - For any transition of the form $(q, \alpha, \mathrm{lck}_i(l), \beta, q') \in \delta_i$, for all j' such that $l \notin L_{j'}$, we add $((q, j, j', \emptyset), \alpha, \epsilon, \beta, (q', j, j', l))$, $((q, j, j', l'), \alpha, \epsilon, \beta, (q', j, j', l'))$ where $l' \neq l$ and $((q, j, j', \emptyset), \alpha, \epsilon, \beta, (q', j, j'', \emptyset))$, where $j'' > j'$ is such that $w[j''] = (l, i)$ and

there is no $j' < k < j''$ such that $w[k] = (l', i)$. i.e. it is the next position in w after j' that belongs to process i.

- For any transition of the form $(q, \alpha, \mathrm{rel}_i(l), \beta, q') \in \delta_i$, for all k such that $l \notin L_{j'}$, we add $((q, j, j', l), \alpha, \epsilon, \beta, (q', j, j', \emptyset))$ and $((q, j, j', l'), \alpha, \epsilon, \beta, (q', j, j', l'))$ where $l' \neq l$.
- For any transition of the form $(q, \perp, i?a, \beta, q') \in \delta_i$, for all $j \leq j' \in [1..k]$, we add $((q, j, j', \emptyset), \perp, ?(a, i, j), \beta, (q', j, j, \emptyset))$
- For any transition of the form $(q, \perp, i!h(a), \beta, q') \in \delta_i$, for all $j, j' \in [1..k]$ and $s \in \mathcal{L} \cup \{\emptyset\}$, we add $((q, j, j', s), \alpha, !(a, h, m), \beta, (q', j, j', s))$ for all $m \geq j'$
- For any transition of the form $(q, \alpha, \epsilon, \beta, q') \in \delta_i$, for all j, j', we add $((q, j', s), \alpha, \epsilon, \beta, (q', j', s))$
- Finally we add for all $\alpha \in \Gamma_i$, the transition $((\boldsymbol{r}(i), j, j', \epsilon, s), \alpha, \epsilon, \perp, (s, j+1))$, where j is determined by the process i and the guiding sequence. For a process i, it is the position of $(\emptyset, i)$ in the guiding sequence.

The corretness of the construction follows from the following lemma. We introduce some notations towards the same. For any task phase γ of process-i, we let $\pi(\gamma) = \epsilon$. For any border phase γ of process-i, we let $\pi(\gamma) = (\emptyset, i)$. For any lock-taking phase γ of process-i such that the lock l is held without being released in this phase, we let $\pi(\gamma) = (l, i)$. Given any run of N-MPDS of the form $(\boldsymbol{c_0}, \boldsymbol{m_0}, \boldsymbol{l_0}) \xrightarrow{\sigma} (\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l})$ such that all phases in σ are executed without interleaving, let $\sigma = \gamma_1 \ldots \gamma_m$ be its phase decomposition. We define $\pi(\sigma) = \pi(\gamma_1) \ldots \pi(\gamma_m)$. We will also define a correspondence between the multi-set of the MPDS $M(w)$ that we constructed for some guiding sequence $w = (l_1, i_1) \ldots (l_k, i_k)$ and the multisets of N-MPDS A that was given to us. For any multiset M over Σ', we define $\|\boldsymbol{M}\| : [1..N] \mapsto M[\Sigma]$ i.e. the function from $[1..N]$ to multiset over Σ as $\|\boldsymbol{M}\|(i) = \Sigma_{y \in [1..k]}(a, i, y)$.

