One Stack to Run Them All[⋆]
Reducing Concurrent Analysis to Sequential Analysis under Priority Scheduling

Nicholas Kidd, Suresh Jagannathan, and Jan Vitek

Purdue University
{nkidd,suresh,jv}@cs.purdue.edu

Abstract. We present a reduction from a concurrent real-time program with priority preemptive scheduling to a sequential program that has the same set of behaviors. Whereas many static analyses of concurrent programs are undecidable, our reduction enables the application of any sequential program analysis to be applied to a concurrent real-time program with priority preemptive scheduling.

1 Introduction

Embedded systems are pervasive and are becoming ever more dependent on complex software with significant correctness and reliability requirements. From automobiles to the space shuttle, software is rapidly becoming the most significant part of development time of new devices. Due to the drastic costs of software errors, it is crucial that verification techniques handle the demands and specific requirements of embedded systems. The goal of this work is to broaden the applicability of known software verification techniques from sequential programs to a large class of real-time concurrent programs.

The programming model used in the vast majority of deployed devices defines a set of periodic tasks—tasks that perform computation at a regular interval (period)—that respond to or monitor events. Each task is typically assigned a priority, and tasks are scheduled by a *priority preemptive scheduler*—a scheduler that always chooses to schedule the highest-priority task that is currently runnable. A lower-priority task is preempted when a higher-priority task becomes runnable, and is rescheduled only when the higher-priority task has finished.

The main contribution of our work is a general reduction from a concurrent program with priority preemptive scheduling to a sequential program, which makes the concurrent program amenable to recent research on automated testing of sequential programs (e.g., DART [1], EXE [2], and KLEE [3], to name a few such systems). Our only two restrictions are that the concurrent program has a finite number of tasks, and that the tasks execute with interleaved semantics (e.g., on a uniprocessor). In the embedded world, these restrictions are the norm as they ensure predictability, which is oftentimes more important than absolute performance.

⋆ Supported by NSF under grants CCF-0811631 and CCF-0701832.

For the important case of finite-data concurrent programs, (i.e., can be modeled as a Boolean program or multi-pushdown system), our reduction shows that the problem of determining the set of all possible reachable program configurations is decidable. (Deciding the set of reachable configurations subsumes many testing notions such as statement and condition coverage). While finite-data may seem restrictive, for embedded systems and especially safety-critical systems, it is often the case that a program will pre-allocate the required amount of memory to provide greater predictability (i.e., to remove unpredictable and potentially costly invocations of the memory allocator).

The reason that it is not readily apparent that a concurrent program with priority preemptive scheduling could be reduced to a sequential program is because all of the characteristics of traditional concurrent programs that make analysis difficult are still present. There are multiple threads of execution, shared state, locks, and preemption. Furthermore, each thread is likely to be non-terminating as it must execute once per period. The key insight behind our reduction is that because a preempted lower-priority thread is not rescheduled until the higher-priority thread has finished, the two threads can *share* the same stack. That is, preemption can be modeled as merely a function call. Thus, a concurrent (multi-stack) program can be reduced to a sequential (one-stack) program.

Another important aspect of real-time programming is avoiding *priority inversion*. Priority inversion occurs when a higher-priority thread t_h cannot make progress because a lower-priority thread t_l has ownership of a shared resource, such as a lock. Even worse, a medium-priority thread t_m can preempt t_l, in effect giving t_m priority over t_h. Overall, priority inversion causes t_h's priority to be *lowered* to that of t_l so long as t_l owns the resource. Coupled with priority scheduling, priority inversion can lead to deadlock. Two common protocols [4] for addressing priority inversion include:

1. *Priority Ceiling Protocol* (PCP) statically associates with each shared resource (lock) the priority of the highest-priority thread that may acquire that resource. When a thread t acquires a resource r, t's priority is temporarily raised to r's priority, and is restored when r is released. Note that due to the way priorities are assigned to resources, r's priority must be at least as high as t.
2. *Priority Inheritance Protocol* (PIP) temporarily elevates the priority of a lower-priority thread t_l that owns a resource r required by a higher-priority thread t_h to that of t_h until t_l has released r.

In comparison, PCP is an eager (or pessimistic) protocol, while PIP is a lazy (or optimistic) protocol that avoids elevating priorities until strictly necessary. Moreover, PCP guarantees dead-lock freedom [4], whereas PIP does not.

Our second contribution is to show that configuration reachability of a concurrent finite-data program with a priority preemptive scheduler (i) remains decidable for a PCP-extended programming model, (ii) is undecidable in general for a PIP-extended programming model, and (iii) is decidable for a PIP-extended programming model with properly nested locks.

2 Reduction

A concurrent program is a shared-memory computation by a finite number of threads $t_1, \ldots, t_n$ that execute with interleaved semantics. Associated with each thread t_i, $1 \leq i \leq n$, is a priority, $\mathsf{priority}(t_i)$, and a period, $\mathsf{period}(t_i)$, in which t_i must perform its computation. We assume that each thread completes its task once per period (i.e., all deadlines are met). In addition, our abstraction of time is a *hyperperiod* H, which is the least common multiple of the periods of all threads. Observe that each thread t_i, $1 \leq i \leq n$, must execute $a_i \triangleq H/\mathsf{period}(t_i)$ times per hyperperiod H. Thus, we reduce a concurrent program with heterogeneous periods to a concurrent program with a single period, namely H, by extending the concurrent program to have a_i copies of t_i, where each copy has the same priority. For the remainder of the paper, all threads are assumed to have the same period H. Finally, each thread (copy) becomes schedulable (i.e., is awoken) non-deterministically.

Remark 1. The reduction to a single hyperperiod is a sound over-approximation. For example, a system with threads T_1 and T_2 with periods 2 resp. 3 will have three copies of T_1 and two copies of T_2 because the l.c.m. of the periods is $H = 6$. The reduction to a single period H allows the schedule $T_1T_1T_1T_2T_2$ which is not allowed in the original system. A more precise reduction can be easily encoded by adding additional scheduling constraints (i.e., a finite amount of data) to the program.

The key insight behind our reduction is that because of priority preemptive scheduling, all running threads can *share* the same stack. Consider the case where a thread t is executing with current stack contents u, and another thread t', such that $\mathsf{priority}(t) < \mathsf{priority}(t')$ is awoken non-deterministically. At this point, and with a traditional non-deterministic scheduler, a concurrent program must maintain two active and distinct stacks, namely u and u', because t' could be preempted at any time to allow t to resume execution. However, with *priority* preemptive scheduling, it is guaranteed that t' will *not* be preempted by t, or by any thread t'' where $\mathsf{priority}(t'') < \mathsf{priority}(t')$. Thus, t' can share the same stack at t (see Fig. 1).

