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 PIP Verif, 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 PIP Verif in REKH 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 priori. 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 $C$ and a time bound $t$, we verify that $C$ does not violate a safety property $\phi$ when executed for time $t$ from an initial state $I$. We assume that $t$, $\phi$, 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), 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 \texttt{PIPVERIF}, for time-bounded verification of periodic programs with PIP locks. \texttt{PIPVERIF} maintains a number $R$ of rounds, starting with $R = \text{the total number of jobs}$. In each iteration, it first checks for counterexamples with $R$ rounds. If such a counterexample is detected, \texttt{PIPVERIF} terminates with UNSAFE. Otherwise, it checks for the presence of executions with more than $R$ rounds. If there are no such executions, \texttt{PIPVERIF} terminates with SAFE. Otherwise, it increments $R$ and continues with the next iteration. \texttt{PIPVERIF} is optimal – it terminates with the smallest $R$ required to either find a counterexample, or prove the program safe and deadlock-free.

Second, we extend \texttt{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 this graph is cyclic.

Finally, we implement \texttt{PIPVERIF} by extending REKH [2]. We validate our tool, called \texttt{REKPI}, on a set of examples derived from the controller of a LEGO Mindstorms robot. In each case, \texttt{REKPI} 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.contrib.andrew.cmu.edu/~schaki/README-rekipip.txt.

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 \texttt{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 $\tau$ is a tuple $(I, T, P, C, A)$, 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 $C = \{\tau_0, \ldots, \tau_{N-1}\}$, where $\tau_i = (I_i, T_i, P_i, C_i, A_i)$. 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) $C$ is schedulable. Let $RT_i$ be the response time of $\tau_i$ (i.e., the maximum time taken by any job of $\tau_i$ to complete) computed via Rate Monotonic Schedulability [11] analysis.

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

$$\forall 0 \leq i < N, A_i + RT_i \leq P_i.$$  

(1)

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

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

(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 \texttt{WAIT}(\tau_i, k_i) returns \text{\texttt{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 \text{\texttt{TRUE}}.

**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 $\tau$) is blocked and the (suspended) thread holding $l$ inherits $\tau$’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, $\tau_0$ starts and acquires $l_1$. At time 1, $\tau_1$ preempts $\tau_0$ and acquires $l_2$. At time 2, $\tau_2$ preempts $\tau_1$. At time 3, $\tau_2$ tries to acquire lock $l_3$ and gets blocked. At this point, $\tau_1$ inherits $\tau_2$’s priority (i.e., 2) and resumes execution. At time 4, $\tau_1$ tries to acquire lock $l_4$ and gets blocked. At this point, $\tau_0$ inherits $\tau_1$’s priority (i.e., 2) and resumes execution. At time 5, $\tau_0$ releases lock $l_1$. The inherited priority of $\tau_0$ drops back to its previous priority, viz., 0, and it is preempted by $\tau_1$ which grabs lock

\(^1\) We assume that time is given in some fixed time unit (e.g., milliseconds).
\[ \text{Algorithm 1} \] The overall verification algorithm. Function \text{VERIF\textsc{Rounds}}(P, R) returns UNSAFE if \( P \) has a counterexample (CEX) with \( R \) rounds, \text{INCR\textsc{Rounds}} if \( P \) has no \( R \) round CEXs, but has legal executions with more than \( R \) rounds, and \text{SAFE} otherwise, i.e., if \( P \) has no CEXs with \( R \) or more rounds.

