Time-Bounded Analysis of Real-Time Systems

Sagar Chaki
SEI/CMU
chaki@sei.cmu.edu

Arie Gurfinkel
SEI/CMU
arie@cmu.edu

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

***Abstract*—Real-Time Embedded Software (RTES) constitutes an important sub-class of concurrent safety-critical programs. We consider the problem of verifying functional correctness of periodic RTES, a popular variant of RTES that execute *periodic tasks* in an order determined by *Rate Monotonic Scheduling* (RMS). A computational model of a periodic RTES is a finite collection of terminating tasks that arrive periodically and must complete before their next arrival.**

We present an approach for *time-bounded* verification of safety properties in periodic RTES. Our approach is based on sequentialization. Given an RTES $\mathcal{C}$ and a time-bound $\mathcal{W}$, we construct (and verify) a sequential program $\mathcal{S}$ that over-approximates all executions of $\mathcal{C}$ up to time $\mathcal{W}$, while respecting priorities and bounds on the number of preemptions implied by RMS. Our algorithm supports partial-order reduction, preemption locks, and priority locks. We implemented our approach for C programs, with properties specified via user-provided assertions. We evaluated our tool on several realistic examples, and were able to detect a subtle concurrency issue in a robot controller.

I. Introduction

Real-Time Embedded Software (RTES) is an important sub-class of concurrent safety-critical programs. They play a crucial role in controlling systems ranging from airplanes and cars, to infusion pumps and microwaves. We are increasingly reliant on these *cyber-physical* systems to maintain our modern technology-driven way of life. As such, verifying the correct operation of RTES is an important and open challenge. Addressing this challenge is the subject of our paper.

Specifically, we focus on systems that receive as input a collection of *periodic tasks*, where each task τ_i has, among other things, a terminating task body T_i and a period P_i. The tasks are *prioritized* in a Rate-Monotonic [1] fashion, which means that tasks with shorter periods have higher priorities. We call such an input pattern a *periodic program*.

A periodic program $\mathcal{C}$ is executed by running its tasks periodically and concurrently with asynchronous *priority-sensitive* interleaving. Thus, at each scheduling point, the active task with the highest priority is selected for execution. A task τ_i becomes inactive at the end of its body T_i, and is reactivated after P_i time has passed since its last activation. A single execution of a task body is called a *job*. It is convenient to view an execution of $\mathcal{C}$ as an asynchronous priority-sensitive interleaving of the jobs statements, where the statements arise from the infinite job streams corresponding to the periodic execution of the task bodies.

Periodic programs constitute an important fragment of RTES that interact with the physical world. In particular, the task periods are dictated by the physical environment and the underlying control algorithms. Consider, for example, the nxt/OSEK-based [2] LEGO MINDSTORM robot controller. It has three periodic tasks: a *balancer*, with a 4 millisecond (ms) period, maintains the balance of the robot; *obstacle*, with a 50 ms period, monitors a sonar sensor to detect obstacles; and *bluetooth*, with a 100 ms period, monitors a bluetooth link for remote commands from the user. Another example is a generic avionic mission system that was described in [3]. It includes 10 periodic tasks, including weapon release (10 ms), radar tracking (40 ms), target tracking (40 ms), aircraft flight data (50 ms), display (50 ms) and steering (80 ms). Other examples of periodic programs include phase-array radars and aircraft collision-avoidance systems.

These examples demonstrate the fact that periodic programs are used for developing a wide range of RTES that interact with the physical world, and play an important role in the correct operation of safety-critical systems. Statically predicting behavior – by verifying logical and timing properties of periodic programs – is a problem of great practical relevance.

Despite a wide body of work, the state-of-the-art in verification of real-time and concurrent programs does not address logical properties of periodic programs (with the exception of the recent work of Kidd et al. [4], which we discuss in Sec. VIII). On one hand, techniques for verifying properties of timed systems [5], [6] are based on Timed Automata [7]. They abstract away significantly the behavior (i.e., control- and data-flow) of target systems, and, therefore, are unsuitable for analyzing logical properties. On the other hand, approaches for concurrent software verification (e.g., [8]) employ a non-deterministic scheduler model (i.e., tasks do not have priorities or periods), and thus cannot handle the execution semantics of periodic programs. Against this backdrop, our main contribution is the development and evaluation of an approach to verify logical properties of periodic programs.

Specifically, we present an approach for *time-bounded* verification of safety properties of periodic programs. The inputs are: (i) a periodic program $\mathcal{C}$; (ii) a safety property expressed via an assertion A embedded in $\mathcal{C}$, (iii) an initial condition $Init$ of $\mathcal{C}$, and (iv) a time bound $\mathcal{W}$. Time-bounded verification can be seen as an analogue of Bounded Model Checking (BMC) for RTES, since time is a natural way to bound an execution of a periodic program for the purpose of verification.