▶ **Lemma 12.** *Given a N MPDS $A = (\Sigma, \mathcal{P}, \mathcal{L}, \Delta)$, $(\boldsymbol{c_0}, \boldsymbol{m_0}, \boldsymbol{l_0}) \xrightarrow{\sigma}_A (\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l})$ for some $\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l}$ such that $\mathbf{Stt}(\boldsymbol{c}(i)) = \boldsymbol{r}(i)$ iff $(s, \perp, \boldsymbol{M_0}) \rightarrow_{M(\pi(\sigma))} ((s, k), \perp, \boldsymbol{M})$ with $\|\boldsymbol{M}\| = \boldsymbol{m}$.*

Proof. $(\Rightarrow)$

For this direction, we assume a run of the form $(\boldsymbol{c_0}, \boldsymbol{m_0}, \boldsymbol{l_0}) \xrightarrow{\sigma}_A (\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l})$ in A. We will further assume that $\sigma = \gamma_1 \ldots \gamma_m$ where each γ_i is either a task phase, boundary phase or a lock-taking phase. This we can safely assume due to lemma-4. Let $\pi(\sigma) = r_1 \ldots r_k$ for some k.

First, we define a function $\Pi : [1..m] \mapsto [1..k]$ which maps the positions of decomposition of σ to the last seen boundary or lock-taking phase in the sequence as $\Pi(j) = |\pi(\gamma_1 \ldots \gamma_j)|$. For $j \in [1..k]$, we let L_j to denote the set of all locks appearing in $r_1 \ldots r_j$.

We define a dependency relation between the phases in σ as follows. Let $\sigma = \gamma_1 \ldots \gamma_m$, mark all the lock-taking phases appearing in it. Now for each of the phases γ_i that are not marked complete, we find a phase j that appears before it and a transition τ inside of it that is not already marked such that the transition generates the task $a \in \Sigma$ that is consumed by the phase. We mark both the transition and the phase and add the relation $i \xrightarrow{a} j$. We continue this till only γ_1 is the only unmarked phase. Now corresponding to each task γ_i, we define the dependancy set $\mathcal{D}(i) = \{(a, j) \mid i \xrightarrow{a} j\}$. The following lemmas try to connect the executions corresponding to task phase, boundary and lock taking phases of each process to execution in the MPDS $M(\pi(\sigma))$.

The lemma-13 below says that corresponding to every task phase in the run of A, there is a sub-computation in $M(\pi(\sigma))$ such that the component refering to state of A in the state of M at the beginning and end of the run matches, The set of tasks produced remains the same further there is a way to produce these tasks such that the dependency relation is respected. The proof for this lemma is a straight forward inductive argument and follows from the fact that for every transition in the run σ of A, we have an equivalent transition in $M(\pi(\sigma))$ that we constructed.

▶ **Lemma 13.** *Let* $(\boldsymbol{c_0}, \boldsymbol{m_0}, \boldsymbol{l_0}) \xrightarrow{\sigma}_A (\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l})$ *be a run in* A *such that* $\sigma = \gamma_1 \dots \gamma_m$*. For any* γ_i *a task phase of* σ *and let* $(\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l}) \xrightarrow{\gamma_i} (\boldsymbol{c'}, \boldsymbol{m'}, \boldsymbol{l'})$ *be the corresponding sub-run. Then corresponding to each such* γ_i*, we have an execution of the form* $(((q, \Pi(i), \emptyset), \bot), \boldsymbol{M}) \rightarrow_{M(\pi(\sigma))} (((q', \Pi(i), \emptyset), \bot), \boldsymbol{M'})$ *such that* $q = \mathbf{Stt}(\boldsymbol{c}(i)), q' = \mathbf{Stt}(\boldsymbol{c'}(i)), \|\boldsymbol{M}\| = \boldsymbol{m}$, $\|\boldsymbol{M'}\| = \boldsymbol{m'}$ *and* $\boldsymbol{M'} \geq \boldsymbol{M} + \lfloor \mathcal{D}(i) \rfloor$.

