Verifying Periodic Programs with Priority Inheritance Locks

Sagar Chaki
Software Engineering Institute
Email: chaki@sei.cmu.edu

Arie Gurfinkel
Software Engineering Institute
Email: arie@cmu.edu

Ofer Strichman
Technion
Email: ofers@ie.technion.ac.il

***Abstract*—Periodic real-time programs are ubiquitous: they control robots, radars, medical equipment, etc. They consist of a set of tasks, each of which executes (in a separate thread) a specific job, periodically. A common synchronization mechanism for such programs is via Priority Inheritance Protocol (PIP) locks. PIP locks have low programming overhead, but cause deadlocks if used incorrectly. We address the problem of verifying safety and deadlock freedom of such programs. Our approach is based on sequentialization – converting the periodic program to an equivalent (non-deterministic) sequential program, and verifying it with a model checker. Our algorithm, called PIPVERIF, is iterative and optimal – it terminates after sequentializing with the smallest number of rounds required to either find a counterexample, or prove the program safe and deadlock-free. We implemented PIPVERIF and validated it on a number of examples derived from a robot controller.**

I. Introduction

Periodic programs are widely used to control safety-critical systems. They consist of multiple tasks, each performing a specific job (typically, by invoking a function) periodically. Each task runs in its own thread of execution. Thus, periodic programs are inherently concurrent. They have, however, unique characteristics. First, the arrival and maximum processing times of jobs are known *a priory*. Second, each thread has a *unique* and – other than the issue of locks discussed below – *fixed* priority. Hence both the inherent non-determinism of job arrival and the complexity of the scheduling policy (e.g., one that depends on a job's time in the queue) that characterize general concurrent software, are absent for periodic programs. Periodic programs are designed to be correct only under these restrictions. Therefore, verifying them against a completely non-deterministic scheduler (as common with general concurrent software) is too imprecise.

To address this challenge, we developed [1][2] an approach for *time-bounded verification* of periodic programs. Our approach leverages the restrictions on scheduling and job arrival mentioned above. Given a periodic program $\mathcal{C}$ and a time bound t, we verify that $\mathcal{C}$ does not violate a safety property φ when executed for time t from an initial state I. We assume that t, φ and I are user-specified. Our scheduler model is not completely non-deterministic. It preserves relative ordering of jobs and priorities, while abstracting away concrete time. It is thus sound for properties that depend only event ordering, and not the exact times at which events occur. Note that restricting execution time (as opposed to, say, number of context switches [3]) is more natural for a periodic program since time maps directly to the program's execution state. For example, the software that deploys an airbag in a car completes in a fixed amount of time, and therefore, during verification, we are interested in bugs that occur within that time limit only.

Periodic programs use locks for synchronization. However, such locks must prevent priority inversion [4], whereby a thread is blocked by another with lower priority. A priority inversion almost caused the failure of the 1997 PathFinder mission [5]. To this end, several locking protocols have been proposed in literature [4]. Real-time operating systems [6] typically support two versions – the Priority Ceiling Protocol (PCP) lock and the Priority Inheritance Protocol (PIP) lock. Both types of locks prevent priority inversion. The PCP lock eliminates deadlocks as well, but requires additional programming effort. In contrast, the PIP lock is easier to use but leads to deadlock if used incorrectly. In earlier work [1][2] we explored the time-bounded verification of periodic programs with PCP locks. In this paper, we deal with PIP locks.

We use the sequentialization paradigm proposed by Lal and Reps [7], and build on our earlier work on sequentializing periodic programs without PIP locks [2]. In [2], every execution of the periodic program is partitioned logically into *rounds*. During sequentialization, we first fix the total number of rounds. Next, each job (i.e., the periodic execution of a task) is *scheduled*, i.e., assigned a starting and an ending round. Jobs are then executed in order of increasing priority and starting time. Before executing each statement, a job non-deterministically *context switches*, i.e., jumps to a higher round, thereby modeling preemption. Finally, constraints are used to ensure that jobs are appropriately scheduled (e.g., a job never starts while another with higher priority is executing), properly preempted (e.g., a job never preempts another with higher priority), and that rounds are consistent (the value of each shared variable at the end of a round equals its value at the beginning of the next round).

In the context of periodic programs with PIP locks, existing sequentialization approaches [7][2] are inadequate for several reasons. First, the priority of a thread changes dynamically. More importantly, due to priority inheritance, it is possible for the priority of a thread to change even while the thread itself is suspended. Second, an exact bound on the number of rounds needed to account for all possible executions cannot be determined efficiently. Finally, periodic programs with PIP locks can deadlock. However, the existing sequentialization-based deadlock detection algorithm for concurrent programs [8] do not work with priorities, because it requires that every deadlock have a wait-free counterexample. This is not true when priorities are involved (see Sec. IV for more details). Against this background, we make the following contributions.

First, we present an iterative algorithm, called PIPVERIF, for verifying a time-bounded periodic program with PIP locks. PIPVERIF maintains a number R of rounds, starting with R = the total number of jobs. In each iteration, it first checks for counterexamples to safety with R rounds. If such a counterexample, is detected, PIPVERIF terminates with UNSAFE. Otherwise, it checks for the presence of executions with more than R rounds. If there are no such executions, PIPVERIF terminates with SAFE. Otherwise, it increments R and continues with the next iteration. PIPVERIF is optimal – it terminates with the smallest R required to either find a counterexample, or prove the program safe.

Second, we extend PIPVERIF to detect deadlocks. To this end, the sequential program that we generate maintains the transitive closure of the Task-Resource Graph (TRG) [9] in an incremental manner. A node of the TRG represents either a task or a PIP lock. An edge from a task t to a lock l means that the currently executing job of t is blocked trying to acquire l. Similarly, an edge from a lock l to a task t means that l is held by the currently executing job of t. Detecting a deadlock state is equivalent to detecting that the TRG is cyclic.

Finally, we implement PIPVERIF by extending REKH [2]. We validate our tool, called REKPIP, on a set of examples derived from the controller of a LEGO Mindstorms robot. In each case, REKPIP produces the correct result, either proving the program SAFE, producing a counterexample for a user-specified safety property, or detecting a deadlock. These results indicate that our approach is feasible. Our tools and benchmarks are available at http://www.andrew.cmu.edu/user/arieg/Rek/start-rekpip.cde.tar.gz.

It is important to note that assuming a nondeterministic scheduler, as done by virtually the entire literature on concurrent program verification, makes these verification methods *inherently incomplete* even when the execution is bounded, simply because in the real system the scheduler is *not* nondeterministic. The current line of work is therefore the first to present, to the best of our knowledge, a sound and *complete* – relative to the time-bound, and for properties that only depend on the ordering of events – verification method for a (particular type of) a concurrent program. It is also the first empirically validated verification method for periodic programs with PIP locks. Given the popularity of such systems and their criticality, preventing deadlocks and guaranteeing their safety properties is no doubt an important problem.

The rest of this paper is organized as follows. In Section II, we present basic concepts and definitions. In Section III, we present PIPVERIF in details. In Section IV, we survey related work. Finally, we present our implementation, benchmarks, and results in Section V, and conclude in Section VI.