Our solution for the time-bounded verification problem is based on *sequentialization* – reducing verification of a concurrent program to verification of a sequential program. It is inspired by work on sequentialization for *Context-Bounded*

Analysis [8], [9], [10], [11], [12] (CBA) and Bounded Model Checking [13] (BMC). A key distinguishing aspect of our work is that instead of bounding the number of context switches (as in CBA), or the number of execution steps (as in BMC), the input time bound $\mathcal{W}$ translates in our model to a bound on the number of jobs. This is a natural consequence of the fact that tasks are periodic and, therefore, are activated a finite number of times within $\mathcal{W}$. Our solution also handles two types of locks used commonly in periodic programs, and incorporates two forms of partial-order reduction aimed at reducing analysis time.

We have implemented our solution in a tool, called REK. Our tool is able to verify periodic programs implemented in C where tasks communicate via shared variables and synchronize via locks. We have used it to verify several variants of a nxt/OSEK-based [2] LEGO MINDSTORM robot controller. In some instances, we found subtle concurrency issues in the controller using our tool. We have also evaluated our tool on several custom versions of the reader-writer protocol.

The rest of the paper is structured as follows: in the next section we formally define a periodic program and its semantics. In Sec. III, we describe time-bounded and job-bounded abstractions. In Sec. IV, we describe our encoding method. In Sec. V, we extend our model with locks, and in Sec. VI describe partial-order reduction. In Sec. VII, we describe our case study and experimental results. Finally, we discuss related work in Sec. VIII, and conclude in Sec. IX.

II. PRELIMINARIES

A *task* τ is a tuple $\langle I, T, P, C, A\rangle$, where I is a task identifier, T – a bounded procedure (i.e., no unbounded loops or recursion) called the task body, P – a period, C – the worst case execution time of T, and A, called the release time, is the time at which the task is first enabled[1]. An N-task *periodic program* $\mathcal{C}$ is a set $\{\tau_0, \ldots, \tau_{N-1}\}$ of N tasks. For simplicity, we assume that the id of task τ_i is i.

In this paper, we restrict the priorities of the tasks to be *rate-monotonic* – tasks with smaller period have higher base priority. For simplicity, we assume that the index of a task represents its base priority. Thus, a task with a lower id has a lower base priority (and higher period).

A periodic program is executed by running each task periodically, starting at the release time. For $k \geq 0$ the k-th job of τ_i becomes enabled at time $A_i^k = A_i + k \times P_i$. The execution is asynchronous and priority-sensitive – at each point the CPU is given to an enabled task with the highest priority. Priorities can change dynamically, but must avoid *priority inversion* – when a low base priority task preempting a higher base priority task. This is known, somewhat misleadingly, as a *fixed-priority preemptive scheduling*.

Formally, the semantics of an N-task periodic program $\mathcal{C} = \{\tau_0, \ldots, \tau_{N-1}\}$ is the asynchronous concurrent program:

$$\|_{i=0}^{N-1}\ k_i := 0\ ;\ \textbf{while}(\textsc{Wait}(\tau_i, k_i))\ (T_i\ ;\ k_i := k_i + 1) \quad (1)$$

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

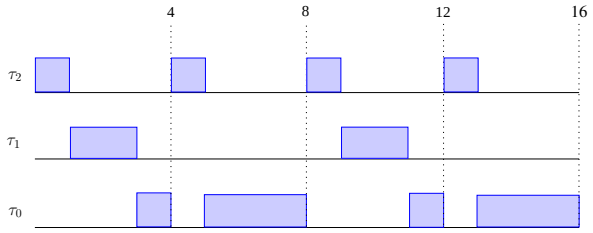

Fig. 1. A schedule of three tasks from Example 1.

where k_i is a numeric variable and $\textsc{Wait}(\tau_i, k_i)$ returns FALSE if the current time is greater than $A_i^{k_i}$, and otherwise it disables τ_i until the time is $A_i^{k_i}$ and then returns TRUE.

An execution of each task body T_i in (1) is called a *job*. A job's *arrival* is the time when it becomes enabled (i.e., $\textsc{Wait}(\tau_i, k)$ in (1) returns TRUE); *start* and *finish* are the times when its first and last instructions are executed, respectively; *response time* is the difference between its finish and arrival times. The *response time* of a task is the maximum of response times of all of its jobs in all possible executions.

Note that WAIT in (1) returns TRUE if a job has finished before its next period. If this is always the case, i.e., WAIT never returns FALSE, then the program is called *schedulable*.