Similarly we have the following lemma-14 which associates the run of each process starting from its boundary phase to completion to a run in M that we constructed. Notice that the run of boundary phase and the lock-taking phases of a process need not be contigous. Hence we have sub-executions of the form $(\boldsymbol{c_1}, \boldsymbol{m_1}, \boldsymbol{l_1}) \xrightarrow{\gamma_{i_1}} (\boldsymbol{c'_1}, \boldsymbol{m'_1}, \boldsymbol{l'_1})$, $(\boldsymbol{c_2}, \boldsymbol{m_2}, \boldsymbol{l_2}) \xrightarrow{\gamma_{i_2}} (\boldsymbol{c'_2}, \boldsymbol{m'_2}, \boldsymbol{l'_2})$ $\dots (\boldsymbol{c_m}, \boldsymbol{m}, \boldsymbol{l_m}) \xrightarrow{\gamma_{i_m}} (\boldsymbol{c'_m}, \boldsymbol{m'_m}, \boldsymbol{l'_m})$ where $\gamma_{i_1}, \gamma_{i_2}, \dots, \gamma_{i_m}$ are the boundary and lock-taking phases occuring in that order in σ. Notice that for each $i \in [1..m-1]$, we have $c'_i(i) = c_{i+1}(i)$. Since for every transition in N-MPDS, our construction adds an equivalent transition in our MPDS, we can find an equivalent run.

▶ **Lemma 14.** *Let* $(\boldsymbol{c_0}, \boldsymbol{m_0}, \boldsymbol{l_0}) \xrightarrow{\sigma}_A (\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l})$ *be a run in* A *such that* $\sigma = \gamma_1 \dots \gamma_m$*. For any* $i \in [1..N]$*, let* α_i *be the sequence obtained by projecting* σ *to* δ_i *and deleting all the task phases from the resulting sequence. Now for any* $\alpha_i = \gamma_{i_1} \dots \gamma_{i_m}$*, let* $(\boldsymbol{c_1}, \boldsymbol{m_1}, \boldsymbol{l_1}) \xrightarrow{\gamma_{i_1}} (\boldsymbol{c'_1}, \boldsymbol{m'_1}, \boldsymbol{l'_1})$*,* $(\boldsymbol{c_2}, \boldsymbol{m_2}, \boldsymbol{l_2}) \xrightarrow{\gamma_{i_2}} (\boldsymbol{c'_2}, \boldsymbol{m'_2}, \boldsymbol{l'_2})$ $\dots (\boldsymbol{c_m}, \boldsymbol{m}, \boldsymbol{l_m}) \xrightarrow{\gamma_{i_m}} (\boldsymbol{c'_m}, \boldsymbol{m'_m}, \boldsymbol{l'_m})$ *be the corresponding induced run, clearly for all* $j \in [1..m-1]$*, we have that* $\boldsymbol{c'_j}(i) = \boldsymbol{c_{j+1}}(i)$*. Corresponding to each such* α_i*, we have an execution of the form* $(((q, j, j, \emptyset, \bot), \boldsymbol{M}) \rightarrow_{M(\pi(\sigma))} (((q', j, j', \emptyset), \bot), \boldsymbol{M'})$ *such that* $q = \mathbf{Stt}(\boldsymbol{c}(i)), q' = \mathbf{Stt}(\boldsymbol{c'}(i)), j = \Pi(i_1) - 1, \|\boldsymbol{M}\| = \boldsymbol{m}, \|\boldsymbol{M'}\| = \boldsymbol{m'_1} + \Sigma_{i \in [2..m]}(\boldsymbol{m'_i} - \boldsymbol{m_i})$ *and* $\boldsymbol{M'} \geq \boldsymbol{M} + \lfloor \mathcal{D}(i) \rfloor$.

Now the required run in our MPDS is obtained as follows. The initialization part in MPDS can be executed such that, we have a run of the form $((s_1, \bot)M_0) \rightarrow (((s, 0), \bot), \boldsymbol{M})$, the initial symbols are tagged with the phase in which they will be used. Now corresponding to every phase in σ, if the phase is a task phase, then we use lemma-13 to obtain an equivalent run and execute it in M. If the phase is a boundary phase then we use lemma-14 to obtain an equivalent run and extend our run. Thus we can inductively construct the required run in $M(\pi(\sigma))$.