1: \textbf{function} \text{PIPV\textsc{ERIF}}(P) 
2: \hspace{1em} \textbf{R} := |J| 
3: \hspace{1em} \textbf{loop} 
4: \hspace{2em} \textbf{x} := \text{VERIF\textsc{Rounds}}(P, \textbf{R}) 
5: \hspace{2em} \textbf{if} \ \textbf{x} = \text{INCR\textsc{Rounds}} \ \textbf{then} \ \textbf{R} := \textbf{R} + 1 
6: \hspace{2em} \textbf{else return} \textbf{x} 
7: \textbf{function} \text{VERIF\textsc{Rounds}}(P, \textbf{R}) 
8: \hspace{1em} \textbf{if} \ \lfloor S_{\textbf{x}}(P, \textbf{R}) \rfloor \neq 0 \ \textbf{then return} \text{UNSAFE} 
9: \hspace{1em} \textbf{if} \ \lfloor S_{\textbf{R}}(P, \textbf{R}) \rfloor \neq 0 \ \textbf{then return} \text{INCR\textsc{Rounds}} 
10: \hspace{1em} \textbf{else return} \text{SAFE} 

We say that \( P \) is unsafe iff \( \exists k \geq 0. \ b^k \cdot a \in [P] \).

### III. Job-Bounded Verification

Our verification algorithm \text{PIPV\textsc{ERIF}} is based on the idea that any execution \( \rho \) of \( C_P \) is partitioned into schedules \( \text{rounds} \) in the following way: (a) \( \rho \) 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 \( \tau_2 \) gets blocked trying to acquire lock \( l_2 \), round 1 is \([3, 4]\) – when \( \tau_1 \) gets blocked trying to acquire lock \( l_1 \), round 2 is \([4, 7]\) – the end of the first job of \( \tau_2 \), 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 \( b^k \cdot a \) has exactly \( |J| + k \) rounds.

For soundness, \text{PIPV\textsc{ERIF}} must therefore use a sufficiently large number of rounds during sequentialization. To this end, \text{PIPV\textsc{ERIF}} starts with a small number of rounds (specifically, \(|J| \)) 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 \text{PIPV\textsc{ERIF}}. Note that, in each iteration, it invokes \text{VERIF\textsc{Rounds}}(P, \textbf{R}) to check if:

1. \( P \) has a counterexample with \( R \) rounds – in this case \text{VERIF\textsc{Rounds}}(P, \textbf{R}) returns UNSAFE.
2. \( P \) has no counterexample with \( R \) rounds, but has legal executions with more than \( R \) rounds – in this case \text{VERIF\textsc{Rounds}}(P, \textbf{R}) returns \text{INCR\textsc{Rounds}}.
3. \( P \) has no legal executions with more than \( R \) rounds – in this case \text{VERIF\textsc{Rounds}}(P, \textbf{R}) returns \text{SAFE}.

**Correctness of \text{PIPV\textsc{ERIF}}.** \text{PIPV\textsc{ERIF}} is correct because it explores all legal executions of the program and only terminates when a real counterexample is detected (i.e., if \text{VERIF\textsc{Rounds}}(P, \textbf{R}) returns UNSAFE) or when it proves...
that no more legal executions remain to be explored (i.e., if \text{VERIFROUNDS}(P, R) \text{ returns } \text{SAFE}).

A. How \text{VERIFROUNDS} Works

Recall that \text{VERIFROUNDS}(P, R) must satisfy the following specification:

- if \( b^R \cdot a \in \{P\} \) then \( \text{VERIFROUNDS}(P, R) = \text{UNSAFE} \)
- else if \( \forall k > R \cdot \{b^k, b^k \cdot a\} \cap \{P\} = \emptyset \) then \( \text{VERIFROUNDS}(P, R) = \text{SAFE} \)
- else \( \text{VERIFROUNDS}(P, R) = \text{INCROUNDS} \)

Consider the pseudo-code of \text{VERIFROUNDS} shown on the right in Alg. 1. First (line 8), it checks if \( b^R \cdot a \in \{P\} \). To this end, it constructs a sequential program \( S_a(P, R) \) such that:

\[
[S_a(P, R)] = \emptyset \iff b^R \cdot a \notin \{P\} \tag{4}
\]

It then checks if \( [S_a(P, R)] = \emptyset \) using a model checker for sequential programs. Next, to prove that:

\[
\forall k > R \cdot \{b^k, b^k \cdot a\} \cap \{P\} = \emptyset
\]

it relies on the following observation:

\[
\forall k > R \cdot \{b^k, b^k \cdot a\} \cap \{P\} = \emptyset \iff b^{R+1} \notin \{P\}
\]

Therefore (line 9), it constructs a sequential program \( S_b(P, R) \) such that:

\[
[S_b(P, R)] = \emptyset \iff b^{R+1} \notin \{P\} \tag{5}
\]

and checks whether \( [S_b(P, R)] = \emptyset \) via a model checker for sequential programs. Finally, if both the previous checks fail, it returns \text{SAFE} (line 10).