Formally, a periodic program $\mathcal{C}$ is schedulable if for each task τ_i, the response time RT_i is less than the period P_i. Response times are computed using Rate Monotonic Analysis (RMA) [14]. For a periodic program $\mathcal{C} = \{\tau_0, \ldots, \tau_{N-1}\}$, the response time RT_i of task τ_i is the smallest solution to the following equation

$$RT_i = C_i + \sum_{i<k<N} \lceil \frac{RT_i}{P_k} \rceil \cdot C_k\ . \quad (2)$$

Intuitively, the response time of a task is equal to its worst-case execution time plus the time taken by all higher-priority tasks that preempted it. RT_i is computed by solving (2) iteratively starting with $RT_i = C_i$ [14]. Note that for the highest-priority task the response time RT_{N-1} is its execution time C_{N-1}.

Example 1 *Consider the task set:*

Task	C_i	P_i
τ_2	*1*	*4*
τ_1	*2*	*8*
τ_0	*8*	*16*

Solving (2) gives us $RT_2 = 1$, $RT_1 = 3$ *and* $RT_0 = 16$. *A schedule demonstrating these values is shown in Fig. 1.*

In this paper, we are interested in logical properties of periodic programs. We assume that any program we analyze meets its basic timing constraints. For that reason, we only deal with schedulable periodic programs.

III. TIME-BOUNDED PERIODIC PROGRAMS

In this section, we present time-bounded semantics of periodic programs and define a job-bounded abstraction that is the formal foundations of our verification technique.

Given a periodic program $\mathcal{C} = \{\tau_0, \ldots, \tau_{N-1}\}$ and a time-bound $\mathcal{W}$, the time-bounded program $\mathcal{C}_\mathcal{W}$ executes like $\mathcal{C}$ for time $\mathcal{W}$ and then terminates. We assume that $\mathcal{W}$ is divisible by the period of each task. Furthermore, we assume that the first job of each task finishes before its period, i.e., $A_i \leq P_i - RT_i$. Under these assumptions, the time bound imposes a natural limit on the number of jobs J_i of each task:

$$J_i = \frac{\mathcal{W}}{P_i} \ . \tag{3}$$

Therefore, the semantics of $\mathcal{C}_\mathcal{W}$ is equivalent to the asynchronous concurrent program:

$$\|_{i=0}^{N-1}\ k_i{:=}0; \textbf{while}(k_i < J_i \wedge \textsc{Wait}(\tau_i, k_i))\ (T_i; k_i{:=}k_i{+}1)\ . \tag{4}$$

This is analogous to the semantics of $\mathcal{C}$ in (1) except that each task τ_i executes J_i jobs.

Job-bounded Abstraction. Since we are interested in logical properties, we need to abstract the absolute time in (4) with relative order of execution. A simple abstraction is to interpret WAIT as a non-deterministic delay, effectively replacing a time-bound with a job-bound. We further refine this abstraction using the following observation about RMA (2). Let τ_i and τ_j be two tasks of a schedulable periodic program $\mathcal{C}$ such that $i < j$. Then RMA defines the *preemption bound* of i by j, written PB_i^j, as follows

$$PB_i^j = \lceil \frac{RT_i}{P_j} \rceil \ . \tag{5}$$

That is, PB_i^j is an upper bound on the number of times τ_j can preempt τ_i. Thus, in addition to treating WAIT as a non-deterministic delay, we only allow for a job of task j to preempt a job of task i if $j > i$ and it does not violate the preemption bound PB_i^j. In other words, we only schedule at most PB_i^j jobs of τ_j while a job of τ_i is active. We call this the job-bounded abstraction of $\mathcal{C}_\mathcal{W}$ and denote it by $\mathcal{C}_{J(\mathcal{W})}$.

Theorem 1 (Soundness of Job-bounded Abstraction) *For a periodic program $\mathcal{C} = \{\tau_0, \ldots, \tau_{N-1}\}$ and a time-bound $\mathcal{W}$ s.t. $\forall i \cdot (A_i \leq P_i - RT_i) \wedge (\mathcal{W} | P_i)$, every execution of $\mathcal{C}_\mathcal{W}$ is also an execution of $\mathcal{C}_{J(\mathcal{W})}$.*

Obviously, job-bounded abstraction is incomplete, i.e., $\mathcal{C}_{J(\mathcal{W})}$ may have more executions than $C_\mathcal{W}$. The primary reason being that the preemption bounds are only upper bounds since they are computed from the worst-case execution times, and rounded upwards in (5). The incompleteness due to rounding up is shown by the following example.