($\Leftarrow$) For this direction, we will assume a run of the form $(s, \bot, \boldsymbol{M_0}) \rightarrow_{M(w)} ((s, k), \bot, \boldsymbol{M})$ for some guiding sequence w. Clearly such a run can be split according to number of times it visits a state of the form (s, i) for some i. From state of the form, we either have a run of the form $(s, j) \rightarrow ((q, j,), \bot, \boldsymbol{M}) \rightarrow ((q', j,), \bot, \boldsymbol{M'}) \rightarrow (s, j)$ or of the form $(s, j) \rightarrow ((q, j, j, \emptyset), \bot, \boldsymbol{M}) \rightarrow ((q', j, j', \emptyset), \bot), \boldsymbol{M'}) \rightarrow (s, j+1)$. The following lemma states that if there is a run in the MPDS of the form $((q, j, \emptyset), \bot, \boldsymbol{M}) \rightarrow ((q', j, \emptyset), \bot, \boldsymbol{M'})$, then for all configurations $(\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l})$ such that the $\boldsymbol{c}(i) = (q, \bot)$ and it is reachable from the initial configuration, there is a run to a configuration of the form $(\boldsymbol{c}[i \leftarrow (q', \bot)], \boldsymbol{m'}, \boldsymbol{l'})$. We need the constraint that $(\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l})$ is reachable from the initial state as this would guarantee that the resulting run will have its lock sequence well nested. Recall that in our N-MPDS all executions that start from the initial state follow well-nested locking behaviour.

For each transition we added in our MPDS, there is an equivalent transition in the N-MPDS that matches the states. Our run in the MPDS guarantees that no locks occuring in $w[1..j]$ are touched. Every valid sequence of transitions in the N-MPDS respects well-netsted locking property. Finally we record for every well nested sequence of locks the first lock that was taken and released. This ensures that we release all the locks corresponding to this sub-execution. As a consequence, we have the following lemma.

Before we state the lemma, we introduce the function 2Lock(). Given a prefix of the guiding sequence $w[1..j]$, we define $\text{2Lock}(w[1..j]) : [1..N] \mapsto \mathbb{P}(\mathcal{L})$ as $\text{2Lock}(w[1..j])(i) = \{l \mid \exists k \in [1..j] : w[k] = (l, i)\}$. Such a function constructs for us a lock configuration corresponding to the guiding sequence.

▶ **Lemma 15.** *For any run of the form $((q, j, \emptyset), \perp, \boldsymbol{M}) \to ((q', j, \emptyset), \perp, \boldsymbol{M'})$, we have for all $\boldsymbol{c}$, $\boldsymbol{c'}, \boldsymbol{m}, \boldsymbol{m'}$ such that $\boldsymbol{c}(i) = (q, \perp), \boldsymbol{c'}(i) = (q', \perp), \boldsymbol{m} = \|\boldsymbol{M}\|, \boldsymbol{m'} = \|\boldsymbol{M'}\|, \boldsymbol{l} = \text{2Lock}(w[1..j])$ and $(\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l})$ is reachable from the initial state, a run of the form $(\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l}) \to (\boldsymbol{c'}, \boldsymbol{m'}, \boldsymbol{l})$. Further such a run does not use any locks from L_j.*