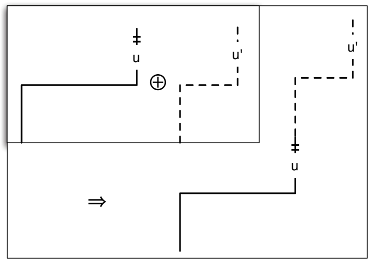

Fig. 1. Sharing stacks u and u'

The reduction is then as follows. First, the priority preemptive scheduler is made explicit by adding to the program the code shown in Fig. 2. The `Hyperperiod` procedure in Fig. 2 executes each thread one time, choosing non-deterministically a sleeping thread to execute via the *`choose`* operation, which returns an index that satisfies the supplied guard. An infinite cycle of hyperperiods is simulated by invoking `Hyperperiod` in a non-terminating loop and ensuring to reset the array `Sleeping` (see below) before doing so. During each hyperperiod, the scheduler has two tasks: (i) it must ensure

```
// Sleeping flags
Sleeping[n] = {true,...,true};
// Thread priorities
Priorities[n] = ...;
// Thread entry points
Threads[n] = ...;

// 0 => choose any thread
Prio = 0;

void Hyperperiod() {
  while ($\bigvee_i$Sleeping[i]) {
    j = choose j: Sleeping[j];
    Sleeping[j] = false;
    Threads[j].entry();
  }
}
```

```
// Wake-up higher-priority thread
void Schedule() {
  // Save current priority
  int prevPrio = Prio;
  for i in (1..n) {
    if (Priorities[i] <= Prio)
      continue;
    if (nondet() && Sleeping[i]) {
      Prio = i;
      Sleeping[i]=false;
      Threads[i].entry();
      break;
    }
  }
  // Restore priority
  Prio = prevPrio;
}
```

Fig. 2. Pseudo-code to execute one hyperperiod

that each thread t is awoken so that t can execute its task; and (ii) the wakeups should happen non-deterministically. The first task is handled by defining a Boolean array of size n, where each entry in the array denotes whether a thread t is sleeping or not. (In Fig. 2, the array is named `Sleeping`.) The scheduler loops until all threads have been awoken and completed their periodic task.

The second task is handled by performing a source-to-source transformation on the code of each thread so that it non-deterministically invokes `Schedule` before each statement `st`. That is, if a thread is comprised of program statements $\mathsf{st}_1,\ldots,\mathsf{st}_k$, then the transformed program will have program statements $\mathsf{st}'_1,\ldots,\mathsf{st}'_k$, where each st' is defined as: $\mathsf{st}' \triangleq$ `Schedule();st`. In the definition of `Schedule` in Fig. 2, the function `nondet` non-deterministically returns true or false. When `Schedule` is invoked, the code of a higher-priority thread $t_{i'}$ than the thread t_i whose code is currently executing may be invoked, which corresponds to t_i being preempted by $t_{i'}$. Before executing a thread t_i by invoking `Threads[i].entry()`, the flag `Sleeping[i]` is set to false to ensure that t_i is executed exactly once per hyperperiod H.

Non-determinism plays a second role, namely, to enumerate all possible orderings of same-priority threads. With priority-preemptive scheduling, a thread will only be preempted by a *higher*-priority thread. If two threads t and t' have the same priority, and because our programming model uses non-deterministic wakeups, schedules in which t executes before t' and *vice versa* must both be considered. Non-determinism allows for both schedules to occur. Moreover, in the finite-data case that is discussed next, *pushdown-system* reachability algorithms naturally consider both schedules.

Table 1. The encoding of an ICFG's edges as PDS rules

PDS Rule	Control flow modeled
$\langle p, n_1 \rangle \hookrightarrow \langle p, n_2 \rangle$	Intraprocedural edge $n_1 \to n_2$
$\langle p, n_c \rangle \hookrightarrow \langle p, e_f\ r_c \rangle$	Call to f, with entry e_f, from n_c that returns to r_c
$\langle p, x_f \rangle \hookrightarrow \langle p, \epsilon \rangle$	Return from f at exit x_f

By reducing a concurrent program with priority preemptive scheduling to a sequential program, existing automated techniques for sequential programs, such as model checkers [5,6] and code-coverage techniques [1,2,3], can be applied to the generated sequential program.

3 Reduction for Multi-PDSs

For the important case of a finite-data programs, each thread can be modeled by a *pushdown system* (PDS), and the concurrent program as a *multi-PDS* [7,8,9]. (We will use the term thread and PDS interchangeably.)

Definition 1. A ***pushdown system*** (PDS) is a tuple $\mathcal{P} = (P, \Gamma, \gamma_0, \Delta)$, where P is a finite set of control states, Γ is a finite stack alphabet, γ_0 is the initial stack symbol of $\mathcal{P}$ specifying the entry point of the modeled thread, and $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ is a finite set of rules. A rule $r \in \Delta$ is denoted by $\langle p, \gamma \rangle \hookrightarrow \langle p', u' \rangle$. A PDS ***configuration*** $\langle p \in P, u \in \Gamma^* \rangle$ is a control state along with a stack. Δ defines a transition system over the set of all configurations. From $c = \langle p, \gamma u \rangle$, $\mathcal{P}$ can make a transition to $c' = \langle p', u'u \rangle$, denoted by $c \Rightarrow c'$, if there exists a rule $\langle p, \gamma \rangle \hookrightarrow \langle p', u' \rangle \in \Delta$. The reflexive transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$.

Without loss of generality, a pushdown rule is restricted to have at most two stack symbols appear on the right-hand side, i.e., for $\langle p, \gamma \rangle \hookrightarrow \langle p', u' \rangle \in \Delta$, $|u'| \leq 2$ [10]. A PDS naturally captures the interprocedural control flow of a thread (see Tab. 1). To model the program state, one typically encodes the global state of the program in P and the local state (i.e., local variables to a function) in Γ. In addition, parameter passing and returning a value from a callee to its caller is modeled by introducing global variables and their corresponding assignments. We direct the reader to Schwoon's thesis [10] for a detailed description.

A concurrent program consists of a set of PDSs $\mathcal{P}_1, \ldots, \mathcal{P}_n$ that share a common set of control states P. For PDS synchronization, any finite-state synchronization protocol can be embedded in P. Because in §4 we consider protocols for addressing priority-inversion in finite-data programs, we will require a mechanism to associate priorities to sections of code that manipulate shared resources (i.e., critical sections). A natural choice—and one common to real-time programming—is to use locks to synchronize execution of critical sections. Thus, we will facilitate these extensions by distinguishing the set $\boldsymbol{L}$, a finite set of *non-reentrant* locks.[1] We now require a mechanism to specify when a thread acquires and releases a lock. We assume that for a PDS $\mathcal{P} = (P, \Gamma, \gamma_0, \Delta)$ and for each lock l in $\boldsymbol{L}$, the following subsets of Δ are defined:

- $\mathsf{acq}(l \in \boldsymbol{L}, \mathcal{P})$ is the set of rules that acquire l;
- $\mathsf{rel}(l \in \boldsymbol{L}, \mathcal{P})$ is the set of rules that release l;
- $\mathsf{acq}(\mathcal{P}) \triangleq \bigcup_{l \in \boldsymbol{L}} \mathsf{acq}(l, \mathcal{P})$ is the set of rules that acquire any lock;
- $\mathsf{rel}(\mathcal{P}) \triangleq \bigcup_{l \in \boldsymbol{L}} \mathsf{rel}(l, \mathcal{P})$ is the set of rules that release any lock; and
- $\mathsf{nolock}(\mathcal{P}) \triangleq \Delta \setminus \big(\mathsf{acq}(\mathcal{P}) \cup \mathsf{rel}(\mathcal{P})\big)$ is the set of non-locking rules.