B. Constructing \( S_a(P, R) \)

\( S_a(P, R) \) reduces the bounded concurrent execution of \( C_R \) into a sequential execution with \( R \) rounds. Initially, jobs are allocated (or scheduled) to rounds. Then, jobs are executed sequentially, in the order 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 \( S_a(P, R) \). At the end of \( S_a(P, R) \), 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, \( S_a(P, R) \) 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 \( j = J(\tau, k) \) has a static base priority \( \pi_b(j) \), which is the priority of the corresponding task \( \tau \). 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, \( S_a(P, R) \) must keep track of the inherited priorities of jobs to encode PIP locks.

**Task-Resource Graph.** To compute the inherited priorities of jobs, \( S_a(P, R) \) encodes the transitive closure of the “task resource graph” [9] (TRG) of the program. The TRG \( \Gamma \) 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 \( \Gamma \) 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 \( \Gamma \) as follows. Let \( \Gamma^* \) denote the transitive closure of \( \Gamma \), i.e., \( (x, y) \in \Gamma^* \) iff there is a path from \( x \) to \( y \) in \( \Gamma \). Then,

\[
\pi_i(j) = \max(\{\pi_b(j') \mid (j', j) \in \Gamma^*\})
\]

Thus, if \( j = J(\tau, k) \), then \( \pi_i(j) \) is the maximum of the priorities of all tasks that reach \( \tau \) (including \( \tau \) itself) in \( \Gamma^* \). \( S_a(P, R) \) uses this fact to maintain \( \Gamma^* \) in an online manner – updating it as soon as \( \Gamma \) changes – and compute \( \pi_i(j) \) on demand.

**Detecting Assertion Violations.** In order to model program termination due to an assertion violation, \( S_a(P, R) \) uses an exception mechanism. We use a distinguished global flag to indicate the occurrence of an assertion violation. The flag is initially set to \text{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 \( S_a(P, R) \) only has a legal execution if \( P \) has an execution with an assertion violation.

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

C. Construction of \( S_a(P, R) \)

The structure of \( S_a(P, R) \) 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 \( S_a(P, R) \), followed by its functions.