Suppose there was a run of the form $((q, j, j, \emptyset), \perp, \boldsymbol{M}) \to ((q', j, j', \emptyset), \perp, \boldsymbol{M'})$ corresponding to process i in the constructed MPDS $M(w)$. Let $u = w\downarrow_{\mathcal{L}\times\{i\}} = (l_1, i) \dots (l_m, i)$. Then we have sub-runs of the form run $((q, j, j, \emptyset), \perp, \boldsymbol{M}) \to ((q_1, j, j, \emptyset), \gamma_1\perp, \boldsymbol{M_1})$ which is the maximal run before lock l_1 is taken and segment number changes from j. Similarly we have the runs $((q_1, j, j, \emptyset), \perp, \boldsymbol{M_1}) \to ((q_2, j, j_2, \emptyset), \perp, \boldsymbol{M_2})$ which is the maximal run before l_2 is taken and segment number changes from j_2 and so on. The following lemma says that for all configurations of the form $(\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l})$ such that it is reachable from the initial state, $\boldsymbol{c}(i) = (q, \perp)$, $\boldsymbol{m} = \|\boldsymbol{M}\|$, $\boldsymbol{l} = \text{2Lock}(w[1..j])$, we have a run of the form $(\boldsymbol{c}, \boldsymbol{m}, \boldsymbol{l}) \to (\boldsymbol{c'}, \boldsymbol{m'}, \boldsymbol{l})$ where $\boldsymbol{c'} = \boldsymbol{c}[i \leftarrow (q', \gamma_1\perp)], \boldsymbol{m'_1} = \|\boldsymbol{M_2} - \boldsymbol{M}\|$. Here in $\boldsymbol{c'}$, only the component of i changes and the number of tasks generated during both the runs are comparable. Similarly for the sub-run of the form $((q_1, j, j, \emptyset), \perp, \boldsymbol{M_1}) \to ((q_2, j, j_2, \emptyset), \perp, \boldsymbol{M_2})$, we have for all configurations $(\boldsymbol{c_1}, \boldsymbol{m_1}, \boldsymbol{l})$ such that it is reachable from the initial configuration, $\boldsymbol{c_1}(i) = (q_1, \gamma_1\perp)$, $\boldsymbol{l} = \text{2Lock}(w[1..j])$, we have a run of the form $(\boldsymbol{c_1}, \boldsymbol{m_1}, \boldsymbol{l_1}) \to (\boldsymbol{c'_1}, \boldsymbol{m'_1}, \boldsymbol{l_1})$, where $\boldsymbol{c'_1} = \boldsymbol{c_1}[i \leftarrow (q_2, \gamma_2\perp)], \boldsymbol{m'_1} - \boldsymbol{m_1} = \|\boldsymbol{M_2} - \boldsymbol{M_1}\|$ and so on. Existence of the sequence of such runs follows from the fact that a transition was added to MPDS because of some transition in the N-MPDS, we simulate the lock and task moves exactly.

▶ **Lemma 16.** *For any $i \in [1..N]$, let $u = w\downarrow_{\mathcal{L}\times\{i\}} = (l_1, i) \dots (l_n, i)$. Let $((q_1, j_1, j_1, \emptyset), \perp, \boldsymbol{M_1}) \to ((q_2, j_1, j_1, \emptyset), \gamma_2\perp, \boldsymbol{M_2} \to ((q_3, j, j_3, \emptyset), \gamma_3\perp, \boldsymbol{M_3}) \to ((q_4, j, j_4, \emptyset), \gamma_4\perp, \boldsymbol{M_4}) \dots ((q_n, j_1, j_n, \emptyset), \perp, \boldsymbol{M_n}))$ be a run of the MPDS (with $q_1, \dots q_n \in Q_i$), where $((q_i, j_1, j_i, \emptyset), \gamma_i\perp, \boldsymbol{M_i}) \to (q_{i+1}, j_1, j_i, \emptyset, \gamma_{i+1}\perp, \boldsymbol{M_{i+1}})$ is the maximal run with j_i occuring in state as a component. Then for all configurations $(\boldsymbol{c_1}, \boldsymbol{m_1}, \boldsymbol{l_1}), (\boldsymbol{c_2}, \boldsymbol{m_2}, \boldsymbol{l_2}), \dots, (\boldsymbol{c_n}, \boldsymbol{m_n}, \boldsymbol{l_n})$ such that they are reachable from the initial state, $\forall m \in [1..n], \boldsymbol{c_m}(i) = (q_i, \gamma_i\perp)$ (here $\gamma_1 = \gamma_n = \epsilon$), $\boldsymbol{l_i} = \text{2Lock}(w[1..j_i])$, then there are sub-executions of the form $(\boldsymbol{c_i}, \boldsymbol{m_i}, \boldsymbol{l_i}) \to (\boldsymbol{c_{i+1}}, \boldsymbol{m_{i+1}}, \boldsymbol{l_i})$ where $\boldsymbol{c_{i+1}} = \boldsymbol{c_i}[i \leftarrow (q_{i+1}, \gamma_{i+1}\perp)]$ and $\boldsymbol{m_{i+1}} - \boldsymbol{m_i} = \|\boldsymbol{M_{i+1}} - \boldsymbol{M_i}\|$.*

with these two lemmas in place, now the required run of the N-MPDS is obtained by arranging the transitions of the corresponding sub-runs of the N-MPDS obtained from lemma-15 and lemma-16 according to the guiding sequence. It is easy to see that transitions thus arranged forms a valid run in the N-MPDS.