Altogether, a concurrent program consists of a global state space P, a finite set of threads $\mathcal{P}_1, \ldots, \mathcal{P}_n$ that share the same state space P, and a finite set of locks $\boldsymbol{L}$. Because a concurrent program consists of a finite number of threads $\mathcal{P}_1, \ldots, \mathcal{P}_n$, we assume that the threads are sorted according to their priority.

Definition 2. A ***multi*-PDS** is a tuple $\Pi = (P, p_0, \mathcal{P}_1, \ldots, \mathcal{P}_n, \boldsymbol{L})$, where P is the shared control state of each PDS $\mathcal{P}_i = (P, \Gamma_i, \gamma_0^i, \Delta_i)$, $1 \leq i \leq n$; $p_0 \in P$ is the initial control state; and $\boldsymbol{L} = \{l_1, \ldots, l_{|\boldsymbol{L}|}\}$ is a finite set of $|\boldsymbol{L}|$ non-reentrant locks. A ***global configuration*** $\langle p, u_1, \ldots, u_n, \bar{o}\rangle$ is a tuple consisting of:

- a control state $p \in P$ modeling the global state of Π;
- a stack u_i for each PDS $\mathcal{P}_i$, $1 \leq i \leq n$, where $u_i \in \Gamma_i^* \cup \{\top\gamma_0^i\}$ and $\top \notin \Gamma_i$ is a unique stack symbol that is used to denote a sleeping thread (discussed below); and
- an ***ownership array*** $\bar{o}$ of length $|\boldsymbol{L}|$, in which each entry indicates the owner of a given lock: for each $1 \leq j \leq |\boldsymbol{L}|$, $\bar{o}[j] \in \{0, 1, \ldots, n\}$ indicates the identity i of the PDS $\mathcal{P}_i$ that holds lock l_j (0 signifies that l_j is free). Given $\bar{o}$, a state change in which $\mathcal{P}_i$ acquires lock l_j is denoted by $\bar{o}[j \mapsto i]$, and a state change in which $\mathcal{P}_i$ releases lock l_j—setting l_j's owner to 0—is denoted by $\bar{o}[j \mapsto 0]$. Let $\bar{o}_0$ denote $\bar{o}$ with all entries set to 0.

The set of all global configurations is denoted by $\mathcal{G}$. The ***initial global configuration*** is $g_0 = \langle p_0, \top\gamma_0^1, \ldots, \top\gamma_0^n, \bar{o}_0\rangle$. $\mathcal{P}_i$ is ***active*** in a global configuration g, denoted $\mathsf{active}(g, \mathcal{P}_i)$ if its stack contents $u_i \neq \top\gamma_0^k \vee \epsilon$, which stipulates that $\mathcal{P}_i$ is neither waiting to begin execution—$u_i \neq \top\gamma_0^i$—nor has finished execution—$u_i \neq \epsilon$. The ***priority*** of g, denoted $\mathsf{priority}(g)$, is the maximum of the active threads: $\mathsf{priority}(g) = \mathsf{max}(\{\mathsf{priority}(\mathcal{P}_i) \mid \mathsf{active}(g, \mathcal{P}_i)\})$.

A global configuration $g = \langle p, u_1, \ldots, u_n, \bar{o}\rangle$ can be thought of as representing the set of (local) PDS configurations $\{\langle p, u_i\rangle \mid 1 \leq i \leq n\}$. For the initial global configuration $g_0 = \langle p_0, \top\gamma_0^1, \ldots, \top\gamma_0^n, \bar{o}_0\rangle$, the special stack symbol $\top$ denotes that each thread is waiting to begin execution.

Interleaved execution of Π is defined by the transition relation $\rightsquigarrow \subseteq \mathcal{G} \times \mathcal{G}$ on global configurations. As is customary, we will use $g \rightsquigarrow g'$ to denote that

[1] Reentrant locks that are acquired and released at procedure boundaries are reducible to non-reentrant locks [11].

$(g, g') \in \rightsquigarrow$. Intuitively, there are two types of transitions that Π can perform to go from g to g'. The first transition type is that a sleeping thread is awoken non-deterministically. In the initial global configuration g_0, the stack contents of each PDS $\mathcal{P}_i$, $1 \leq i \leq n$, is $\top\gamma_0^i$, where the special stack symbol $\top$ denotes that $\mathcal{P}_i$ is sleeping. For $\mathcal{P}_i$ to be awoken, the special stack symbol $\top$ must be popped from the top of $\mathcal{P}_i$'s stack. We observe that at a global configuration g where $\mathcal{P}_i$ is sleeping, delaying the wake-up of $\mathcal{P}_i$ until after all currently-running higher-priority threads (i.e., $\{\mathcal{P}_{i'} \mid i \neq i' \land \mathsf{priority}(\mathcal{P}_i) < \mathsf{priority}(\mathcal{P}_{i'}) \land \mathsf{active}(g, \mathcal{P}_{i'})\}$) have finished execution results in the same set of configurations being reachable from g—$\{g' \mid g \rightsquigarrow^* g'\}$—*modulo*$_\top$, where modulo$_\top$ denotes that the stacks $\top\gamma_0^i$ and γ_0^i are considered equal. The reasoning is straightforward: even if $\mathcal{P}_i$ were to be awoken, it would not be able to perform any computation steps until $\mathcal{P}_{i'}$ has finished execution, at which point non-determinism in $\rightsquigarrow$ would allow $\mathcal{P}_i$ to be awoken resulting in the same set of reachable configurations modulo$_\top$.

The second transition type is that the highest-priority thread that has already been awoken is able to update the global state and its (local) stack. Only the highest-priority thread is able to make a transition because the programming model uses a priority preemptive scheduler. We now formally define exactly when $g \rightsquigarrow g'$ holds for Π.

1. $\langle p, u_1, \ldots, \top\gamma_0^i, \ldots, u_n, \bar{o}\rangle \rightsquigarrow \langle p, u_1, \ldots, \gamma_0^i, \ldots, u_n, \bar{o}\rangle$ *iff* $\mathsf{priority}(g) < \mathsf{priority}(\mathcal{P}_i)$. Thread $\mathcal{P}_i$ is only awoken if $\mathcal{P}_i$ has a higher-priority than the currently executing thread.
2. $\langle p, u_1, \ldots, \gamma_i u_i, \ldots, u_n, \bar{o}\rangle \rightsquigarrow \langle p', u_1, \ldots, u' u_i, \ldots, u_n, \bar{o}'\rangle$ *iff* $\mathsf{priority}(g) = \mathsf{priority}(\mathcal{P}_i)$ and $r_i = \langle p, \gamma\rangle \hookrightarrow \langle p', u'\rangle \in \Delta_i$ and:
   (a) If $r_i \in \mathsf{nolock}(\mathcal{P}_i)$, then $\bar{o}' = \bar{o}$. The transition enabled by r_i does not update the state of any lock $l_j \in \boldsymbol{L}$.
   (b) If $r_i \in \mathsf{acq}(l_j \in \boldsymbol{L}, \mathcal{P}_i)$ and $\bar{o}[j] = 0$, then $\bar{o}' = \bar{o}[j \mapsto i]$. The lock l_j must be free in g, and is owned by $\mathcal{P}_i$ in g'.
   (c) If $r_i \in \mathsf{rel}(l_j \in \boldsymbol{L}, \mathcal{P}_i)$ and $\bar{o}[j] = i$, then $\bar{o}' = \bar{o}[j \mapsto 0]$. The lock l_j must be owned by $\mathcal{P}_i$ in g, and is free in g'.