**Global Variables of \( S_a(P, R) \).** Recall that \( S_a(P, R) \) executes the jobs of \( P \) in the order defined by Defn. 1. Each job \( j \) is assigned a starting and an ending round during scheduling – these are stored in \( \text{start}[j] \) and \( \text{end}[j] \), respectively. Variable \( \text{round} \) stores the current round in which a job is executing. Variable \( B[r] \) 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 \( P[r] \) indicates the priority at which the system is executing at round \( r \) – this equals the
Algorithm 2  The structure of $S_0(P,R)$. Notation: $T$ = set of all tasks; $L$ = set of all PIP locks; $J$ = set of all jobs; $G$ = set of global variables of $P$; $i_g$ = initial value of $g$ according to $C$; ‘$*$’ = non-deterministic value; $\alpha()$ = assume().

```
var rnd, start[ ], end[ ], B[ ], e[ ], v_e[ ], P[], v_p[], S[], L[], T[], LER[], HYP[], OBS[], var g[], v_g

function MAIN( )
1: INITGLOBS(); HYPPer(); CHECKASSUMPS()

function INITGLOBS( )
2: $e[0] := 0; \forall l \in L, S[l][0] := -1$
3: $\forall t1 \in T, t2 \in T, T[t1][t2][0] := 0$
4: $\forall l \in L, l \in L, L[l][l][0] := 0$
5: $\forall g \in G, g[0] := i_g$
6: $\forall r \in [1, R], e[r] := v_e[r] := *; P[r] := v_p[r] := *$
7: $\forall l \in L, r \in [1, R], S[l][r] := v_s[l][r] := *$
8: $\forall t1, t2 \in T, r \in [1, R], T[t1][t2][r] := v_T[t1][t2][r] := *$
9: $\forall l \in L, r \in [1, R], L[l][l][r] := v_L[l][l][r] := *$
10: $\forall r \in [1, R], g[r] := v_g[r] := *$

function HYPPer( )
11: SCHEDULEJOBS()
12: let $j_0 \subset j_1 \subset \ldots \subset j_{n-1}$ be the job ordering from Defn. 1
13: RUNJOB($j_0$); \ldots; RUNJOB($j_{n-1}$)

function RUNJOB($j$)
14: $rnd := start[j]; o := P[rnd]; P[rnd] := \pi_b(j)$
15: if $e[rnd] = 0$ then $T(j)$
16: $cs(j); P[rnd] := o; \alpha(rnd = end[j])$

20: function $T(Job, j)$
21: let $\sigma \equiv if e[rnd] = 1$ then return
22: $T$ is obtained from $T_j$ by replacing each ‘lock(l)’ with:
23: $cs(j); \sigma; \alpha; LOCK(l, j); \sigma$
24: $\alpha; UNLOCK(l, j); \sigma$
25: each ‘assert(s)’ with:
26: $cs(j); \sigma; \alpha; \Rightarrow ABORT(j)$
27: return and each statement ‘$*$’ with:
28: $cs(j); \sigma; \alpha; \Rightarrow UNLOCK(l, j)$

29: function CHECKASSUMPS( )
30: for $r \in [0, R - 1]$ do $\text{let } r' = r + 1$
31: $\alpha; e[r] = e[r'] \Rightarrow P[r] = v_p[r']$
32: $\alpha; e[r] = e[r'] \Rightarrow S[l][r] = v_s[l][r']$
33: $\alpha; e[r] = e[r'] \Rightarrow L[l][l][r] = v_L[l][l][r']$
34: $\alpha; e[r] = e[r'] \Rightarrow v_g[r']$
35: $\forall r \in [0, R], \alpha; B[r] = 0 \Rightarrow e[R - 1] = 1$
36: function ABORT($Job j = J, k$)
37: $e[rnd] := 1$
38: $\forall l \in L, S[l][rnd] = \tau \Rightarrow UNLOCK(l, j)$
```

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

Arrays $S, T$ and $L$ encode the state of the PIP locks and the transitive closure $\Gamma^*$ of the TRG. Specifically, $S[l][r]$ is the priority of the task holding lock $l$ at round $r$. If $l$ is free at round $r$, then $S[l][r] = -1$. For every pair of tasks $(t_1, t_2), T[t_1][t_2][r] = 1$ iff $(t_1, t_2) \in \Gamma^*$ at round $r$. For every task $t$ and lock $l$, $L[l][l][r] = 1$ iff $(t, l) \in \Gamma^*$ at round $r$. Prophecy variables $v_e[l][r], v_p[t_1][t_2][r]$ and $v_g[l][l][r]$ record the guessed initial values of $S[l][r], T[t_1][t_2][r]$ and $L[l][l][r]$, respectively. The values of $S[l][r], T[t_1][t_2][r]$ and $L[l][l][r]$ are updated by jobs executing in round $r$ only.

**Functions of $S_0(P,R)$**. The top-level function is MAIN (see Alg. 2). It initializes all global variables by invoking INITGLOBS (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 $P$. 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 $\subset$ defined by Defn. 1 – by invoking RUNJOB($j$).

In SCHEDULEJOBS (see Alg. 3), line 2 initializes $B$ 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).

$T(j)$ (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, $T(j)$ 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, $T(j)$ checks if an exception has been thrown and terminates the job in this case (using the statement $\alpha$). 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 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 $S_a(P, R)$ given by (4).