II. PRELIMINARIES

A *task* τ is a tuple $\langle I, T, P, C, A\rangle$, where I is the priority, T – a bounded procedure (i.e., no unbounded loops or recursion) called the task body, P – the period, C – the worst case execution time (WCET) of T, and A, called the release time, is the time at which the task is first enabled[1]. A *periodic program* (PP) is a set of tasks. In this paper, we consider a N-task PP $\mathcal{C} = \{\tau_0, \dots, \tau_{N-1}\}$, where $\tau_i = \langle I_i, T_i, P_i, C_i, A_i\rangle$. We assume that: (i) for simplicity, $I_i = i$; (ii) execution times are positive, i.e., $C_i > 0$; (iii) priorities are *rate-monotonic* [10] and distinct – tasks with smaller period have higher priority; and (iv) $\mathcal{C}$ is schedulable. Let RT_i be the response time of τ_i (i.e., the maximum time taken by any job of τ_i to complete) computed via Rate Monotonic Schedulability [11] analysis.

Bounding Time and Jobs. We verify $\mathcal{C}$ assuming that it executes for one "hyper-period" $\mathcal{H}$ [11], where $\mathcal{H}$ is the least common multiple of $\{P_0, \dots, P_{N-1}\}$. We refer to the resulting time-bounded program as $\mathcal{C}_{\mathcal{H}}$. We also assume that the first job of each task finishes before its period, i.e.,

$$\forall 0 \le i < N \centerdot\ A_i + RT_i \le P_i\ . \qquad (1)$$

Under this restriction, the number of jobs of task τ_i that executes in $\mathcal{C}_{\mathcal{H}}$ is:$J_i = \frac{\mathcal{H}}{P_i}$. The semantics of $\mathcal{C}_{\mathcal{H}}$ is the asynchronous concurrent program:

$$\|_{i=0}^{N-1}\ k_i := 0; \textbf{while}(k_i < J_i \wedge \text{WAIT}(\tau_i, k_i))\ (T_i; k_i := k_i + 1)\ . \qquad (2)$$

where $\|$ is preemptive priority-sensitive interleaving (the CPU is always given to the enabled task with the highest priority, preempting the currently executing task if necessary), $k_i \in \mathbb{N}$ is a counter and $\text{WAIT}(\tau_i, k_i)$ returns FALSE if the current time is greater than $A_i + k_i \times P_i$, and otherwise blocks until time $A_i + k_i \times P_i$ and then returns TRUE. In the rest of the paper, for simplicity and brevity, we write $\mathcal{C}$ to mean $\mathcal{C}_{\mathcal{H}}$.

Synchronization. We assume that tasks synchronize via priority inheritance protocol (PIP) locks [4]. Trying to acquire a PIP lock l involves one of two possibilities. If l is available, it is taken and execution proceeds normally. If the lock is unavailable, the current thread (executing, e.g., task τ) is blocked and the (suspended) thread holding l inherits τ's priority and hence resumes execution. The resumed thread drops back to its previous (i.e., prior to resumption) priority as soon as it releases l, and goes back to being suspended. Note that PIP locks cause blocking, and therefore deadlocks, if used improperly.

Example 1: Consider the task set in Fig. 1(a). A partial schedule (up to time 9) for these values is shown in Fig. 1(b). At time 0, τ_0 starts and acquires l_1. At time 1, τ_1 preempts τ_0 and acquires l_2. At time 2, τ_2 preempts τ_1. At time 3, τ_2 tries to acquire lock l_2 and gets blocked. At this point, τ_1 inherits τ_2's priority (i.e., 2) and resumes execution. At time 4, τ_1 tries to acquire lock l_1 and gets blocked. At this point, τ_0 inherits τ_1's priority (i.e., 2) and resumes execution. At time 5, τ_0 releases lock l_1. The inherited priority of τ_0 drops back to its previous

[1]We assume that time is given in some fixed unit (e.g., milliseconds).

(a)

Task	I_i	A_i	C_i	P_i
τ_2	2	2	2	10
τ_1	1	1	4	20
τ_0	0	0	3	40

(b)

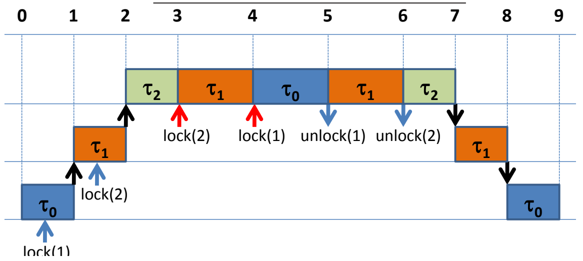

Fig. 1. (a) Three tasks from Example 1; (b) A schedule of the three tasks.

priority, viz., 0, and it is preempted by τ_1 which grabs lock l_1. At time 6, τ_1 releases lock l_2. The inherited priority of τ_1 drops to 1, and it is preempted by τ_2 which grabs lock l_2. At time 7, τ_2 releases lock l_2 and terminates, and τ_1 resumes execution. At time 8, τ_1 releases lock l_1 and terminates, and τ_0 resumes execution. At time 9, τ_0 terminates.

We write $\mathsf{J}(\tau, k)$ to denote the k-th job (i.e., the job at the k-th position) of task τ. Thus, the set of all jobs of $\mathcal{C}$ is:

$$\mathsf{J} = \bigcup_{0 \le i < N} \{\mathsf{J}(\tau_i, k) \mid 0 \le k < J_i\} \,. \tag{3}$$

Job Ordering. Consider a job $j = \mathsf{J}(\tau_i, k_i)$. Recall that A_i, P_i and RT_i are, respectively, the release time, period, and response time of τ_i. Then, the arrival time of j is $A(j) = A_i + k_i \times P_i$, and the departure time of j is $D(j) = A(j) + RT_i$. Since we assume that $RT > 0$, we know that $A(j) < D(j)$. Let $\pi(j) = i$, i.e., the priority of τ_i. We define three ordering relations (developed in our earlier work [2]) on jobs.

Definition 1: The relations $\lhd$, $\uparrow$ and $\sqsubset$ are defined as:

$$\begin{aligned}
j_1 \lhd j_2 &\iff (\pi(j_1) \le \pi(j_2) \wedge D(j_1) \le A(j_2)) \vee \\
&\qquad (\pi(j_1) > \pi(j_2) \wedge A(j_1) \le A(j_2)) \\
j_1 \uparrow j_2 &\iff \pi(j_1) < \pi(j_2) \wedge A(j_1) < A(j_2) < D(j_1) \\
j_1 \sqsubset j_2 &\iff (A(j_1) < A(j_2)) \vee \\
&\qquad (A(j_1) = A(j_2) \wedge \pi(j_1) > \pi(j_2))
\end{aligned}$$

Note that $j_1 \sqsubset j_2$ means that either j_1 always completes before j_2, or it is possible for j_1 to be preempted by j_2. Also, $\sqsubset$ is a total strict ordering since it is a lexicographic ordering by (arrival time, -priority).