The reflexive transitive closure of $\rightsquigarrow$ is denoted by $\rightsquigarrow^*$.

3.1 Model Checking Problem

As is common in PDS-based model checking [12,13,7,8], the problem of interest is to compute reachability.

Problem 1. Given Π and $g \in \mathcal{G}$, compute the set of forwards reachable configurations $G' = \{g' \mid g \rightsquigarrow^* g'\}$.

We restrict ourselves to reachability from a single global configuration g not for any technical reason, but because of the nature of embedded software. As discussed in §2, the target application consists of a finite set of periodic tasks (threads), and it is assumed that each thread has the same period and completes one task each period (i.e., makes its deadline). Hence, the concurrent program consists of an infinite cycle of periods, where for the finite-data case, the only difference between starting configurations is the initial state p, which is p_0 at program onset. Given a black box to solve *Problem 1* (i.e., to compute the set of *single-period* reachable configurations G' from $g \in \mathcal{G}$), then the set of *all* reachable configurations can be computed via repeated queries—there are only a finite number of states p to start from because P is finite, the stack of each PDS $\mathcal{P}_i$ always begins in the initial stack $\top\gamma_0^i$, and a successive period can only begin from a state p in the set $\{p \mid \langle p, \epsilon_1, \ldots, \epsilon_n, \bar{o}_0\rangle \in G'\}$.[2]

Problem 1 is decidable for Π, and shown by reduction to context-bounded analysis (CBA) [7,14].[3]

Theorem 1. *Given $\Pi = (P, p_0, \mathcal{P}_1, \ldots, \mathcal{P}_n, \boldsymbol{L})$ and $g \in \mathcal{G}$, the set $G' = \{g' \mid g \rightsquigarrow^* g'\}$ of single-period forwards reachable configurations from g is computable in at most $O(n)$ execution contexts.*

Proof. A thread $\mathcal{P}_i$ can preempt another thread $\mathcal{P}_j$ at most one time because once $\mathcal{P}_i$ preempts $\mathcal{P}_j$, by definition $\mathcal{P}_j$ cannot restart execution until $\mathcal{P}_i$ has finished execution. Thus, the number of preemptions is bounded by $O(n)$ which also bounds the number of execution contexts by $O(n)$. $\square$

3.2 A More Efficient Reduction

We now present a reduction from a multi-PDS Π with priority preemptive scheduling to a single PDS $\mathcal{P}_\Pi$, the benefit of which is that all of the known existing techniques for model checking PDSs, including those for expressive logics both linear and branching [12,15], can be used for model checking multi-PDSs with priority preemptive scheduling. Moreover, the most efficient algorithms for CBA [14] require creating a copy of the global state space for each execution context, resulting in an algorithm to solve *Problem 1* with complexity $O(|P \times \bar{\boldsymbol{O}}|^{2n})$, where $\bar{\boldsymbol{O}}$ is the finite set of all ownership arrays.[4] Because of priority preemptive scheduling, our reduction avoids the need to create copies, resulting in a complexity on the order of $O(|P \times \bar{\boldsymbol{O}}|^2 2^n)$, where the 2^n factor accounts for the n bits in the array `Sleeping` that track whether a thread has run during the (current) hyperperiod. In other words, our reduction adds n *bits*, whereas [14] would add n *copies* of P. (We note that [14] solves a harder problem because it allows for the non-deterministic preemption of any thread, i.e., a stack must be maintained for each thread.)

[2] We assume that each thread releases its acquired locks before completing the desired task. Otherwise, one would also have to possibly enumerate over the ownership arrays when starting a new period as well.

[3] CBA is a program analysis that only considers executions with a bounded number of execution contexts, where an execution context is one continuous (sequential) execution of a single thread (albeit there can be many execution contexts of a thread due to context switching).

[4] $\bar{\boldsymbol{O}}$ is finite because there are a finite number of locks and threads (indices), and can thus be encoded in the control state of a PDS.

Combining $\mathcal{P}_1$,...,$\mathcal{P}_n$, and ownership arrays. The first part of the reduction follows naturally from the definition of Π, $\mathcal{G}$, and $\rightsquigarrow$ from §3. Recall that the PDSs of Π and, in particular, their constituent stack contents in a configuration $g = \langle p, u_1, \ldots, u_n, \bar{o}\rangle \in \mathcal{G}$ are sorted based on priority. Because of priority preemptive scheduling, one can view g as having a stack of stacks. For example, consider a concurrent program Π_3 that consists of three PDSs $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_3$ and set of locks $\boldsymbol{L}_3$, and let $g_3 = \langle p, u_1, u_2, u_3, \bar{o}\rangle$ be a configuration of Π_3. To represent g_3 as a *single-PDS configuration* c_3, we must rearrange the stacks into a single stack as follows: $c_3 = \langle p, u_3u_2u_1\rangle$. We must also store the ownership array $\bar{o}$ somewhere in c_3, and the natural solution is to pair it with the control state p, yielding $c_3 = \langle (p, \bar{o}), u_3u_2u_1\rangle$. Of course, if a thread has yet to be awoken (e.g., $u_3 = \top\gamma_0^3$), then it must not be included in c_3, for otherwise threads of lesser priority (e.g., $\mathcal{P}_1$ and $\mathcal{P}_2$) would not be able to make progress.

Our first step towards defining $\mathcal{P}_\Pi$ is to define the PDS $\mathcal{P}_1^n$ that models the execution of PDSs $\mathcal{P}_1, \ldots, \mathcal{P}_n$ of Π. From the above example configuration c_3, we can see that the ownership array $\bar{o}$ must be encoded in the control state, and the PDS rules of $\mathcal{P}_1^n$ must perform updates to the embedded ownership array. With $\bar{\boldsymbol{O}}$ being the set of all ownership arrays, we define for each PDS $\mathcal{P}_i$, $1 \le i \le n$, the PDS $\mathcal{P}_i'$ whose PDS rules have been modified to account for ownership arrays as follows:

Definition 3. Given a PDS $\mathcal{P}_i$ and set of ownership arrays $\bar{\boldsymbol{O}}$, define $\mathcal{P}_i'$ as follows: $\mathcal{P}_i' = (P \times \bar{\boldsymbol{O}}, \Gamma_i, \gamma_0^i, \Delta_i')$, where $P \times \bar{\boldsymbol{O}}$ encodes an ownership array in each control state of $\mathcal{P}_i'$, Γ_i and γ_0^i are unchanged from the definition of $\mathcal{P}_i$, and Δ_i' contains a set of rules for each rule $r = \langle p, \gamma\rangle \hookrightarrow \langle p', u'\rangle \in \Delta_i$, where each set is r extended to update ownership arrays, defined as follows:

– If $r \in \mathsf{acq}(l_j \in \boldsymbol{L}, \mathcal{P}_i)$, then Δ_i' contains the set of rules: $\{\langle (p, \bar{o}), \gamma\rangle \hookrightarrow \langle (p', \bar{o}'), u'\rangle \mid \bar{o} \in \bar{\boldsymbol{O}} \wedge \bar{o}[j] = 0 \wedge \bar{o}' = \bar{o}[j \mapsto i]\}$.
– If $r \in \mathsf{rel}(l_j \in \boldsymbol{L}, \mathcal{P}_i)$, then Δ_i' contains the set of rules: $\{\langle (p, \bar{o}), \gamma\rangle \hookrightarrow \langle (p', \bar{o}'), u'\rangle \mid \bar{o} \in \bar{\boldsymbol{O}} \wedge \bar{o}[j] = i \wedge \bar{o}' = \bar{o}[j \mapsto 0]\}$.
– If $r \in \mathsf{nolock}(\mathcal{P}_i)$, then Δ_i' contains the set of rules: $\{\langle (p, \bar{o}), \gamma\rangle \hookrightarrow \langle (p', \bar{o}), u'\rangle \mid \bar{o} \in \bar{\boldsymbol{O}}\}$.

Definition 4. Given $\Pi = (P, p_0, \mathcal{P}_1, \ldots, \mathcal{P}_n, \boldsymbol{L})$, and for each $\mathcal{P}_i = (P, \Gamma_i, \gamma_0^i, \Delta_i)$, $1 \le i \le n$, define $\mathcal{P}_i' = (P \times \bar{\boldsymbol{O}}, \Gamma_i, \gamma_0^i, \Delta_i')$ according to Defn. 3, then the PDS $\mathcal{P}_1^n$ that models the execution of Π's constituent PDSs is defined as: $\mathcal{P}_1^n = (P \times \bar{\boldsymbol{O}}, \Gamma_1^n = \bigcup_{i=1}^n \Gamma_i, \gamma_0^1, \Delta_1^n = \bigcup_{i=1}^n \Delta_i')$.

From Defn. 4, we can see that a control state $(p, \bar{o})$ of $\mathcal{P}_1^n$ is a pair that models a control state $p \in P$ from Π, as well as an ownership array $\bar{o}$. The stack alphabet is merely the union of the stack alphabets of the constituent PDSs. By defining $\mathcal{P}_i'$ for PDS $\mathcal{P}_i$, the set of PDS rules have been modified to properly update the ownership array when a PDS transition is made. Overall, $\mathcal{P}_1^n$ models the execution of each PDS, as well as tracking the ownership status of each lock $l \in \boldsymbol{L}$. What is missing is the priority preemptive scheduler that non-deterministically awakens threads and schedules the highest-priority active thread.

Explicit Scheduler. The scheduler shown in Fig. 2 on page 248 is finite-data (i.e., a Boolean program [16]), and thus convertible into a PDS [10], which we will refer to as $\mathcal{P}_{\text{sched}} = (P_{\text{sched}}, \Gamma_{\text{sched}}, \gamma_H, \Delta_{\text{sched}})$, where

- $P_{\text{sched}} = \{1 \ldots n\} \times \{0, 1\}^n$ is a pair where the first component holds the current value of `Prio`, and the second component is the Boolean array `Sleeping`.[5]
- $\Gamma_{\text{sched}} = \{1 \ldots n\} \times \mathsf{Locs}$ is a pair where the first component is the current value of `prevPrio` and the second component is the set of program locations for the code in Fig. 2.
- γ_H is the program location for the start of the `Hyperperiod` procedure in Fig. 2.
- Δ_{sched} is defined using standard Boolean program-to-PDS conversion [10]. (Essentially, interprocedural control flow is encoded via the template in Tab. 1, and global resp. local Boolean variables are encoded in the PDS control state P resp. stack alphabet Γ.)

Combining P_1^n with $\mathcal{P}_{sched}$. We now define from $\mathcal{P}_1^n$ and $\mathcal{P}_{\text{sched}}$, the PDS $\mathcal{P}_\Pi$ whose transition system $\Rightarrow$ simulates the multi-PDS Π with transition system $\rightsquigarrow$. Observe that the transition system of P_Π must include both $\mathcal{P}_1^n$ and $\mathcal{P}_{\text{sched}}$, and thus to the first degree the two PDSs are joined together. The only modification to either PDS is to stitch the set of control states together, and reflect this join in the final set of PDS rules of $\mathcal{P}_\Pi$.

Definition 5. Given $\mathcal{P}_1^n = (P_1^n, \Gamma_1^n, \gamma, \Delta_1^n)$ and $\mathcal{P}_{\text{sched}} = (P_{\text{sched}}, \Gamma_{\text{sched}}, \gamma_H, \Delta_{\text{sched}})$, define $\mathcal{P}_\Pi = (P_\Pi, \Gamma_\Pi, \gamma_H, \Delta_\Pi)$, where

- $P_\Pi = P_{\text{sched}} \times P_1^n$ is a pair where each component holds a value from its constituent set of control states. Recall that $P_{\text{sched}} = \{1 \ldots n\} \times \{0, 1\}^n$ is a priority and an array that determines whether a PDS is sleeping or not, and $P_1^n = P \times \bar{\boldsymbol{O}}$ is P, the original set of control states of Π, paired with $\bar{\boldsymbol{O}}$, the set of ownership arrays.
- $\Gamma_\Pi = \Gamma_1^n \cup \Gamma_{\text{sched}}$ is the union of the constituent stack symbols.
- γ_H is the program location for the start of the `Hyperperiod` procedure in Fig. 2.
- Δ_Π consists of the following two sets of rules:
  1. For each rule $r = \langle (p, \bar{o}), \gamma \rangle \hookrightarrow \langle (p', \bar{o}'), u' \rangle \in \Delta_1^n$ and control state $(\varsigma, \bar{b}) \in P_{\text{sched}}$, Δ_Π contains the set of rules:

     $$\{\langle (\varsigma, \bar{b}, p, \bar{o}), \gamma \rangle \hookrightarrow \langle (\varsigma, \bar{b}, p', \bar{o}'), u' \rangle, \langle (\varsigma, \bar{b}, p, \bar{o}), \gamma \rangle \hookrightarrow \langle (\varsigma, \bar{b}, p, \bar{o}), \gamma_{n_5} \gamma \rangle\}$$

     In the set, the first rule is r extended with a control state from P_{sched}. The control state is not modified as the rules from Δ_1^n do not modify the state of the scheduler. The second rule implements a function call to `Schedule` in Fig. 2, which will non-deterministically invoke the code of a higher-priority thread or return. Moreover, from a configuration $\langle (\varsigma, \bar{b}, p, \bar{o}), \gamma u \rangle$ of $\mathcal{P}_\Pi$, $\mathcal{P}_\Pi$ non-deterministically chooses to simulate $\mathcal{P}_1^n$ or $\mathcal{P}_{\text{sched}}$ depending on which rule is invoked.

[5] The number of distinct priorities is bounded by n because there are only n threads.