**Algorithm 3** The structure of $S_a(P, R)$ continued from Alg. 2.

1: function SCHEDULEJOBS($\tau$

2: \hspace{0.5cm} $\forall r \in [0, R)$. $B[r] := 1; \forall j \in J$. $\text{start}[j] = *; \text{end}[j] = *$

3: \hspace{1cm} // Jobs are sequential

4: \hspace{1cm} $\forall i \in [0, N), \forall k \in [0, J_i)$. $\alpha(0 \leq \text{start}[J(i, k)] \leq \text{end}[J(i, k)] < R)$

5: \hspace{1cm} // Jobs are well-separated

6: \hspace{1cm} $\forall j_1 < j_2, \alpha(\text{end}[j_1] \leq \text{end}[j_2])$; $\forall j_1 < j_2, \alpha(\text{start}[j_1] \leq \text{start}[j_2])$

7: \hspace{1cm} // Jobs are well-nested

8: \hspace{1cm} $\forall j$. $\text{end}[j] = 0$

9: \hspace{1cm} $\forall l$. $\text{Job}[J(l, \tau)]$

10: \hspace{1cm} $\forall \tau$. $\text{DELLOCKTask}(l, \tau)$

11: \hspace{1cm} $\forall \tau$. $\text{DELOCKTask}(l, \tau)$

12: \hspace{1cm} $\forall \tau$. $\text{DELOCKTask}(l, \tau)$

13: \hspace{1cm} $\forall \tau$. $\text{DELLOCKTask}(l, \tau)$

14: \hspace{1cm} $\forall \tau$. $\text{DELLOCKTask}(l, \tau)$

15: \hspace{1cm} $\forall \tau$. $\text{DELLOCKTask}(l, \tau)$

16: \hspace{1cm} $\forall \tau$. $\text{DELLOCKTask}(l, \tau)$

17: \hspace{1cm} $\forall \tau$. $\text{DELLOCKTask}(l, \tau)$

18: \hspace{1cm} $\forall \tau$. $\text{DELOCKTask}(l, \tau)$

19: \hspace{1cm} $\forall \tau$. $\text{DELOCKTask}(l, \tau)$

20: \hspace{1cm} $\forall \tau$. $\text{DELOCKTask}(l, \tau)$

21: function INHERPrioTask($\tau$

22: \hspace{0.5cm} return $\text{MAX}([\tau \cup \{ t | T[t][\tau][\text{rand}] = 1 \})$

23: \hspace{1cm} function UNLOCK(int $l$, Job $J$

24: \hspace{1.5cm} $\alpha(B[\text{rand}] = 1) ; B[\text{rand}] := 0$

25: \hspace{1.5cm} $o := \text{rand} ; \text{rand} := *$

26: \hspace{1.5cm} $\alpha(o < \text{rand} \leq \text{end}[j])$

27: \hspace{1.5cm} $\alpha(P[o] = P[\text{rand}]) ; \alpha(S[t][\text{rand}] = -1)$

28: \hspace{1cm} function LOCK(int $l$, Job $J$

29: \hspace{1.5cm} $\text{if} S[t][\text{rand}] = -1$

30: \hspace{1.5cm} $\text{ADDLOCKTask}(l, \tau)$

31: \hspace{1cm} else

32: \hspace{1.5cm} $\text{ABORT}(\tau)$

33: \hspace{1cm} $\text{return}$

34: \hspace{1cm} $\text{ADDLOCKTask}(l, \tau)$

35: \hspace{1cm} $\text{UNBLOCK}(\tau)$

36: \hspace{1cm} $\text{DELOCKTask}(l, \tau)$

37: \hspace{1cm} $\text{if} e[\text{rand}] = 1$

38: \hspace{1cm} $\text{return}$

39: \hspace{1cm} $\text{ADDLOCKTask}(l, \tau)$

40: \hspace{1cm} $\text{DELOCKTask}(l, \tau)$

41: \hspace{1cm} $\text{ADDLOCKTask}(l, \tau)$

42: \hspace{1cm} $\text{ADDLOCKTask}(l, \tau)$

IV. Related Work

Sequentialization has been used in several projects [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 via prophecy variables instead of
Algorithm 4 The structure of $S_d(P, R)$. We only show functions that are different from $S_o(P, R)$.