◀

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

C.1 Proof of Lemma-6

Proof. The proof is done by reduction from the well-known problem of the satisfiability of a 3-SAT formula to the reachability problem for a stateless well-nested 1-MPDS under the task-locking policy. Given an 3-SAT formula Ψ, let $X = \{x_1, \cdots x_n\}$ be the set of variables and $C = \{C_1, \cdots, C_m\}$ be the set of clauses appearing in Ψ. The 1-MPDS A that we wish to construct is defined by the tuple $(\{a\}, \{P\}, \mathcal{L})$, where $\mathcal{L} = \{l_x^0, l_x^1 \mid x \in X\}$ is the set of locks, two per variable. The thread P executes in three phases. In the first phase, the thread P posts a task a to itself and upon removing the task a from the multi-set, the 1-MPDS moves to a new state from which the stack content is always $a \cdot \bot$. (This because our model does not allow lock operations when the stack is empty). Then, the second phase can start and the 1-MPDS iterates over all variables while acquiring exactly one lock from the set $\{l_x^0, l_x^1\}$ for each variable $x \in X$. Corresponding to a variable x, if P acquires the lock l_x^i then it amounts to assigning it the value $1-i$. Once the 1-MPDS has acquired one lock per variable, the third phase can start. During this phase the 1-MPDS checks the satisfiability of the formula Ψ by checking if each clause can be satisfied by the variable assignments chosen in the previous phase. For this, it iterates over all the clauses one by one. For each of the clause it guesses the variable in that clause that makes the clause true and checks if the corresponding lock is free. More precisely, the 1-MPDS checks the lock l_x^1 (resp. l_x^0) if it guesses that variable x in a clause C evaluates to true (resp. false). Finally, the 1-MPDS moves to a special state if it succeeds to make all the clauses true. It is easy to see that the given 3-SAT formula is satisfiable iff the special state is reachable by the 1-MPDS. Furthermore, since the locks are only acquired but never released, the 1-MPDS is well-nested. The 1-MPDS is also stateless and satisfies the task-locking policy since it removes only one task from its multi-set (namely the task a) in the first phase when all the locks are free.

◀

C.2 Proof of lemma 7

Proof. Let $\alpha_1 \cdots \alpha_n$ be the phase decomposition of σ. Observe that such decomposition is possible thanks to Lemma 4. We will show how to construct a shorter execution whose phase decomposition is polynomially bounded. We define a function `time` that takes as an input a phase π in $\{\alpha_1, \ldots, \alpha_n\}$ and returns: (1) a triple $(a, k, \alpha') \in \Sigma \times \{1, \ldots, N\} \times \{\alpha_1, \ldots, \alpha_n\}$ if the phase α belongs to the thread k and starts by removing a pending task of type a that was posted in the phase α', and (2) 0 otherwise (i.e., the task α is an initial task). We require also that the number of tasks of the form (a, k) that have been created in a phase α' should be always larger than the number of phases α such that $\texttt{time}(\alpha) = (a, k, \alpha')$. Observe that the existence of such function `time` follows immediately from the semantics of N-MPDS.

As a first step towards the shorter execution, we construct, inductively, a directed graph G (representing the causality between the phases) whose set of nodes is $\{\alpha_1, \cdots, \alpha_n\}$ and the set of edges is initially empty. Furthermore, we assume that all the nodes are colored: (1) *white* means that the node is not yet discovered, (2) *red* means that the node is discovered but not yet processed, and (3) *blue* means that all the node is discovered and processed. Initially all the nodes of G are white. Then, we change the color of all lock-holding phases to blue. Observe that there can be at-most $|\mathcal{L}|$ many of blue nodes in G.