2. For each rule $r = \langle(\varsigma, \bar{b}), \gamma\rangle \hookrightarrow \langle(\varsigma', \bar{b}'), u'\rangle \in \Delta_{\text{sched}}$ and control state $(p, \bar{o}) \in P_1^n$, Δ_Π contains the set of rules:
$$\{\langle(\varsigma, \bar{b}, p, \bar{o}), \gamma\rangle \hookrightarrow \langle(\varsigma', \bar{b}', p, \bar{o}), u'\rangle\}.$$
These rules combine the rules of $\mathcal{P}_{\text{sched}}$ with the control states P_1^n of $\mathcal{P}_1^n$. Similar to the above set of rules, the control state of $\mathcal{P}_1^n$ is "passed through" unmodified because the scheduler does not affect that control state of $\mathcal{P}_1^n$.

3.3 Correctness

Correctness of the reduction is established by defining a weak bisimulation between the transition systems of Π and $\mathcal{P}_\Pi$. Weak bisimulation is used because in $\mathcal{P}_\Pi$, the scheduler is made explicit whereas it is implicit in the definition of $\rightsquigarrow$ for Π. Thus, configurations of $\mathcal{P}_\Pi$ should only be considered *visible* if the top-of-stack symbol is not a member of Γ_{sched}. Formally, for a configuration $c = \langle(\varsigma, \bar{b}, p, \bar{o}), \gamma u\rangle$ of $\mathcal{P}_\Pi$, we define $\mathsf{vis}(c) = \gamma \notin (\Gamma_{\text{sched}} \setminus \{\gamma_H\})$, and extend vis to sets of configurations in the usual way. Finally, we define the transition relation $\Rightarrow_{\mathsf{vis}}$ between visible configurations of $\mathcal{P}_\Pi$ as follows:

$$\left\{ c \Rightarrow_{\mathsf{vis}} c' \mid \mathsf{vis}(c) \wedge \mathsf{vis}(c') \wedge \exists c_1, \ldots, c_k : c \Rightarrow c_1 \Rightarrow \ldots \Rightarrow c_k \Rightarrow c' \bigwedge_{1 \le i \le k} \neg\mathsf{vis}(c_i) \right\}$$

We define the relation $\succ \subseteq \mathcal{G} \times \mathsf{vis}(\mathcal{C})$ from the set $\mathcal{G}$ of all global configurations of Π to the set $\mathsf{vis}(\mathcal{C})$ of all visible configurations of $\mathcal{P}_\Pi$ as follows: $g \succ c$ *iff* $g = \langle p, u_1, \ldots, u_n, \bar{o}\rangle \wedge c = \langle(\mathsf{priority}(g), \bar{b}, p, \bar{o}), u_n \circ \cdots \circ u_1\rangle$, where $\bar{b}[i] \triangleq u_i = \top\gamma_0^i$, $\circ$ denotes stack concatenation with the exception that the "sleeping stack" $\top\gamma_0^i$ for thread $\mathcal{P}_i$ is considered a neutral element with respect to concatenation. In addition, we special case the initial global configuration by defining $g_0 \succ \langle(0, \bar{b}, p_0, \bar{o}_0), \gamma_H\rangle$ (note that $\bar{b}$ is true in each position because $u_i = \top\gamma_0^i$ for all i in g_0).

Theorem 2. *The binary relation $\succ \subseteq \mathcal{G} \times \mathsf{vis}(\mathcal{C})$ is a weak bisimulation between the transition systems $(\mathcal{G}, \rightsquigarrow)$ and $(\mathcal{C}, \Rightarrow_{\mathsf{vis}})$ of Π and $\mathcal{P}_\Pi$, respectively.*

Proof (Sketch). The proof proceeds by showing that for $g \succ c$ and $g \rightsquigarrow g'$, then there exists a configuration $c' \in \mathsf{vis}(\mathcal{C})$ such that $c \Rightarrow_{\mathsf{vis}} c'$ and $g' \succ c'$. Likewise, if $g \succ c$ and $c \Rightarrow_{\mathsf{vis}} c''$, then there exists a global configuration g'' such that $g \rightsquigarrow g''$ and $g'' \succ c''$. The complete proof is given in the accompanying technical report [17]. □

4 Priority Inversion

In systems with priority preemptive scheduling, a situation known as *priority inversion* occurs when a higher-priority thread $\mathcal{P}_h$ cannot make progress because it waits on a resource (lock) currently owned by a lower-priority thread $\mathcal{P}_l$. Two protocols for addressing priority inversion are Priority Ceiling Protocol (PCP) and Priority Inheritance Protocol (PIP). We next define each protocol, and show that Problem 1 is (i) decidable for PCP-extended semantics, (ii) undecidable in general for PIP-extended semantics, and (iii) decidable for PIP-extended semantics when lock usage is properly nested.

4.1 Priority Ceiling Protocol

Priority Ceiling Protocol (PCP) statically associates with each shared resource (lock) the priority of the highest-priority thread that may acquire that resource. When a thread acquires a resource, that thread's priority is temporarily set to the priority of the resource, and is restored when the resource is released.

A multi-PDS Π is extended as follows to define the PCP-extended semantics:

1. Π is equipped with a map $\mathcal{M}_{\boldsymbol{L}}$ from (sets of) locks to (sets of) priorities.
2. For a global configuration $g = \langle p, u_1, \ldots, u_n, \bar{o}\rangle$, define $\mathsf{LocksHeld}(\mathcal{P}_i) = \{l_j \mid \bar{o}[j] = i\}$ to be the set of locks held by $\mathcal{P}_i$ at configuration g.
3. The PCP-extended priority of $\mathcal{P}_i$, denoted by $\mathsf{priority}_{\mathsf{PCP}}(\mathcal{P}_i)$, is the maximum of $\mathcal{P}_i$'s statically determined priority and of the set of locks held by $\mathcal{P}_i$: $\mathsf{priority}_{\mathsf{PCP}} = \mathsf{max}(\mathsf{priority}(\mathcal{P}_i), \mathcal{M}_{\boldsymbol{L}}(\mathsf{LocksHeld}(\mathcal{P}_i))$.

We now show that for the PCP-extended semantics, *Problem 1* remains decidable. Decidability follows from Thm. 1. Though not presented here, it is also possible to extend the construction of $\mathcal{P}_\Pi$ to support PCP-extended semantics, which would benefit from the improved complexity.

Theorem 3. *For concurrent program $\Pi = (P, p_0, \mathcal{P}_1, \ldots, \mathcal{P}_n, \boldsymbol{L}, \mathcal{M}_{\boldsymbol{L}})$ with priority preemptive scheduling and PCP-extended semantics,* Problem 1 *is decidable.*

Proof. Thm. 3 follows from Thm. 1. PCP-extended semantics reduces the number of threads that can preempt the currently executing thread $\mathcal{P}_i$: if $\mathcal{P}_i$ has acquired a lock l_j such that $\mathcal{M}_{\boldsymbol{L}}(l_j) > \mathsf{priority}(\mathcal{P}_i)$, then fewer threads can preempt $\mathcal{P}_i$ until $\mathcal{P}_i$ releases l_j. Thus, the number of execution contexts remains bounded by $O(n)$ because the number of valid schedules (i.e., preemptions) of PCP-extended semantics is a subset of non-extended semantics, and the problem is decidable. □

4.2 Priority Inheritance Protocol

Priority Inheritance Protocol (PIP) temporarily elevates the priority of a low-priority thread that owns a resource required by a high-priority thread to that of the high-priority thread until it has released the resource. The PIP-extended semantics is defined by extending Π in the following ways:

1. Let $\Gamma = \bigcup_i \Gamma_i$. Extend each Γ_i, $1 \leq i \leq n$, with the set of fresh stack symbols $\{\bot_l \mid l \in \boldsymbol{L}\}$ where for each $l \in \boldsymbol{L}$, $\bot_l \notin \Gamma$. The new symbol $\bot_l$ is used to denote that a thread is waiting to acquire the lock l.