1: function $T$(Job $j$)
   
   let $\sigma$ be the statement if $e[rnd] = 1$ then return $T$ is obtained from $T_1$ by replacing each ‘lock($l$)’ with:
   
   2: $CS(j); \sigma; LOCK(l, j); \sigma$
   
   3: each ‘unlock($l$)’ with: $CS(j); \sigma; UNLOCK(l, j)$
   
   4: each ‘assert($x$)’ with: $CS(j); \sigma; \alpha(x)$
   
   5: and each statement ‘$s!' with: $CS(j); \sigma; s[g \leftarrow g[rnd]]$
   
   6: function $LOCK$($int l$, Job $j = (\tau, k)$)
   
   7: if $S[l][rnd] = -1$ then
   
   8: $S[l][rnd] = \tau$; ADDLOCKTask($l, \tau$)
   
   9: else
   
   10: $\alpha(-T[S[l][rnd]][\tau][rnd])$
   
   11: if $\forall r \in [0, R) \cdot \neg B[r]$ then ABORT($j$); return
   
   12: ADDTaskLOCK($l, \tau$); UNBLOCK()
   
   13: DELTaskLOCK($l, \tau$)
   
   14: if $e[rnd] = 1$ then return
   
   15: $S[l][rnd] = \tau$; ADDLOCKTask($l, \tau$)

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 use 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.

Deadlock detection via sequentialization was also explored by Rabinovitz and Grumberg [8]. They assume 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. For the interested reader, a proof of this claim is in the appendix.

Task resource graphs have been used for deadlock detection via runtime analysis [14][15] of concurrent software. However, these projects assume a non-deterministic scheduler, and do not use sequentialization. In addition, some of them [15] over-approximate the TRG and report false deadlocks.

<table>
<thead>
<tr>
<th>File</th>
<th>J</th>
<th>Rn</th>
<th>Vars</th>
<th>Cls</th>
<th>SAT</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>nxt.bug1a.c</td>
<td>29</td>
<td>15</td>
<td>1.4M</td>
<td>4.3M</td>
<td>26</td>
<td>UNSAFE</td>
</tr>
<tr>
<td>nxt.bug1b.c</td>
<td>58</td>
<td>15</td>
<td>2.5M</td>
<td>7.5M</td>
<td>54</td>
<td>UNSAFE</td>
</tr>
<tr>
<td>nxt.bug1c.c</td>
<td>61</td>
<td>15</td>
<td>2.6M</td>
<td>8.1M</td>
<td>57</td>
<td>UNSAFE</td>
</tr>
<tr>
<td>nxt.ok1.c</td>
<td>446</td>
<td>15</td>
<td>2.9M</td>
<td>9.0M</td>
<td>714</td>
<td>SAFE</td>
</tr>
<tr>
<td>aso.bug1a.c</td>
<td>73</td>
<td>15</td>
<td>2.7M</td>
<td>8.3M</td>
<td>68</td>
<td>UNSAFE</td>
</tr>
<tr>
<td>aso.bug1b.c</td>
<td>64</td>
<td>15</td>
<td>2.6M</td>
<td>8.0M</td>
<td>59</td>
<td>UNSAFE</td>
</tr>
<tr>
<td>aso.bug1c.c</td>
<td>53</td>
<td>15</td>
<td>1.7M</td>
<td>5.1M</td>
<td>29</td>
<td>UNSAFE</td>
</tr>
<tr>
<td>aso.ok1.c</td>
<td>4108</td>
<td>15</td>
<td>5.5M</td>
<td>10.0M</td>
<td>4.088</td>
<td>SAFE</td>
</tr>
<tr>
<td>aso.bug2a.c</td>
<td>43</td>
<td>15</td>
<td>1.6M</td>
<td>4.9M</td>
<td>39</td>
<td>UNSAFE</td>
</tr>
<tr>
<td>aso.bug2b.c</td>
<td>48</td>
<td>15</td>
<td>1.7M</td>
<td>5.1M</td>
<td>45</td>
<td>UNSAFE</td>
</tr>
<tr>
<td>aso.bug2c.c</td>
<td>35</td>
<td>15</td>
<td>1.5M</td>
<td>4.6M</td>
<td>32</td>
<td>UNSAFE</td>
</tr>
<tr>
<td>aso.bug3c.c</td>
<td>55</td>
<td>15</td>
<td>1.6M</td>
<td>4.9M</td>
<td>52</td>
<td>UNSAFE</td>
</tr>
<tr>
<td>aso.ok2.c</td>
<td>879</td>
<td>15</td>
<td>1.8M</td>
<td>5.3M</td>
<td>866</td>
<td>SAFE</td>
</tr>
<tr>
<td>aso.bug4a.c</td>
<td>63</td>
<td>15</td>
<td>2.0M</td>
<td>6.1M</td>
<td>58</td>
<td>UNSAFE</td>
</tr>
<tr>
<td>aso.bug4b.c</td>
<td>908</td>
<td>16</td>
<td>2.1M</td>
<td>6.4M</td>
<td>898</td>
<td>SAFE</td>
</tr>
<tr>
<td>aso.ok1.b</td>
<td>3047</td>
<td>15</td>
<td>2.2M</td>
<td>6.7M</td>
<td>3.027</td>
<td>SAFE</td>
</tr>
</tbody>
</table>