Example 2 *Let $\mathcal{C} = \{\tau_0, \tau_1\}$ such that $P_0 = 25$, $RT_0 = 22$ and $P_1 = 10$, $RT_1 = 10$, and $\mathcal{W} = 100$. Then, $PB_0^1 = 3$ and, therefore, in $\mathcal{C}_{J(\mathcal{W})}$ two consecutive jobs of τ_0 can be preempted three times, each, by jobs of τ_1. However, in $\mathcal{C}_\mathcal{W}$ two consecutive jobs of τ_0, taken together, can be preempted at most five times by jobs of τ_1.*

In the next section, we give an algorithm for constructing and verifying the job-bounded abstraction $\mathcal{C}_{J(\mathcal{W})}$ from a periodic program $\mathcal{C}$.

IV. SEQUENTIALIZATION OF A PERIODIC PROGRAM

We use a two-step approach to verify an N-task periodic program $\mathcal{C} = \{\tau_0, \ldots, \tau_{N-1}\}$ under a time bound $\mathcal{W}$. The first step, sequentialization, outputs a non-deterministic sequential program with assume statements $\mathcal{S}$, as shown in Alg. 1. The program $\mathcal{S}$ is semantics preserving w.r.t. $\mathcal{C}_{J(\mathcal{W})}$ (see Theorem 2). The second step is the verification of $\mathcal{S}$ with an off-the-shelf program verifier. In the rest of this section, we focus on sequentialization.

A. Sequentialization: Intuition

Our key insight is that any execution π of $\mathcal{C}$ can be partitioned into scheduling *rounds* in the following way: (a) π 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). For example, the bounded execution shown in Fig. 1 is partitioned into 7 rounds as follows: round 0 is the time interval $[0, 1]$ – the end of the first job of τ_2, round 1 is $[1, 3]$ – the end of the first job of τ_1, round 2 is $[3, 5]$ – the end of the second job of τ_2 (note that there is only one job of τ_0 and it ends at time 16), round 3 is $[5, 9]$, etc.

Observe that in each round, the tasks are executed sequentially in the order of their priority starting with the task of the lowest priority. Furthermore, a bounded execution in which exactly k jobs start and end has exactly k rounds.

The basic idea of our sequentialization is to reduce a bounded concurrent execution with k jobs into a sequential execution with k rounds. Initially, jobs are allocated (or scheduled) to rounds. Then, each round is executed independently (and sequentially) starting from a guessed initial state. Finally, we check for each $0 \leq i < (k-1)$, that the guessed initial state at round $i+1$ is the final state of round i.

Our sequentialization is inspired by the work of Lal and Reps [9], but differs from it in several significant ways. First, we deal with periodic programs and not non-deterministically scheduled concurrent programs. Second, since we are not exploring non-deterministic schedules, we use a more refined scheduling scheme than the non-deterministic round-robin used in [9]. Third, we partition each task (or, correspondingly, a thread in [9]) into jobs and preserve *all* executions in which *all* jobs terminate. In contrast, [9] only preserves executions with a bounded number of thread-preemptions. Finally, we take into account that preemption between tasks must respect priorities and preemption bounds.

B. Sequentialization: Details

The sequential program $\mathcal{S}$ starts in MAIN (see Alg. 1). It first allocates jobs to rounds (line 5), initializes global variables (lines 6–8), executes all jobs of all tasks sequentially starting with the first job of the lowest priority task (lines 9–13), and finally checks correctness of guessed variables and assertions (lines 14–15).

Job scheduling. A job schedule is a pair of mappings $start$ and end that map each job of each task to a starting and ending round, respectively. We say that a job schedule is legal iff it satisfies the following three properties: (a) jobs are sequential

– for a given task, for any pair $i < j$, job i starts and finishes (i.e., precedes) job j, and (b) jobs are well-nested – if a higher and a lower priority jobs overlap, then the higher priority job must start and end within the rounds of the lower priority one, and (c) jobs respect preemption bounds – no more than $PB_{t_i}^{t_j}$ jobs of task t_j are scheduled inside any job of task t_i.

SCHEDULEJOBS fills arrays $start$ and end with a non-deterministic *legal* job schedule. Line 24 ensures that the jobs are sequential, while line 25 ensures that they are well-nested, and line 26 ensures that they respect preemption bounds. To understand line 26, suppose that $PB_{t_1}^{t_2} = 3$, $j_2 = 4$ and that j_2 is nested in j_1. Then, to satisfy $PB_{t_1}^{t_2}$, we require that job 1 of t_2 has ended before j_1 started. Otherwise, because the jobs are sequential and well-nested, j_1 contains jobs 1, 2, 3, and 4 of t_2 – that is, it is preempted by more than three t_2 jobs.