2. For a global configuration $g = \langle p, u_1, \ldots, u_n, \bar{o}\rangle$ and lock l, define $\mathsf{Waiting}(l)$ to be the set of threads whose top-of-stack symbol is $\bot_l$, i.e., $\mathsf{Waiting}(l) = \{\mathcal{P}_i \mid u_i = \bot_l u'_i \wedge u'_i \in \Gamma_i^*\}$. The set $\mathsf{Waiting}(l)$ is the set of threads that are blocked waiting to acquire the lock l. We extend $\mathsf{Waiting}$ to operate over sets of locks in the natural way.
3. The PIP-extended priority of thread $\mathcal{P}_i$, denoted by $\mathsf{priority}_{\mathsf{PIP}}(\mathcal{P}_i)$, is defined as the maximum of $\mathcal{P}_i$'s statically determined priority and of the threads that wait on a lock owned by $\mathcal{P}_i$:

$$\mathsf{priority}_{\mathsf{PIP}}(\mathcal{P}_i) = \mathsf{max}(\mathsf{priority}(\mathcal{P}_i), \mathsf{priority}_{\mathsf{PIP}}(\mathsf{Waiting}(\mathsf{LocksHeld}(\mathcal{P}_i)))).$$

   The recurrence of $\mathsf{priority}_{\mathsf{PIP}}$ in its own definition ensures that $\mathcal{P}_i$'s priority includes the transitive closure of all threads that are blocked because of the locks $\mathcal{P}_i$ holds, i.e., the threads waiting on locks held by $\mathcal{P}_i$, the threads waiting on locks held by those threads, and so on.
4. Extend $\rightsquigarrow$ to include transitions to and from global configurations where threads are waiting to acquire a lock l as follows:
   (a) $g = \langle p, u_1, \ldots, \gamma_i u_i, \ldots, u_n, \bar{o}\rangle \rightsquigarrow \langle p, u_1, \ldots, \bot_l \gamma_i u_i, \ldots, u_n, \bar{o}\rangle$ *iff* $\mathsf{priority}(g) = \mathsf{priority}(\mathcal{P}_i)$ and $r_i \in \mathsf{acq}(l \in \boldsymbol{L}, \mathcal{P}_i)$ and $\bar{o}[l] \neq 0$. This rule defines a set of transitions where the highest-priority thread in global configuration g attempts to acquire a currently held lock l. Because l is held by another thread, $\mathcal{P}_i$ makes a transition to the waiting state by pushing $\bot_l$ on the top of its stack.[6]
   (b) $g = \langle p, u_1, \ldots, \gamma_i u_i, \ldots, u_n, \bar{o}\rangle \rightsquigarrow \langle p', u'_1, \ldots, u' u_i, \ldots, u'_n, \bar{o}'\rangle$ *iff* $\mathsf{priority}(g) = \mathsf{priority}(\mathcal{P}_i)$ and $r_i = \langle p, \gamma_i\rangle \hookrightarrow \langle p', u'\rangle$ and $r_i \in \mathsf{rel}(l, \mathcal{P}_i)$, where $\bar{o}' = \bar{o}[l \mapsto 0]$ and $u'_k = u''_k$ if $u_k = \bot_l u''_k$ and u_k otherwise. By removing $\bot_l$ from the top of the stack of all threads, those threads that were waiting to acquire l can now re-attempt to do so (while still adhering to priority scheduling).

Each of the listed modifications extends Π by only a finite amount of data and hence the same effect could be achieved by augmenting Π with additional state and PDS rules to encode the scheduling logic.

We consider two cases, that of non-nested and nested lock usage, where lock usage is said to be properly nested if for all program paths, locks are released in the opposite order in which they were acquired. We show that *Problem 1* for a concurrent program with PIP-extended semantics is undecidable in general, and decidable for properly nested locks.

Non-nested locks. When lock usage is not restricted to proper nesting, *Problem 1* for a concurrent program with PIP-extended semantics is undecidable. The

[6] We assume that if PDS $\mathcal{P}_i$ attempts to acquire a lock it has no other transition that can fire from (local) configuration $\langle p, \gamma_i u_i\rangle$. (Such can be made the case via the addition of new stack symbols and rules.) Otherwise, when $\mathcal{P}_i$ is released from waiting (see the next item), it could non-deterministically decide to *not* acquire the lock and hence violate priority scheduling.

proof of undecidability follows from Kahlon et al. [8]. Consider a 2-PDS with three locks $(P, p_0, \mathcal{P}_1, \mathcal{P}_2, \{l_1, l_2, l_3\})$, where $\mathcal{P}_2$ has a higher priority (2) than $\mathcal{P}_1$ (1). One way to show that reachability analysis is undecidable in general for such a system is to develop a scenario where $\mathcal{P}_1$ and $\mathcal{P}_2$ move in lock-step, which would allow the 2-PDS to determine the emptiness of the intersection of two context-free languages—a well-known undecidable problem. To make P_1 and $\mathcal{P}_2$ move in lock-step we must use the PIP-extended semantics. Namely, the PDSs need to acquire and release locks in such a fashion that $\mathcal{P}_2$, which has a higher priority than $\mathcal{P}_1$, repeatedly needs to acquire a lock that is held by $\mathcal{P}_1$. Thus, $\mathcal{P}_1$ will repeatedly inherit $\mathcal{P}_2$'s priority so that it can release the lock.

In [8], this is accomplished by acquiring and releasing the three locks $l_{1\text{–}3}$ in a cycle using hand-over-hand locking. Assume that $\mathcal{P}_1$ currently owns l_1, then $\mathcal{P}_1$ will first acquire l_2 before releasing l_1, and subsequently will acquire l_3 before releasing l_2, and so on *ad infinitum.* In the same scenario, assume that $\mathcal{P}_2$, which in our programming model has a higher priority than $\mathcal{P}_1$, currently owns l_2 and acquires and releases the locks in the same fashion. We can see then that $\mathcal{P}_2$ will acquire l_3, release l_2, and then attempt to acquire l_1, which causes $\mathcal{P}_1$ to inherit the priority of $\mathcal{P}_2$. However, instead of reaching a state when $\mathcal{P}_1$ releases the resources needed by $\mathcal{P}_2$, $\mathcal{P}_1$ acquired l_2 and then releases l_1, which will cause $\mathcal{P}_2$ to again wait on $\mathcal{P}_1$ then next time it completes the cycle and needs l_2. The end result is that $\mathcal{P}_1$ and $\mathcal{P}_2$ chase each other around the lock cycle, which leads to an unbounded number of execution contexts and the ability to solve undecidable problems.[7]

Theorem 4. *For concurrent program $\Pi = (P, p_0, \mathcal{P}_1, \ldots, \mathcal{P}_n, \boldsymbol{L})$ with priority preemptive scheduling and PIP-extended semantics,* Problem 1 *is undecidable.*

Proof. The proof follows from the proof of *Theorem 8* [8, Section 11]. □

Nested locks. When lock usage is properly nested, *Problem 1* is decidable for the PIP-extended semantics. The proof is by reduction to CBA.