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; $CIs =$ max # of SAT clauses (in millions) produced by CBMC; $SAT =$ total time used by SAT solver.

V. Experiments

We implemented pipVerif by extending REKIP [2]. We call our tool REKPip. The input to REKPip is a C program containing the task bodies, and annotations to specify priorities, periods, and WCETs. REKPip uses CIL [16] for sequentialization, and CBMC [17] 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 robot2. All our experiments were done on a Core-i7 machine with four cores (each running at 2.7GHz) and 8GB of RAM. Since pipVerif is the only verification tool known to us that deals with periodic programs and PIP locks, the main purpose of our experiments is to evaluate the feasibility of our approach.

The Controller. The robot controller consists of three tasks ($\tau_0$, $\tau_1$, $\tau_2$) with priorities $(0, 1, 2)$, periods $(48, 24, 4)$, and WCETs $(12, 12, 1)$, respectively. All tasks arrive at time zero. Note that the system is schedulable, the hyper-period $H$ is 48, and there are 15 jobs in $C_H$. The key property that the controller must guarantee is that when an obstacle is detected, the robot must move backward and not turn, even if the human operator indicates otherwise. This property, which we call NOCOLLISION, is expressed by an assertion in the controller code. The assertion involves shared variables that all three tasks access. 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 – $\tau_2$ balances and controls the motion (i.e., speed and direction) of the robot, and receives user commands via Bluetooth. $\tau_1$ detects obstacles using a sonar sensor, and $\tau_0$ prints log messages. Task $\tau_0$ does not access shared variables.

related to NoCollision, while \( \tau_1 \) and \( \tau_2 \) ensure NoCollision by using a PIP lock to protect access to the shared variables. The \text{nxt.bug1*.c} examples are buggy variations of \text{nxt.ok1.c} that use the PIP lock inappropriately.

The \text{aso.*) examples are derived from a modified version of the controller that we constructed by refactoring out the functionality that receives Bluetooth commands from \( \tau_2 \) to \( \tau_7 \). Example \text{aso.ok1.c} uses a single PIP lock to protect the shared variables and ensure NoCollision. The \text{aso.bug1*} series of examples are buggy variations of \text{aso.ok1.c} that fail to use the PIP lock appropriately.