Execution. Let $x \bullet y$ be the concatenation of x and y. An execution ρ is a finite sequence of actions where an action is either a job getting blocked (b), or an assertion being violated (a). Note that, for any $k \ge 0$, b^k is the set of executions with k blocks and $b^k \bullet a$ is the set of executions that end with assertion violations and have k blocks. The semantics of a periodic program $\mathcal{C}$, denoted by $[\![\mathcal{C}]\!]$, is a set of executions. Let $[\![\mathring{\mathcal{C}}]\!]$ be the prefix-closure of $[\![\mathcal{C}]\!]$, i.e.,

$$[\![\mathring{\mathcal{C}}]\!] = \{x \mid \exists y \in \{b, a\}^* \centerdot\ x \bullet y \in [\![\mathcal{C}]\!]\}$$

We say that $\mathcal{C}$ is unsafe iff $\exists k \ge 0 \centerdot\ b^k \bullet a \in [\![\mathcal{C}]\!]$.

Algorithm 1 The overall verification algorithm. Function VERIFROUNDS($\mathcal{C}, R$) returns UNSAFE if $\mathcal{C}$ has a counterexample (CEX) with R rounds, INCROUNDS if $\mathcal{C}$ has no R round CEXs, but has legal executions with more than R rounds, and SAFE otherwise, i.e., if $\mathcal{C}$ has no CEXs with R or more rounds.

```
1:  function PIPVERIF($\mathcal{C}$)
2:      $R := |\mathsf{J}|$
3:      loop
4:          $x :=$ VERIFROUNDS($\mathcal{C}, R$)
5:          if $x =$ INCROUNDS then $R := R + 1$
6:          else return $x$

7:  function VERIFROUNDS($\mathcal{C}, R$)
8:      if $[\![\mathcal{S}_a(\mathcal{C}, R)]\!] \neq \emptyset$ then return UNSAFE
9:      if $[\![\mathcal{S}_b(\mathcal{C}, R)]\!] \neq \emptyset$ then return INCROUNDS
10:     else return SAFE
```

III. Job-Bounded Verification

Our verification algorithm PIPVERIF uses the idea that any execution ρ of $\mathcal{C}$ is partitioned into scheduling *rounds* as follows: (a) ρ begins in round 0, and (b) a round ends and a new one begins every time a job ends (i.e., the last instruction of some task body is executed) or gets blocked when trying to acquire a lock.

Example 2: The bounded execution in Fig. 1(b) is partitioned into 5 rounds as follows: round 0 is the time interval $[0, 3]$ – when τ_2 gets blocked trying to acquire lock l_2, round 1 is $[3, 4]$ – when τ_1 gets blocked trying to acquire lock l_1, round 2 is $[4, 7]$ – the end of the first job of τ_2, round 3 is $[7, 8]$, and round 4 is $[8, 9]$.

Since the number of rounds that an execution is partitioned into depends on the number of times a job gets blocked, different executions have different number of rounds. More specifically, the execution b^k or $b^k \bullet a$ has exactly $|\mathsf{J}| + k$ rounds. For soundness, PIPVERIF must therefore use a sufficiently large number of rounds during sequentialization. To this end, PIPVERIF starts with a small number of rounds (specifically, $|\mathsf{J}|$) and iteratively increases it till either a real error is detected, or we prove that all executions have been accounted for.

Algorithm 1 shows the pseudo-code of PIPVERIF. Note that, in each iteration, it invokes VERIFROUNDS($\mathcal{C}, R$) to check if:

1) $\mathcal{C}$ has a counterexample with R rounds – in this case VERIFROUNDS($\mathcal{C}, R$) returns UNSAFE.
2) $\mathcal{C}$ has no counterexample with R rounds, but has legal executions with more than R rounds – in this case VERIFROUNDS($\mathcal{C}, R$) returns INCROUNDS.
3) $\mathcal{C}$ has no legal executions with more than R rounds – in this case VERIFROUNDS($\mathcal{C}, R$) returns SAFE.

Correctness of PIPVERIF. PIPVERIF is correct because it explores all legal executions of the program and only terminates when a real counterexample is detected (i.e., if VERIFROUNDS($\mathcal{C}, R$) returns UNSAFE) or when it proves that no more legal executions remain to be explored (i.e., if VERIFROUNDS($\mathcal{C}, R$) returns SAFE).

A. How VERIFROUNDS *Works*

Recall that VERIFROUNDS$(\mathcal{C}, R)$ must satisfy the following specification:

- if $b^{R-|\mathsf{J}|} \bullet a \in [\![\mathcal{C}]\!]$ then VERIFROUNDS$(\mathcal{C}, R) = \texttt{UNSAFE}$
- else if $\forall k > R - |\mathsf{J}| \centerdot \{b^k, b^k \bullet a\} \cap [\![\mathcal{C}]\!] = \emptyset$ then VERIFROUNDS$(\mathcal{C}, R) = \texttt{SAFE}$
- else VERIFROUNDS$(\mathcal{C}, R) = \texttt{INCROUNDS}$

Consider the pseudo-code of VERIFROUNDS (see Alg. 1). First (line 8), it checks if $b^{R-|\mathsf{J}|} \bullet a \in [\![\mathcal{C}]\!]$. To this end, it constructs a sequential program $\mathcal{S}_a(\mathcal{C}, R)$ such that:

$$[\![\mathcal{S}_a(\mathcal{C}, R)]\!] = \emptyset \iff b^{R-|\mathsf{J}|} \bullet a \notin [\![\mathcal{C}]\!] \tag{4}$$

It then checks if $[\![\mathcal{S}_a(\mathcal{C}, R)]\!] = \emptyset$ using a model checker for sequential programs. Next, to prove that:

$$\forall k > R - |\mathsf{J}| \centerdot \{b^k, b^k \bullet a\} \cap [\![\mathcal{C}]\!] = \emptyset$$

it relies on the following observation:

$$\forall k > R - |\mathsf{J}| \centerdot \{b^k, b^k \bullet a\} \cap [\![\mathcal{C}]\!] = \emptyset \iff b^{R+1-|\mathsf{J}|} \notin [\![\mathring{\mathcal{C}}]\!]$$

Therefore (line 9), it constructs a sequential program $\mathcal{S}_b(\mathcal{C}, R)$ such that:

$$[\![\mathcal{S}_b(\mathcal{C}, R)]\!] = \emptyset \iff b^{R+1-|\mathsf{J}|} \notin [\![\mathring{\mathcal{C}}]\!] \tag{5}$$

and checks whether $[\![\mathcal{S}_b(\mathcal{C}, R)]\!] = \emptyset$ via a model checker for sequential programs. Finally, if both the previous checks fail, it returns $\texttt{SAFE}$ (line 10). In terms of complexity, the construction of $\mathcal{S}_a(\mathcal{C}, R)$ and $\mathcal{S}_b(\mathcal{C}, R)$ are each polynomial in the size of $\mathcal{C}$. The complexity of the subsequent model checking depends on the tool used (e.g., NP for CBMC).