Theorem 5. *For concurrent program $\Pi = (P, p_0, \mathcal{P}_1, \ldots, \mathcal{P}_n, \boldsymbol{L})$ with priority preemptive scheduling, PIP-extended semantics, and nested locks,* Problem 1 *is decidable.*

Proof. From Thm. 1, each thread $\mathcal{P}_i$ can still perform at most one preemption. Once $\mathcal{P}_i$ is executing, it can cause lower-priority threads to inherit its priority at most $|\boldsymbol{L}|$ times because lock usage is properly nested and hence the number of locks held by a lower-priority thread is monotonically decreasing with each priority inheritance. Thus, for n threads there is at most one preemption and $|\boldsymbol{L}|$ inheritances per thread which bounds the number of execution contexts by $O(n|\boldsymbol{L}|)$. □

[7] For the reader concerned with reaching a configuration where $\mathcal{P}_1$ owns l_1 and $\mathcal{P}_2$ owns l_2, refer to [8, Appendix].

5 Related Work

Lal and Reps [14] gave a reduction from analysis of a concurrent program under a context bound to analysis of a sequential program. A context bound is required because reachability analysis is undecidable in general for their programming model. By considering only programs that run under priority preemptive scheduling (and not the general preemption model considered by Lal and Reps), the problem becomes decidable. Hence, our reduction is sound and complete, i.e., it is not an under-approximation aimed at bug-finding but a technique for verifying properties of concurrent real-time programs.

Jhala and Majumdar [18] showed that interprocedural analysis of concurrent asynchronous programs is decidable. Whereas they take advantage of asynchrony, we take advantage of having a priority preemptive scheduler. Atig et al. [19] generalized the asynchronous programming model to allow for a finite number of priority levels. They show that reachability analysis of the more general programming model is decidable by reduction to the reachability problem of Petri nets with inhibitor arcs. Their model is more general; however, they do not consider important protocols for addressing priority inversion and moreover our reduction to a single-PDS is more efficient.

KISS [20] coined the merging of two-threaded programs into single-threaded programs. Our scheduler concretization is the generalization of their technique where thread T_1 non-deterministically invokes thread T_2 and the return to T_1 is also non-deterministic. We take advantage of the properties of priority preemptive scheduling to show that the model checking problem is in fact decidable.

Lindstrom et al. [21] use Java PathFinder (JPF) [22] to model check Real-Time Java [23]. While they also consider priority preemptive scheduling, and other RTSJ details not covered here, their approach is a bug-finding approach because JPF is an explicit state model checker that in general cannot explore the entire state space.

6 Concluding Remarks

Our reduction shows that a concurrent real-time program is, in essence, a sequential program under the covers. By reducing the multi-PDS Π to a PDS $\mathcal{P}_{\Pi}$, we are able to leverage efficient algorithms for sequential program analysis to an important class of concurrent ones. A limitation of our approach is the lack of a model of time. For future work, we intend to consider how timed automata [24] could be integrated with Π, and how it would affect the reduction to $\mathcal{P}_{\Pi}$.

Acknowledgements. The authors would like to thank Tomas Kalibera, Akash Lal, and Pavel Parizek.

References

1. Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed automated random testing. In: PLDI 2005: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 213–223. ACM, New York (2005)

2. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: Exe: Automatically generating inputs of death. ACM Trans. Inf. Syst. Secur. 12(2), 1–38 (2008)
3. Cadar, C., Dunbar, D., Engler, D.R.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: 8th USENIX Symposium on Operating Systems Design and Implementation, pp. 209–224. USENIX Association (2008)
4. Sha, L., Rajkumar, R., Lehoczky, J.P.: Priority inheritance protocols: An approach to real-time synchronization. IEEE Trans. Comput. 39(9), 1175–1185 (1990)
5. Ball, T., Rajamani, S.: Automatically validating temporal safety properties of interfaces. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 103–122. Springer, Heidelberg (2001)
6. Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Symposium on Principles of Programming Languages, pp. 58–70. ACM, New York (2002)
7. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)
8. Kahlon, V., Ivancic, F., Gupta, A.: Reasoning about threads communicating via locks. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 505–518. Springer, Heidelberg (2005)
9. Kahlon, V., Gupta, A.: On the analysis of interacting pushdown systems. In: Symposium on Principles of Programming Languages, pp. 303–314. ACM, New York (2007)
10. Schwoon, S.: Model-Checking Pushdown Systems. PhD thesis, Technische Universität München (2002)
11. Kidd, N., Lal, A., Reps, T.: Language strength reduction. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 283–298. Springer, Heidelberg (2008)
12. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model checking. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 135–150. Springer, Heidelberg (1997)
13. Finkel, A., Willems, B., Wolper, P.: A direct symbolic approach to model checking pushdown systems. Elec. Notes in Theor. Comp. Sci. 9 (1997)
14. Lal, A., Reps, T.: Reducing concurrent analysis under a context bound to sequential analysis. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 37–51. Springer, Heidelberg (2008)
15. Walukiewicz, I.: Model checking CTL properties of pushdown systems. In: Kapoor, S., Prasad, S. (eds.) FST TCS 2000. LNCS, vol. 1974, pp. 127–138. Springer, Heidelberg (2000)
16. Ball, T., Rajamani, S.K.: Bebop: a path-sensitive interprocedural dataflow engine. In: PASTE 2001: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, pp. 97–103. ACM, New York (2001)
17. Kidd, N., Jagannathan, S., Vitek, J.: One stack to run them all. Technical Report 10-005, Purdue University (May 2010)
18. Jhala, R., Majumdar, R.: Interprocedural analysis of asynchronous programs. In: POPL 2007: Proceedings of the 34th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 339–350. ACM, New York (2007)
19. Atig, M.F., Bouajjani, A., Touili, T.: Analyzing asynchronous programs with preemption. In: IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, Schloss Dagstuhl - Leibniz-Zentrum Fuer Informatik, pp. 37–48 (2008)

20. Qadeer, S., Wu, D.: Kiss: Keep it simple and sequential. In: PLDI 2004: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, pp. 14–24. ACM, New York (2004)
21. Lindstrom, G., Mehlitz, P.C., Visser, W.: Model checking real-time java using java pathfinder. In: Peled, D.A., Tsay, Y.-K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 444–456. Springer, Heidelberg (2005)
22. The Java PathFinder Team: Java PathFinder (2010), `http://babelfish.arc.nasa.gov/trac/jpf/`
23. Bollella, G., Gosling, J., Brosgol, B., Dibble, P., Furr, S., Turnbull, M.: The Real-Time Specification for Java. Addison-Wesley, Reading (2000)
24. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)