Global variables. For every global variable g, we create two arrays $g[\,]$ and $v_g[\,]$ such that $g[i]$ is the value of g in round i, and $v_g[i]$ is the guess of the initial value of g in round i. The element $g[0]$ is initialized in line 7 to the user-specified initial value i_g, and each other elements $g[r]$ is assigned the corresponding initial guess $v_g[r]$ in line 8.

In addition, a variable rnd tracks the current round, job tracks the current job, and $endRnd$ is the scheduled end round of the current job.

Tasks. For each task t, MAIN uses a modified version $\hat{T}_t$ obtained from the original task body T_t by preceding each statement st with a call to CS to emulate a preemption, and replacing every global variable g in st with $g[rnd]$. This is based on an assumption, without loss of generality, that each global variable g is explicitly loaded and stored, i.e., g only appears in statements of the form $g := l$ or $l := g$, where l is a local variable.

Preemption. CS models a preemption (a context switch to a higher priority task) by increasing non-deterministically the value of rnd to the round in which this task resumes execution. However, not every round between the start and end of the current job is legitimate. Line 21 ensures that the execution is not resumed in a round used by a higher-priority task. CS returns TRUE iff a preemption has occurred. This value is used later in Sec. VI.

Prophecy-Check. Line 14 ensures that the value of each global variable at the end of a round is equal to its guessed value at the beginning of the next round.

Assertions. Assertions need special handling. They can only be checked after the guesses have been validated via Prophecy-Check. Without loss of generality, we assume a single call to assert in the body of each task. We use an array *localAssert* that maps a task and a job to the value of the assertion in it. The element *localAssert*$[t][j]$ is initialized to TRUE (line 6), set to the value of the asserted expression (line 29), and asserted to be TRUE (line 15) after the Prophecy-Check.

Our sequentialization procedure is semantic preserving as expressed in the following theorem.

Theorem 2 *Let $\mathcal{C}$ be a periodic program and $\mathcal{W}$ a time-bound satisfying the conditions in Theorem 1. Then, for every execution π of $\mathcal{C}_{J(W)}$ that violates an assertion in job j of task t, there is a corresponding execution π' of the sequential program $\mathcal{S}$ that violates localAssert$[t][j]$, and vice versa.*

V. LOCKS

We support two types of locks: preemption locks and priority ceiling locks. These locks are common in periodic programs because they are non-blocking (acquiring a lock always succeeds) and avoid common pitfalls such as priority inversion and deadlocks.

A periodic program has a single preemption lock pl. Acquiring pl disables the scheduler, preventing all priority-based preemptions. Releasing pl re-enables the scheduler. An example of such a mechanism is the `taskLock` / `taskUnlock` routines in VxWorks [15].

Priority ceiling locks, or *priority locks* for short, are based on dynamically raising the priority of the current job. Each priority lock lck is associated with a fixed priority LOCKPRIORITY(lck), which is given to a task if it acquires lck. It is illegal for a task with current priority p to acquire a priority lock lck such that LOCKPRIORITY(lck) is less than p. Releasing a lock restores the priority. Priority locks are used in the Highest Locker-Priority (HLP) protocol [16], where the priority of a resource r is as high as the priority of any task accessing r. This guarantees mutual exclusion (assuming there is only one CPU) while avoiding blocking, deadlocks, and priority inversion. Multiple priority locks must be acquired in increasing order of priority, but can be released in an arbitrary order. We now show our encoding of these locks.

Preemption locks. The preemption lock pl is modeled by introducing a Boolean variable $lock$ into $\hat{T}_t$. Specifically, $lock =$ TRUE iff the current task has acquired pl. The variable is FALSE initially and is reset to FALSE at the end of a job. Finally, calls to CS are conditioned by $lock =$ FALSE.

Priority locks. To model priority locks, we need to model the dynamic priority of each task. Let BASEPRIORITY(t) be a function that returns the base priority of task t. We introduce an array $priority$ such that for every round r, $priority[r]$ is the priority of the task currently executing in round r. The array is maintained by the function $\hat{T}_t$-WRAPPER (shown in Alg. 2) that wraps the body $\hat{T}_t$ of each task t. The wrapper function saves the current priority (line 2), ensures that it is below the base priority of the current task (line 3), raises the priority to the priority of the current task (line 4), executes the task body (line 5), and finally resets the priority (line 6). Note that the task body is only executed if the task has higher priority than the current dynamic one, and that the priority can be raised further inside the task body itself.