B. Constructing $\mathcal{S}_a(\mathcal{C}, R)$

$\mathcal{S}_a(\mathcal{C}, R)$ reduces the bounded concurrent execution of $\mathcal{C}$ into a sequential execution with R rounds. Initially, jobs are allocated (or scheduled) to rounds. Then, jobs are executed sequentially, in the order $\sqsubset$ defined by Defn. 1. For each global variable g, we guess the initial value of g at the beginning of each round at the start of $\mathcal{S}_a(\mathcal{C}, R)$. At the end of $\mathcal{S}_a(\mathcal{C}, R)$, we ensure that the guessed value of g at the beginning of each round equals its final value at the end of the previous round. In addition, $\mathcal{S}_a(\mathcal{C}, R)$ encodes the inherited priority of jobs and an exception mechanism to detect assertion violations and deadlocks. We now describe these in more detail.

Inherited Priority. Every job $j = \mathsf{J}(\tau, k)$ has a static *base priority* $\pi_b(j)$, which is the priority of the corresponding task τ. In addition, j also has an *inherited priority* $\pi_i(j)$, which changes dynamically as locks are acquired and released. Specifically, at any instant, $\pi_i(j)$ is the maximum of $\pi_b(j)$, and the inherited priorities of all jobs that are blocked on a lock held by j. Note that $\pi_i(j)$ is a global property – it depends not only on the state of j but also on the states of other jobs. The scheduler always executes the non-blocked job with the highest (possibly inherited) priority. Thus, $\mathcal{S}_a(\mathcal{C}, R)$ must keep track of the inherited priorities of jobs to encode PIP locks.

Task-Resource Graph. To compute the inherited priorities of jobs, $\mathcal{S}_a(\mathcal{C}, R)$ encodes the transitive closure of the "task resource graph" [9] (TRG) of the program. The TRG Γ is a dynamic data structure. Its nodes are either tasks or PIP locks. However, its edges depend on the program's execution state. Specifically, an edge from a task t to a lock l means that the currently executing job of t is blocked trying to acquire l. Similarly, an edge from a lock l to a task t means that l is held by the currently executing job of t. Since a job can be blocked on at most one lock at a time, and since a PIP lock can be held by at most one job at a time, a periodic program falls under the category of Single-Resource Model [9] system. For such systems, it is known that Γ is a forest, unless the program's execution state has (two or more) deadlocked tasks [9].

The value of $\pi_i(j)$ is computed from Γ as follows. Let Γ^* denote the transitive closure of Γ, i.e., $(x, y) \in \Gamma^*$ iff there is a path from x to y in Γ. Then,

$$\pi_i(j) = MAX(\{\pi_b(j') \mid (j', j) \in \Gamma^*\}) \ .$$

Thus, if $j = \mathsf{J}(\tau, k)$, then $\pi_i(j)$ is the maximum of the priorities of all tasks that reach τ (including τ itself) in Γ^*. $\mathcal{S}_a(\mathcal{C}, R)$ uses this fact to maintain Γ^* in an online manner – updating it as soon as Γ changes – and compute $\pi_i(j)$ on demand.

Detecting Assertion Violations. In order to model program termination due to an assertion violation, $\mathcal{S}_a(\mathcal{C}, R)$ uses an *exception* mechanism. We use a distinguished global flag to indicate the occurrence of an assertion violation. The flag is initially set to FALSE. Whenever an assertion violation is detected, the corresponding job sets a global flag and exits. All jobs starting (or resuming) in the future check the flag, find it to be set, and also exit. Finally, the flag is used to ensure that $\mathcal{S}_a(\mathcal{C}, R)$ only has a legal execution if $\mathcal{C}$ has an execution with an assertion violation.

Detecting Deadlocks. A deadlock occurs in $\mathcal{C}$ iff its TRG Γ becomes cyclic [9]. More specifically, the deadlocked tasks are exactly the ones whose nodes belong to a cycle in Γ. Therefore, $\mathcal{S}_a(\mathcal{C}, R)$ looks for cycles in Γ whenever a job gets blocked trying to acquire a lock. Since $\mathcal{S}_a(\mathcal{C}, R)$ maintains Γ^* in an online manner, a cycle created in Γ by the addition of an edge is detected in constant time. If a cycle is detected, $\mathcal{S}_a(\mathcal{C}, R)$ uses the exception mechanism described above to indicate an error and abort program execution.

C. Construction of $\mathcal{S}_a(\mathcal{C}, R)$

The structure of $\mathcal{S}_a(\mathcal{C}, R)$ is given by the pseudo-code in Alg. 2 and Alg. 3. Note that $\alpha(e)$ terminates all executions where e evaluates to false. We first describe the global variables of $\mathcal{S}_a(\mathcal{C}, R)$, followed by its functions.

Global Variables of $\mathcal{S}_a(\mathcal{C}, R)$. Recall that $\mathcal{S}_a(\mathcal{C}, R)$ executes the jobs of $\mathcal{C}$ in the order $\sqsubset$ defined by Defn. 1. Each job j is assigned a starting and an ending round during scheduling – these are stored in $start[j]$ and $end[j]$, respectively. Variable rnd stores the current round in which a job is executing. Variable $\mathtt{B}[r]$ indicates whether a job running at round r is allowed to block. Variable $e[r]$ indicates if an exception has been thrown in round r. Variable $\mathtt{P}[r]$ indicates the priority at which the system is executing at round r – this equals the