Example \text{aso.bug2a.c} tries to ensure NoCollision without requiring the highest priority \( \tau_2 \) to do any locking or unlocking (thereby ensuring that \( \tau_2 \) never blocks). Unfortunately, \text{aso.bug2a.c} is buggy. In contrast, \text{aso.ok3.c} achieves this goal successfully by combining of a PIP lock and a transaction-based protocol. The \text{aso.bug3*} series of examples are buggy variations of \text{aso.ok3.c} that use either the PIP lock, or the transaction-based protocol inappropriately.

Example \text{aso.ok4.c} improves on \text{aso.ok4.c} by using two PIP locks for more fine-grained locking. Examples \text{aso.bug4a.a} and \text{aso.bug4b.c} are buggy variations of \text{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 \( \tau_0 \) and \( \tau_1 \) attempt to acquire the two PIP locks in opposite order).

**Results.** Table 1 summarizes our results. PIPVERIF produces the correct result for all examples. For \text{nxt.bug1*.c}, columns Rnds and Jobs are always equal, i.e., counterexamples are detected in the first iteration of PIPVERIF. For \text{nxt.ok1.c}, two extra rounds are required to prove safety since there are executions with two blockings between (different jobs of) \( \tau_1 \) and \( \tau_2 \) via the PIP lock.

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

For \text{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 \( L_0 \) and \( L_1 \), \( \tau_0 \) acquires them in the order \((L_0, L_1)\) and \( \tau_1 \) acquires them in the opposite order. Then for a deadlock to occur, the following situation must occur: \( \tau_0 \) gets \( L_0 \), \( \tau_0 \) is preempted by \( \tau_1 \), \( \tau_1 \) gets \( L_1 \), \( \tau_1 \) tries to get \( L_0 \) but is blocked, \( \tau_0 \) inherits \( \tau_1 \)'s priority and resumes execution, \( \tau_1 \) tries to get \( L_1 \), and we have a deadlock.

In general, verifying an \text{nxt.*) example is faster than verifying a \text{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 state space. The success of REK/Pip 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**

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.

**References**


\*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-000437
Rabinovitz et al. [8] proposed an approach based on sequentialization to verify reachability properties of concurrent programs. They leveraged the following result in their encoding.

**Lemma 1:** Let $P$ be a concurrent program. Two executions $\rho$ and $\rho'$ of $P$ are said to be mutex-free equivalent, denoted by $\rho \approx_w \rho'$, iff they have the same states ignoring the internal implementations of mutexes (i.e., locks). A wait-free execution of $P$ is one where no thread blocks when trying to acquire a lock. Then, for every non-wait-free execution $\rho$ of $P$ there exists a wait-free execution $\rho'$ of $P$ such that $\rho \approx_w \rho'$.

In effect, Lemma 1 was used by Rabinovitz et al. [8] to only allow wait-free executions in the sequentialized program, without losing soundness. However, we now show that Lemma 1 does not hold for periodic programs with PIP locks, and therefore only allowing wait-free executions in the sequentialized program would be unsound in this case.

```c
void t1()
{
1: lock();
2: x = 1;
3: unlock();
}

void t2()
{
4: x = 2;
5: lock();
6: assert(x == 2);
7: unlock();
}
```

Fig. 2. Example of a periodic program to show that considering only wait-free paths for verification of reachability properties is unsound.

Consider the periodic program $P$ with two tasks $t_1$ and $t_2$ having low and high priority, respectively. The bodies of the two tasks are given, respectively, by the functions `t1()` and `t2()` shown in Figure 2. Assume that the global variable $x$ is initialized to zero. Then, the assertion is violated by the following non-wait-free execution of $P$: $(1, 4, 5, 2, 3, 6)$. However, no wait-free execution of $P$ leads to an assertion violation. Note that while the wait-free execution $(4, 1, 2, 3, 5, 6)$ does lead to an assertion violation, it is not possible under priority-based scheduling, and hence is not a legal execution of $P$. 