We model priority locks with two functions GETLOCK and RELEASELOCK shown in Alg. 2. The set of all locks held by a task is maintained in a set $lockSet$ that is local to each task. Information about locks of other tasks is propagated through priorities. Acquiring a lock (GETLOCK) raises the dynamic priority to the one of the lock, releasing a lock (RELEASELOCK) resets the priority to the base priority of the task or to the priority of the highest lock in the $lockSet$ in

Algorithm 1 A sequential program $\mathcal{S}$ for a periodic program $\mathcal{C}$ bounded by time $\mathcal{W}$. Notation: T is the set of tasks of $\mathcal{C}$; G is the set of global variables of $\mathcal{C}$; $\mathsf{J}(t)$ is the set of jobs of task t; $R = \sum_{t \in \mathsf{T}, j \in Job(t)} |\mathsf{J}(t)|$ is the number of rounds, last(t, j) is true iff j is the last job of $t \in \mathsf{T}$; for $t_i, t_j \in \mathsf{T}$, $t_i < t_j$ is true iff t_i is of lower priority than t_j; '$*$' is a non-deterministic value.

1: **var** $rnd, job, endRnd, start[][], end[][]$
2: $\forall g \in \mathsf{G} \cdot$ **var** $g[], v_g[]$
3: **var** *localAssert*$[][]$
4: **function** MAIN( )
5: SCHEDULEJOBS()
6: $\forall t \in \mathsf{T}, j \in \mathsf{J}(t) \cdot$ *localAssert*$[t][j] :=$ TRUE
7: $\forall g \in \mathsf{G} \cdot g[0] := i_g$
8: $\forall g \in \mathsf{G} \forall r \in [1, R) \cdot g[r] := v_g[r]$
9: **for** $t \in \mathsf{T}, job \in \mathsf{J}(t)$ **do**
10: $rnd := start[t][job]$
11: $endRnd := end[t][job]$
12: $\hat{T}_t()$
13: assume$(rnd = endRnd)$
14: $\forall g \in \mathsf{G}, r \in [0, R-1) \cdot$ assume$(g[r] = v_g[r+1])$
15: $\forall t \in \mathsf{T}, j \in \mathsf{J}(t) \cdot$ assert$(localAssert[t][j])$

16: **function** CS(Task t)
17: **if** (*) **then return** FALSE
18: $o := rnd$
19: $rnd := *$
20: assume$(o < rnd \leq endRnd)$
21: $\forall t' \in \mathsf{T}, j' \in \mathsf{J}(t') \cdot$ assume$(t < t' \Rightarrow (rnd \leq start[t'][j'] \vee rnd > end[t'][j']))$
22: **return** TRUE

23: **function** SCHEDULEJOBS( )

// Jobs are sequential

24: $\forall t \in \mathsf{T}, j \in \mathsf{J}(t) \cdot$ assume$(0 \leq start[t][j] \leq end[t][j] \leq R \wedge (\neg \text{last}(t, j) \Rightarrow end[t][j] \leq start[t][j+1]))$

// Jobs are well-nested

25: $\forall t_1 \in \mathsf{T}, t_2 \in \mathsf{T}, j_1 \in \mathsf{J}(t_1), j_2 \in \mathsf{J}(t_2) \cdot$ assume$\big((t_1 < t_2 \wedge start[t_1][j_1] \leq end[t_2][j_2] \wedge start[t_2][j_2] \leq end[t_1][j_1]) \Rightarrow (start[t_1][j_1] \leq start[t_2][j_2] \leq end[t_2][j_2] < end[t_1][j_1])\big)$

// Jobs respect preemption bounds

26: $\forall t_1 \in \mathsf{T}, t_2 \in \mathsf{T}, j_1 \in \mathsf{J}(t_1), j_2 \in \mathsf{J}(t_2) \cdot$ assume$\big((t_1 < t_2 \wedge j_2 \geq PB_{t_1}^{t_2} \wedge start[t_1][j_1] \leq start[t_2][j_2] \leq end[t_2][j_2] < end[t_1][j_1]) \Rightarrow end[t_2][j_2 - PB_{t_1}^{t_2}] < start[t_1][j_1])\big)$

27: **function** $\hat{T}_t$( )
Same as T_t*, but*
each statement 'st' is replaced with:
28: CS(t) ; $st[g \leftarrow g[rnd]]$,
and each 'assert(e)' is replaced with:
29: *localAssert*$[t][job] := e$

case this set is not empty. Note that this allows for acquiring multiple locks and releasing them in an arbitrary order. We check that the priority of the locks is assigned correctly (i.e., acquiring a lock must never lower the dynamic priority) by adding an assertion that checks this in line 8 of GETLOCK.

Correctness. We need to show that CS and SCHEDULEJOBS in Alg. 1, which are based on the base priorities, are still correct when priorities can change due to priority locks.