Algorithm 2 The structure of $\mathcal{S}_a(\mathcal{C}, R)$. Notation: T = set of all tasks; L = set of all PIP locks; J = set of all jobs; G = set of global variables of $\mathcal{C}$; i_g = initial value of g according to $\mathcal{C}$; '$*$' = non-deterministic value; $\alpha()$ = assume().

```
var $rnd, start[], end[], \mathtt{B}[], e[], v_e[], \mathtt{P}[], v_{\mathtt{P}}[], \mathtt{S}[][], v_{\mathtt{S}}[][], \mathtt{T}[][][], v_{\mathtt{T}}[][][], \mathtt{L}[][][], v_{\mathtt{L}}[][][]$    $\forall g \in \mathsf{G}$ . var $g[], v_g[]$
1:  function MAIN( )
2:      INITGLOBS(); HYPPER(); CHECKASSUMPS()
3:  function INITGLOBS( )
4:      $e[0] := 0; \forall l \in \mathsf{L} .\ \mathtt{S}[l][0] := -1$
5:      $\forall t_1 \in \mathsf{T}, t_2 \in \mathsf{T} .\ \mathtt{T}[t_1][t_2][0] := 0$
6:      $\forall t \in \mathsf{T}, l \in \mathsf{L} .\ \mathtt{L}[t][l][0] := 0$
7:      $\forall g \in \mathsf{G} .\ g[0] := i_g$
8:      $\forall r \in [1, R) .\ e[r] := v_e[r] := *; \mathtt{P}[r] := v_{\mathtt{P}}[r] := *$
9:      $\forall l \in \mathsf{L}, r \in [1, R) .\ \mathtt{S}[l][r] := v_{\mathtt{S}}[l][r] := *$
10:     $\forall t_1, t_2 \in \mathsf{T}, r \in [1, R) .\ \mathtt{T}[t_1][t_2][r] := v_{\mathtt{T}}[t_1][t_2][r] := *$
11:     $\forall t \in \mathsf{T}, l \in \mathsf{L}, r \in [1, R) .\ \mathtt{L}[t][l][r] := v_{\mathtt{L}}[t][l][r] := *$
12:     $\forall g \in \mathsf{G}, r \in [1, R) .\ g[r] := v_g[r] := *$
13: function HYPPER( )
14:     SCHEDULEJOBS()
    let $j_0 \sqsubset j_1 \sqsubset \ldots j_{|\mathsf{J}|-1}$ be the job ordering from Defn. 1
15:     RUNJOB($j_0$); ...; RUNJOB($j_{|\mathsf{J}|-1}$)
16: function RUNJOB(Job $j$)
17:     $rnd := start[j]; o := \mathtt{P}[rnd]; \mathtt{P}[rnd] := \pi_b(j)$
18:     if $e[rnd] = 0$ then $\hat{T}(j)$
19:     CS($j$); $\mathtt{P}[rnd] := o$; $\alpha(rnd = end[j])$
20: function $\hat{T}$(Job $j$)
    let $\sigma \equiv$ if $e[rnd] = 1$ then return
    $\hat{T}$ is obtained from $T_t$ by replacing each 'lock(l)' with:
21:     CS($j$); $\sigma$; LOCK($l, j$); $\sigma$
22: each 'unlock(l)' with: CS($j$); $\sigma$; UNLOCK($l, j$)
    each 'assert(x)' with:
23:     CS($j$); $\sigma$; if $\neg x$ then ABORT($j$); return
    and each statement 'st' with:
24:     CS($j$); $\sigma$; $st[g \leftarrow g[rnd]]$
25: function CHECKASSUMPS( )
26:     for $r \in [0, R-1)$ do //let $r' = r + 1$
27:         $\alpha(e[r] = e[r']); \alpha(\mathtt{P}[r] = v_{\mathtt{P}}[r'])$
28:         $\forall l \in \mathsf{L} .\ \alpha(\mathtt{S}[l][r] = v_{\mathtt{S}}[l][r'])$
29:         $\forall t_1 \in \mathsf{T}, t_2 \in \mathsf{T} .\ \alpha(\mathtt{T}[t_1][t_2][r] = v_{\mathtt{T}}[t_1][t_2][r'])$
30:         $\forall t \in \mathsf{T}, l \in \mathsf{L} .\ \alpha(\mathtt{L}[t][l][r] = v_{\mathtt{L}}[t][l][r'])$
31:         $\forall g \in \mathsf{G} .\ \alpha(g[r] = v_g[r'])$
32:     $\forall r \in [0, R) .\ \alpha(\mathtt{B}[r] = 0); \alpha(e[R-1] = 1)$
33: function ABORT(Job $j = \mathsf{J}(\tau, k)$)
34:     $e[rnd] := 1$
35:     $\forall l \in \mathsf{L} .\ \mathtt{S}[l][rnd] = \tau \implies$ UNLOCK($l, j$)
```

(possibly inherited) priority of the currently executing job. For each global variable g of $\mathcal{C}$, variable $g[r]$ indicates its value in round r. The *prophecy* variables $v_e[r]$, $v_{\mathtt{P}}[r]$ and $v_g[r]$ indicate the guessed initial values of $e[r]$, $\mathtt{P}[r]$ and $g[r]$, respectively. The values of $e[r]$, $\mathtt{P}[r]$ and $g[r]$ are updated by the jobs executing in round r only.

Arrays $\mathtt{S}$, $\mathtt{T}$ and $\mathtt{L}$ encode the state of the PIP locks and the transitive closure Γ^* of the TRG. Specifically, $\mathtt{S}[l][r]$ is the priority of the task holding lock l at round r. If l is free at round r, then $\mathtt{S}[l][r] = -1$. Since a task's priority equals its id, we use a task and its priority interchangably. For every pair of tasks (t_1, t_2), $\mathtt{T}[t_1][t_2][r] = 1$ iff $(t_1, t_2) \in \Gamma^*$ at round r. For every task t and lock l, $\mathtt{L}[t][l][r] = 1$ iff $(t, l) \in \Gamma^*$ at round r. Prophecy variables $v_{\mathtt{S}}[l][r]$, $v_{\mathtt{T}}[t_1][t_2][r]$ and $v_{\mathtt{L}}[t][l][r]$ record the guessed initial values of $\mathtt{S}[l][r]$, $\mathtt{T}[t_1][t_2][r]$ and $\mathtt{L}[t][l][r]$, respectively. The values of $\mathtt{S}[l][r]$, $\mathtt{T}[t_1][t_2][r]$ and $\mathtt{L}[t][l][r]$ are updated by jobs executing in round r only.

Functions of $\mathcal{S}_a(\mathcal{C}, R)$. The top-level function is MAIN (see Alg. 2). It initializes all global variables by invoking INIT-GLOBS (line 2), schedules and executes all jobs by invoking HYPPER (line 2), and finally ensures that only legal executions that terminate with an assertion violation or deadlock are allowed by invoking CHECKASSUMPS (line 2).

INITGLOBS (see Alg. 2) initializes all global variables at each round. In particular, for round 0, all globals are initialized (lines 4–7) to their values at the start of the execution of $\mathcal{C}$. For the remaining rounds, they are initialized (lines 8–12) to non-deterministic guessed values. The guessed values are also recorded in the corresponding prophecy variables.

HYPPER (see Alg. 2) first creates a legal schedule for all jobs by invoking SCHEDULEJOBS (line 14) and then executes each job j (line 15) – in the order $\sqsubset$ defined by Defn. 1 – by invoking RUNJOB(j).

In SCHEDULEJOBS (see Alg. 3), line 2 initializes $\mathtt{B}$ to allow jobs to block in all rounds; line 2 also initializes $start$ and end to non-deterministic values; line 3 ensures that $start[j]$ and $end[j]$ are sequential and within legal bounds; line 4 ensures that jobs are properly separated; line 5 ensures that jobs are well-nested – if j_2 preempts j_1, then it finishes before j_1; and line 6 disables job blocks in all rounds in which a job has been scheduled to end.

RUNJOB(j) (see Alg. 2) sets rnd to the round at which j is scheduled to start (line 17), saves the current system priority and then updates it to the base priority of j (line 17), executes a modified version of j but only if no exception has been thrown (line 18), restores the system priority and ensures that j terminates at the appropriate round (line 19).

$\hat{T}(j)$ (see Alg. 2) is identical to the body of j's task, except that it invokes functions LOCK and UNLOCK (shown in Alg. 3) to model the acquiring and releasing of PIP locks (lines 21–22), models assertion violations by invoking ABORT (line 23), and uses variable $g[rnd]$ instead of g (line 24). In addition, $\hat{T}(j)$ increases the value of rnd non-deterministically (by invoking function CS) to model preemption by higher priority jobs prior to each statement. Finally, whenever the value of rnd increases, $\hat{T}(j)$ checks if an exception has been thrown and terminates the job in this case (using the statement σ). Note that rnd can increase only after a call to CS or LOCK.