Now for each thread i, we change the color to red of its unique boundary phase or the last occurring task phase. Clearly there are at-most N nodes in G that are colored with red. Furthermore, we add an edge from the boundary phase (if it exists) of the thread i to any subsequent locking-holding phase of the thread i. This edge will be labeled by $(i, i, 0)$.

Then, we perform the following steps on the graph G until there is no more red nodes. In each step i, we start by choosing a red node (say α) in G and proceed according of the following cases: The first case is defined when $\texttt{time}(\alpha) = \emptyset$. In this case, we end this step by updating the color of the node α to blue. The second step is defined when $\texttt{time}(\alpha) = (a, k, \alpha')$ and the color of the node α' is not white. In this case, we end the this step by (i) updating the color of the node α to blue, and (ii) adding an edge from α' to α labeled by (a, k, i). Finally, the third case happens when $\texttt{time}(\alpha) = (a, k, \alpha')$ and the color of the node α' is white. In this case, we end this step by (i) updating the color of the node α to blue, (ii) setting the color of the node α' to red, and (iii) adding an edge from α' to α labeled by (a, k, i). Observe that the number of red nodes at any iteration does not exceed the number of threads (i.e., N). Furthermore, the number of nodes with multiple outgoing edges is bounded by N. This due to the fact that adding an edge to a red or blue will reduce the number of red nodes by one.

To get the shorter execution, we first delete all the phases from σ that do not have a blue color in G. Let β the result of this operation. Clearly the phases that are not blue do not contribute to the final stack and lock configurations of the N-MPDS and we can safely remove them. Now we will show how to shorten the sequence β to the required size. For this, let $k \in \{1, \ldots, N\}$. Then, let $\alpha_{i_1} \xrightarrow{(a_{i_1},k,i_1)} \alpha_{i_2} \xrightarrow{(a_{i_2},k,i_2)} \cdots \xrightarrow{(a_{i_s},k,i_s)} \alpha_{i_{s+1}}$ be the maximal dependancy path of the thread k in the graph G whose edges are labeled by a symbol of the form (a, k, i) for some a and i. We say a subsequence $\alpha_{i_r} \xrightarrow{(a_{i_r},k,i_r)} \alpha_{i_{r+1}} \xrightarrow{(a_{i_{r+1}},k,i_{r+1})} \cdots \xrightarrow{(a_{i_j},k,i_j)} \alpha_{i_{j+1}}$ is deletable if each of the phases α_{i_r}, $\cdots, \alpha_{i_{j+1}}$ are task phases, none of them has a multiple outgoing edges and $a_{i_r} = a_{i_{j+1}}$. Once such a deletable subsequence is identified, we delete the phases $\alpha_{i_{r+1}}, \cdots, \alpha_{i_{j+1}}$ from β to obtain a new phase sequence. Then, we also update the graph G by removing the nodes $\alpha_{i_{r+1}}, \cdots, \alpha_{i_{j+1}}$ and adding an edge from $\alpha_{i_{r+1}}$ to $\alpha_{i_{j+2}}$ labeled by (a_{i_1}, k, i_1). Finally, we repeat this operation for each thread i till no more deletable sub-sequence can be found. We claim that the resulting phase sequence σ' is the required compact sequence.

First we will argue about the size. Notice that for each i, the maximal dependancy path in the result G can be of size at-most $\mathcal{O}((2 \cdot N + |\mathcal{L}|) \times |\Sigma|)$: There are at-most N nodes with multiple outgoing edges, at most N boundary phases, $|\mathcal{L}|$ lock-holding phases and there can be at-most $|\Sigma|$ many task phases between them (otherwise we can find a deletable sequence). It is easy to see that all the task phases in σ' are covered by the maximal dependancy path of at-least one of the threads. Now combining this with the observation that there are at-most $N + |\mathcal{L}|$ many locking phases and boundary phases gives us the required bound.

Towards proving that σ' is an execution, we observe that we have only deleted intermediary task phases that have no effect on boundary phase or the last task phase (if the boundary phase does not exist) of each thread and hence has no effect on the reachable stack and lock configurations.

◀