First, we need to show that the schedules permitted by SCHEDULEJOBS cover the legitimate schedules in the presence of priority locks. This is indeed the case because priority-locks cannot lead to *priority inversion* – a situation in which a job with a low base priority preempts a job with a high base priority. We will demonstrate this using jobs j_1 and j_2 with base priorities 1 and 2, respectively. j_1 cannot preempt j_2 even if it raises its own priority to 3, simply because j_2 is not running when j_1 acquires the lock. It can, therefore, only delay the start time of j_2, not preempt it. Thus, it is not necessary to explore schedules in which a low-base-priority job preempts a high-base-priority job.

Next, consider CS. Line 21 in CS guarantees that j_1 does not resume control in a round in which j_2 is still active. If j_1 raises its priority then j_2 cannot preempt it, and therefore this constraint blocks a computations in which j_1 resumes when j_2 was initially scheduled to run. Such a computation is illegal, however, because it corresponds to a preemption of a high-priority job j_1 by a lower-priority job j_2, it is accordingly blocked by the assume statement on line 3 of $\hat{T}_t$-WRAPPER.

Finally, since j_1 can raise its priority, it seems that we also need the opposite constraint (i.e., that j_2 does not resume control in a round in which j_1 is still active). However, there is no need for this constraint because SCHEDULEJOBS does not allow a schedule in which j_1 preempts j_2.

VI. PARTIAL-ORDER REDUCTION

Computations of a concurrent system have a natural partitioning: two computations are in the same class iff they reach the same observable states. Thus, for verification, it suffices to examine only one representative from each class. This is known as *Partial-Order Reduction* (POR) [17], [18].

POR is used widely and effectively in explicit-state Model Checking (e.g., [19]). Recently, it has been shown to be

Algorithm 2 Priority locks.

```
1: function T̂_t-WRAPPER( )
2:     sp := priority[rnd]
3:     assume(sp ≤ BASEPRIORITY(t)))
4:     priority[rnd] := BASEPRIORITY(t)
5:     T̂_t( )
6:     priority[rnd] := sp
7: function GETLOCK(Lock lck)
8:     assert(LOCKPRIORITY(lck) ≥ priority[rnd])
9:     lockSet := lockSet ∪ {lck}
10:    priority[rnd] := LOCKPRIORITY(lck)
11: function RELEASELOCK(Task t, Lock lck)
12:    lockSet := lockSet \ {lck}
13:    if lockSet = ∅ then
14:        priority[rnd] := BASEPRIORITY(t)
15:    else
16:        priority[rnd]:=priority of highest lock in lockSet
```

effective for symbolic methods as well [20]. In symbolic verification, POR translates into additional constraints to the underlying verification engine (in our case, CBMC and SAT). This does not always makes the solver faster. We report on our experience with this reduction in Sec. VII.

We propose two approaches for POR: syntactic and semantic. In syntactic POR, we first partition global variables into two groups – *task local* (accessed by a single task) and *shared* (accessed by multiple tasks). Intuitively, *task local* variables are local to a task, but preserved across jobs (e.g., *static* variables in C). Second, we allow preemptions only before statements that access a shared variable. Note that this also reduces the number of variables in the sequentialization since global variables that are not shared do not need to be guessed across rounds. We do not describe this reduction in more details since it is well-known and used by many other sequentialization approaches (e.g. [9]).

The idea behind semantic POR is to allow, for each shared variable g, a task to be preempted: (i) before a store $g := l$ only by a computation that loads or stores g, and (ii) before a load $l := g$ only by a computation that stores g. Intuitively, if a preempting computation does not affect the access to g then it is scheduled after the access, while preserving behavior.

The semantic POR is implemented by adding Boolean global read/write flags W_g and R_g for each shared g to indicate whether g was stored or loaded, respectively. These flags are treated as regular shared variables (i.e., guessed at the beginning of each round and checked at the end of the program). Each task body is changed as shown in Alg. 3. Only the case for a store $g := l$ is shown; the load $l := g$ is similar and is illustrated later with an example.

When a preemption happens before $g := l$, the read/write flags of g are reset (line 4) in the round in which the preemption happens. Then, at least one of the flags is assumed to become true in the round in which the task resumes. Thus, any computation in which the current task is preempted but g is not accessed, is blocked. Note that since in the sequential program we can access any round at any time, the resetting of the read/write flags (line 4) follows the preemption sequentially, but precedes it in the execution order. Finally, line 6 sets W_g to true to indicate that g was stored.