CHECKASSUMPS (see Alg. 2) ensures that the final value of each global variable at each round is equal to its prophesied

Algorithm 3 The structure of $\mathcal{S}_a(\mathcal{C}, R)$ continued from Alg. 2.

```
1: function SCHEDULEJOBS( )
2:     $\forall r \in [0, R) \centerdot \texttt{B}[r] := 1; \forall j \in \mathsf{J} \centerdot start[j] = *; end[j] = *$
    // Jobs are sequential
3:     $\forall i \in [0, N) \centerdot \forall k \in [0, J_i) \centerdot \alpha(0 \leq start[\mathsf{J}(i,k)] \leq end[\mathsf{J}(i,k)] < R)$
    // Jobs are well-separated
4:     $\forall j_1 \lhd j_2 \centerdot \alpha(end[j_1] < start[j_2]); \forall j_1 \uparrow j_2 \centerdot \alpha(start[j_1] \leq start[j_2])$
    // Jobs are well-nested
5:     $\forall j_1 \uparrow j_2 \centerdot \alpha(start[j_2] \leq end[j_1] \Rightarrow (start[j_2] \leq end[j_2] < end[j_1]))$
6:     $\forall j \in \mathsf{J} \centerdot \texttt{B}[end[j]] = 0$
7: function UNLOCK(int $l$,Job $\mathsf{J}(\tau, k)$)
8:     $\texttt{S}[l][rnd] := -1$; DELLOCKTASK$(l, \tau)$
9: function ADDLOCKTASK(int $l$,Task $\tau$)
10:     $\forall t \in \mathsf{T} \setminus \{\tau\} \centerdot \texttt{T}[t][\tau][rnd] := (\texttt{L}[t][l][rnd] = 1)\ ?\ 1 : \texttt{T}[t][\tau][rnd]$
11: function DELLOCKTASK(int $l$,Task $\tau$)
12:     $\forall t \in \mathsf{T} \setminus \{\tau\} \centerdot \texttt{T}[t][\tau][rnd] := (\texttt{L}[t][l][rnd] = 1)\ ?\ 0 : \texttt{T}[t][\tau][rnd]$
13: function ADDTASKLOCK(int $l$,Task $\tau$)
14:     let $c(t) \equiv (t = \tau \vee \texttt{T}[t][\tau][rnd] = 1)$
15:     $\forall t \in \mathsf{T} \centerdot \texttt{L}[t][l][rnd] := c(t)\ ?\ 1 : \texttt{L}[t][l][rnd]$
16:     $s := \texttt{S}[l][rnd]; \forall t \in \mathsf{T} \centerdot \texttt{T}[t][s][rnd] := c(t)\ ?\ 1 : \texttt{T}[t][s][rnd]$
17: function CS(Job $j = \mathsf{J}(\tau, k)$)
18:     if $(*)$ then return
19:     $o := rnd\,; rnd := *; \alpha(o < rnd \leq end[j])$
20:     $\alpha(\texttt{P}[rnd] = \text{INHERPRIO}(\tau))$
21: function INHERPRIO(Task $\tau$)
22:     return
            $MAX(\{\tau\} \cup \{t \mid \texttt{T}[t][\tau][rnd] = 1\})$
23: function UNBLOCK(int $l$,Job $j$)
24:     $\alpha(\texttt{B}[rnd] = 1); \texttt{B}[rnd] := 0$
25:     $o := rnd\,; rnd := *$
26:     $\alpha(o < rnd \leq end[j])$
27:     $\alpha(\texttt{P}[o] = \texttt{P}[rnd]);\ \alpha(\texttt{S}[l][rnd] = -1)$
28: function LOCK(int $l$,Job $j = \mathsf{J}(\tau, k)$)
29:     if $\texttt{S}[l][rnd] = -1$ then
30:         $\texttt{S}[l][rnd] = \tau$
31:         ADDLOCKTASK$(l, \tau)$
32:     else
33:         if $\texttt{T}[\texttt{S}[l][rnd]][\tau][rnd]$ then
34:             ABORT$(j)$; return
35:         ADDTASKLOCK$(l, \tau)$
36:         UNBLOCK$(l, j)$; DELTASKLOCK$(l, \tau)$
37:         if $e[rnd] = 1$ then return
38:         $\texttt{S}[l][rnd] = \tau$
39:         ADDLOCKTASK$(l, \tau)$
40: function DELTASKLOCK(int $l$,Task $\tau$)
41:     let $c(t) \equiv (t = \tau \vee \texttt{T}[t][\tau][rnd] = 1)$
42:     $\forall t \in \mathsf{T} \centerdot \texttt{L}[t][l][rnd]:=$
            $c(t)\ ?\ 0 : \texttt{L}[t][l][rnd]$
```

initial value at the next round (lines 26–31), all rounds have been exhausted by either a job termination or a job block (line 32), and an exception has been thrown (line 32). Line 32 is critical to ensure the property of $\mathcal{S}_a(\mathcal{C}, R)$ given by (4).

ABORT(j) (see Alg. 2) sets the error flag (line 34) and releases all locks held by j (line 35). To release a lock, it invokes UNLOCK (see Alg. 3) which sets the owner of the lock to -1 (line 8) and then removes the edge in the TRG from the current task to the lock (line 8) via DELLOCKTASK.

DELLOCKTASK (see Alg. 3) updates Γ^* by *removing* an edge in Γ from a lock to a task. In contrast, ADDLOCKTASK (see Alg. 3) updates Γ^* by *adding* an edge in Γ from a lock to a task. Similarly, functions ADDTASKLOCK and DELTASKLOCK (see Alg. 3) update Γ^* by, respectively, adding and removing an edge from a task to a lock.

INHERPRIO(τ) (see Alg. 3) returns the inherited priority of the current job task τ at round rnd. It is invoked by CS (see Alg. 3) to ensure (line 20) that whenever a job is preempted, it only resumes at a round where the system priority equals its inherited priority. In addition, CS ensures (line 19) that a job always resumes in a round permitted by the schedule.

LOCK (see Alg. 3) acquires a lock. If the lock is available (line 29) it updates its owner to the current task (line 30) and adds an edge in the TRG (line 31). However, if the lock is held (line 32), it (i) checks for deadlock and aborts if necessary (lines 33–34); (ii) adds an edge in the TRG from the task to lock (line 35); (iii) preempts the task and resumes it in a future round where the lock is available by invoking UNBLOCK (line 36); (iv) deletes the TRG edge from the task to the lock (line 36); (v) checks if an exception has been thrown and aborts if necessary (line 37); (vi) updates the owner of the lock to the current task (line 38); and (vii) adds a TRG edge from the lock to the task (line 39).

UNBLOCK (see Alg. 3) resumes a blocked job in a future round. It ensures that the current round is available for blocking and makes it unavailable for blocking in the future (line 24), and updates the round to a value that is allowed by the schedule (lines 25–26), where the system priority is the same as the current system priority (line 27), and where the lock is available (line 27).

D. Construction of $\mathcal{S}_b(\mathcal{C}, R)$

Recall that $\mathcal{S}_b(\mathcal{C}, R)$ must have the property defined by (5). The structure of $\mathcal{S}_b(\mathcal{C}, R)$ is similar to $\mathcal{S}_a(\mathcal{C}, R)$. The only difference is in $\hat{T}(j)$ and LOCK, which are shown in Alg. 4. Specifically, in $\mathcal{S}_b(\mathcal{C}, R)$: (i) $\hat{T}(j)$ assumes that assertions are never violated (line 4), and (ii) LOCK assumes that whenever a job blocks, then there is no deadlock (line 10), and aborts if there are no available rounds for job blocking (line 11).

IV. Related Work

Several projects use sequentialization [3][8][12] to verify concurrent software. All these approaches assume a non-deterministic scheduler, which is an over-approximation for periodic programs. Of these, our sequentialization is closest to that of Lal and Reps [7] – scheduling is implemented

Algorithm 4 The structure of $\mathcal{S}_b(\mathcal{C}, R)$. We only show functions that are different from $\mathcal{S}_a(\mathcal{C}, R)$.

```
1:  function $\hat{T}$(Job $j$)
      let $\sigma$ be the statement if $e[rnd] = 1$ then return
      $\hat{T}$ is obtained from $T_t$ by replacing each 'lock(l)' with:
2:        $\text{CS}(j); \sigma; \text{LOCK}(l, j); \sigma$
3:    each 'unlock(l)' with: $\text{CS}(j); \sigma; \text{UNLOCK}(l, j)$
4:    each 'assert(x)' with: $\text{CS}(j); \sigma; \alpha(x)$
5:    and each statement 'st' with: $\text{CS}(j); \sigma; st[g \leftarrow g[rnd]]$
6:  function LOCK(int $l$, Job $j = \mathsf{J}(\tau, k)$)
7:      if $\mathtt{S}[l][rnd] = -1$ then
8:          $\mathtt{S}[l][rnd] = \tau$; ADDLOCKTASK($l, \tau$)
9:      else
10:         $\alpha(\neg\mathtt{T}[\mathtt{S}[l][rnd]][\tau][rnd])$
11:         if $\forall r \in [0, R)$ . $\neg\mathtt{B}[r]$ then ABORT($j$); return
12:         ADDTASKLOCK($l, \tau$); UNBLOCK($l, j$)
13:         DELTASKLOCK($l, \tau$)
14:         if $e[rnd] = 1$ then return
15:         $\mathtt{S}[l][rnd] = \tau$; ADDLOCKTASK($l, \tau$)
```

via prophecy variables instead of function calls. Furthermore, our approach limits verification via execution time, instead of context switches [3][8] or some other means.

Kidd et al. [13] also propose to verify real-time software using sequentialization. They model preemptions using function calls, and do not present any tools or experimental results. Their encoding, while useful for obtaining theoretical results, is too imprecise from a practical verification perspective, since it only uses priorities to limit possible preemptions. Indeed, we have shown [2] that the use of job ordering relations (see Defn. 1) eliminates false warnings compared to an approach that uses priorities only. In contrast, we use prophecy variables, following Lal and Reps [7], limit preemptions using job orderings, and validate our approach empirically.

This paper also extends our earlier work on verifying periodic programs [1][2] by handling PIP locks, executions with blockings, and deadlock detection. This requires a more sophisticated sequentialization (e.g., one that encodes the task resource graph), as well as an iterative algorithm to minimize the number of sequentialization rounds.

Lindstrom et al. [14] have used JavaPathfinder to model check real-time Java programs. Their approach is based on discrete event simulation, and does not: (a) rely on WCET, and (b) consider all possible execution times in the range [0,WCET]. Thus, it is not comparable directly to our approach.

Deadlock detection via sequentialization, explored by Rabinovitz and Grumberg [8], assumes that every deadlock has a wait-free counterexample, i.e., an execution where no thread blocks (except at the end where it deadlocks). This is true if the scheduler is non-deterministic (their situation) but not for periodic programs (this work) where priorities are involved.

Task resource graphs have been used for deadlock detection via runtime analysis [15][16] of concurrent software. However, these projects assume a non-deterministic scheduler, and do not use sequentialization. In addition, some of them [16] over-approximate the TRG and report false deadlocks.

File	T	J	Rn	Vars	Cls	SAT	Result
nxt.bug1a.c	29	15	15	1.4M	4.3M	26	UNSAFE
nxt.bug1b.c	58	15	15	2.5M	7.5M	54	UNSAFE
nxt.bug1c.c	61	15	15	2.6M	8.1M	57	UNSAFE
nxt.ok1.c	746	15	17	2.9M	9.0M	714	SAFE
aso.bug1a.c	73	15	15	2.7M	8.3M	68	UNSAFE
aso.bug1b.c	64	15	15	2.6M	8.0M	59	UNSAFE
aso.bug1c.c	33	15	15	1.7M	5.1M	29	UNSAFE
aso.ok1.c	4148	15	19	3.5M	10.9M	4,088	SAFE
aso.bug2a.c	43	15	15	1.6M	4.9M	39	UNSAFE
aso.bug3a.c	48	15	15	1.7M	5.1M	45	UNSAFE
aso.bug3b.c	35	15	15	1.5M	4.6M	32	UNSAFE
aso.bug3c.c	55	15	15	1.6M	4.9M	52	UNSAFE
aso.ok3.c	879	15	16	1.8M	5.5M	866	SAFE
aso.bug4a.c	63	15	15	2.0M	6.1M	58	UNSAFE
aso.bug4b.c	908	15	16	2.1M	6.4M	898	UNSAFE
aso.ok4.c	3047	15	17	2.2M	6.7M	3,027	SAFE

TABLE I
EXPERIMENTAL RESULTS. T = TOTAL TIME (SEC); J = # OF JOBS; RN = # OF ROUNDS AT COMPLETION; VARS = MAX # OF SAT VARIABLES (IN MILLIONS) PRODUCED BY CBMC; CLS = MAX # OF SAT CLAUSES (IN MILLIONS) PRODUCED BY CBMC; SAT = TOTAL TIME USED BY SAT SOLVER.

V. EXPERIMENTS

Our implementation of PIPVERIF, called REKPIP, builds on REKH [2] . The input to REKPIP is a C program containing the task bodies, and annotations to specify priorities, periods, and WCETs. REKPIP uses CIL [17] for sequentialization, and CBMC [18] to verify the resulting C programs. As in other work [7], REKPIP only allows preemption before access of global variables, without losing soundness. We validated REKPIP on several examples derived from the controller of a LEGO Mindstorms robot[2]. All our experiments were done on a Core-i7 machine with four cores (each running at 2.7GHz) and 8GB of RAM. We know of no tool that is comparable directly with REKPIP. Hence, the main purpose of our experiments is to evaluate the feasibility of our approach.

The Controller. The robot controller consists of three tasks (τ_0, τ_1, τ_2) with priorities $(0, 1, 2)$, periods $(48, 24, 4)$, and WCETs $(12, 12, 1)$, respectively. All tasks arrive at time zero. The system is schedulable, the hyper-period $\mathcal{H}$ is 48, and there are 15 jobs in $\mathcal{C}$. The controller must guarantee that when an obstacle is detected, the robot must move backward and not turn, even if the human operator indicates otherwise. This property, called NOCOLLISION, is expressed by an assertion in the controller code. The assertion involves shared variables accessed by multiple tasks. Hence, appropriate mutual exclusion mechanisms must be used to ensure NOCOLLISION.

The Benchmark. The benchmark consists of a set of examples derived from the controller described above. Example `nxt.ok1.c` is derived from the original version of the controller – τ_2 balances and controls the motion (i.e., speed and direction) of the robot, and receives user commands via bluetooth, τ_1 detects obstacles using a sonar sensor, and τ_0 prints log messages. Task τ_0 does not access shared variables

[2]See http://lejos-osek.sourceforge.net/nxtway_gs.htm for more details.

related to NoCollision, while τ_1 and τ_2 ensure NoCollision by using a PIP lock to protect access to the shared variables. The `nxt.bug1*` examples are buggy variations of `nxt.ok1.c` that use the PIP lock inappropriately.

The `aso.*` examples are derived from a modified version of the controller that we constructed by refactoring out the functionality that receives bluetooth commands from τ_2 to τ_0. Example `aso.ok1.c` uses a single PIP lock to protect the shared variables and ensure NoCollision. The `aso.bug1*` series of examples are buggy variations of `aso.ok1.c` that fail to use the PIP lock appropriately.

Example `aso.bug2a.c` tries to ensure NoCollision without requiring the highest priority τ_2 to do any locking or unlocking (thereby ensuring that τ_2 never blocks). Unfortunately, `aso.bug2a.c` is buggy. In contrast, `aso.ok3.c` achieves this goal successfully by combining of a PIP lock and a transaction-based protocol. The `aso.bug3*` series of examples are buggy variations of `aso.ok3.c` that use either the PIP lock, or the transaction-based protocol inappropriately.

Example `aso.ok4.c` improves on `aso.ok4.c` by using two PIP locks for more fine-grained locking. Examples `aso.bug4a.a` and `aso.bug4b.c` are buggy variations of `aso.ok4.c`. The former performs the fine-grained locking incorrectly (one of the tasks releases a lock prematurely), while the latter has a deadlock (tasks τ_0 and τ_1 attempt to acquire the two PIP locks in opposite order).

Results. Table I summarizes our results. pipVerif produces the correct result for all examples. For `nxt.bug1*.c`, columns Rnds and Jobs are always equal, i.e., counterexamples are detected in the first iteration of pipVerif. For `nxt.ok1.c`, two extra rounds are required to prove safety since there are executions with two blockings between (different jobs of) τ_1 and τ_2 via the PIP lock.

For `aso.bug1*.c`, `aso.bug2*.c` and `aso.bug3*.c`, counterexamples are also detected in the first iteration of pipVerif. However, for `aso.ok1.c` and `aso.ok3.c`, pipVerif goes through several iterations, and only proves safety at rounds greater than the number of jobs. In particular, `aso.ok1.c` requires four extra rounds, while `aso.ok3.c` requires only one extra round.

For `aso.bug4b.c`, the deadlock is detected using one extra round. This is because any execution leading to a deadlock must have at least one job blocking. Suppose that two PIP locks are `L0` and `L1`, τ_0 acquires them in the order `(L0, L1)` and τ_1 acquires them in the opposite order. Then for a deadlock to occur, the following situation must occur – τ_0 gets `L0`, τ_0 is preempted by τ_1, τ_1 gets `L1`, τ_1 tries to get `L0` but is blocked, τ_0 inherits τ_1's priority and resumes execution, τ_0 tries to get `L1`, and we have a deadlock.

In general, verifying an `nxt.*` example is faster than verifying a `aso.*` example. We believe that this is due to the factoring out of complex functionality into a separate task (i.e., thread), which results in increased complexity and a larger statespace. The success of RekPip on these benchmarks indicates that our approach is effective, and advances the state-of-the-art in verifying periodic programs with PIP locks.

VI. Conclusion

We presented an iterative algorithm to verify safety and deadlock freedom of periodic programs. Our algorithm is based on sequentialization – reducing the verification of a concurrent program to that of verifying an equivalent (non-deterministic) sequential program. It extends earlier work in this area by handling synchronization via Priority Inheritance Protocol (PIP) locks, and being able to detect deadlocks. It is also optimal in the sense that it terminates with the minimum number of (sequentialization) rounds needed to prove a periodic program safe, or find a counterexample. Empirical validation of our algorithm indicates its feasibility.

Acknowledgment

Copyright 2013 Carnegie Mellon University and FMCAD, Inc. [3]

References

[1] S. Chaki, A. Gurfinkel, and O. Strichman, "Time-Bounded Analysis of Real-Time Systems," in *FMCAD*, 2011.
[2] S. Chaki, A. Gurfinkel, S. Kong, and O. Strichman, "Compositional Sequentialization of Periodic Programs," in *VMCAI*, 2013.
[3] S. Qadeer and D. Wu, "KISS: Keep It Simple and Sequential," in *PLDI*, 2004.
[4] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE TC*, vol. 39, no. 9, 1990.
[5] M. Jones, "What really happened on Mars?" http://research.microsoft.com/ mbj/Mars_Pathfinder.
[6] "RTEMS Real Time Operating System," http://www.rtems.org.
[7] A. Lal and T. W. Reps, "Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis," in *CAV*, 2008.
[8] I. Rabinovitz and O. Grumberg, "Bounded Model Checking of Concurrent Programs," in *CAV*, 2005.
[9] C. Shih and J. A. Stankovic, "Survey of Deadlock Detection in Distributed Concurrent Programming Environments and Its Application to Real-Time Systems and Ada," University of Massachusetts, Technical report UM-CS-1990-069, 1990.
[10] J. P. Lehoczky, L. Sha, and Y. Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior," in *RTSS*, 1989.
[11] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *JACM*, vol. 20, no. 1, 1973.
[12] L. Cordeiro and B. Fischer, "Verifying multi-threaded software using smt-based context-bounded model checking," in *ICSE*, 2011.
[13] N. Kidd, S. Jagannathan, and J. Vitek, "One Stack to Run Them All - Reducing Concurrent Analysis to Sequential Analysis under Priority Scheduling," in *SPIN*, 2010.
[14] G. Lindstrom, P. C. Mehlitz, and W. Visser, "Model Checking Real Time Java Using Java PathFinder," in *Proc. of ATVA*, 2005.
[15] K. Havelund, "Using Runtime Analysis to Guide Model Checking of Java Programs," in *SPIN*, 2000.
[16] R. Agarwal and S. D. Stoller, "Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables," in *PADTAD*, 2006.
[17] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs," in *CC*, 2002.
[18] E. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs," in *TACAS*, 2004.

[3] This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN AS-IS BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT. This material has been approved for public release and unlimited distribution. DM-0000437