Algorithm 3 A fragment of $\hat{T}_t$ from Alg. 1 with POR. Only the case of a store to shared variable g is shown.

```
1: function T̂_t( )
   Same as T_t, but
     each statement 'g := l' is replaced with:
2:     oldRnd := rnd
3:     if CS(t) then
4:         W_g[oldRnd] := R_g[oldRnd] := FALSE
5:         assume(W_g[rnd] ∨ R_g[rnd])
6:     W_g[rnd] := TRUE
7:     g[rnd] := l
```

Example 3 *Under semantic POR, the two assignments*

1: $x := g\,;\, g := y$

in T_t, where x, y are local and g is shared, become the sequence in $\hat{T}_t$ that appears in Fig. 2.

```
1:  oldRnd := rnd
2:  if CS(t) then
3:      W_g[oldRnd] := R_g[oldRnd] := FALSE
4:      assume(W_g[rnd])
5:  R_g[rnd] := TRUE
6:  x := g[rnd]
7:  oldRnd := rnd
8:  if CS(t) then
9:      W_g[oldRnd] := R_g[oldRnd] := FALSE
10:     assume(W_g[rnd] ∨ R_g[rnd])
11: W_g[rnd] := TRUE
12: g[rnd] := y
```

Fig. 2. An encoding for Example 3.

Let $\mathcal{S}^{po}$ denote the sequentialization with POR. The following theorem shows that it is semantics preserving.

Theorem 3 *Let $\mathcal{C}$, $\mathcal{W}$ and $\mathcal{S}$ be as in Theorem 2 and $\mathcal{S}^{po}$ the corresponding POR. Then, for every execution π of $\mathcal{S}$ that violates a local assertion localAssert$[t][j]$ of task t and job j there is a corresponding execution π' of $\mathcal{S}^{po}$ that violates localAssert$[t][j]$, and vice versa.*

VII. Case Studies

We implemented our approach in a tool called Rek. Rek is built on top of CIL [21]. It takes as input C programs annotated with entry points of each task, their periods, worst case execution times, and the time bound $\mathcal{W}$. The output is a sequential C program $\mathcal{S}$ that is then verified by CBMC [22].

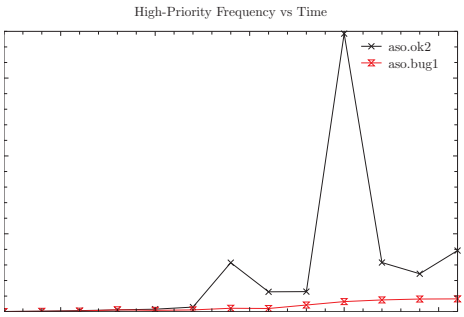

Fig. 3. Analysis time versus frequency of the highest-period time in 'aso'.

To evaluate our approach, we have used REK to verify several periodic programs. In the rest of this section, we report on this experience. The tool and the case studies are available at: http://www.andrew.cmu.edu/user/arieg/Rek.

Robot controller. The NXTway-GS controller, nxt for short, runs on nxtOSEK [2] – a real-time operating system ported to the LEGO MINDSTORM platform. nxtOSEK supports programs written in C with periodic tasks and priority ceiling locks. It is the target for Embedded Coder Robot NXT – a Model-Based Design environment for using Simulink models with LEGO robots.

The basic version of the controller has 3 periodic tasks: a *balancer*, with period of 4ms, that keeps the robot upright and monitors the bluetooth link for user commands, an *obstacle*, with period 50ms, that monitors a sonar sensor for obstacles, and a 100ms background task that prints debug information on an LCD screen.

We verified several versions of this controller. All of the properties verified involved the high-frequency balancer task. The balancer goes through 3 modes of execution: INIT, CALIBRATE, and CONTROL. In INIT mode all variables are initialized, and in CALIBRATE a gyroscope is calibrated. After that, balancer goes to CONTROL mode in which it iteratively reads the bluetooth link, reads the gyroscope, and sends commands to the two motors on the robot's wheels.

The results are shown in the top part of Table I. We have used $\mathcal{W} = 100$ms, which is the minimum time needed for all tasks to execute at least once. We did not enable semantic POR since it was irrelevant in this case (all shared variables were accessed by all tasks in all paths).

Experiments nxt.ok1 (nxt.bug1) check that the balancer is in a correct (respectively, incorrect) mode at the end of the time bound. Experiment nxt.ok2 checks that the balancer is always in one of its defined modes. Experiment nxt.bug3 checks that whenever *balancer* detects an obstacle, the *balancer* responds by moving the robot. We found that since the shared variables are not protected by a lock there is a race condition that causes the *balancer* to miss a change in the state of *obstacle* for one period. Experiment nxt.ok3 is the version of the controller where the race condition has been resolved using locks.

For the second part of the robot case study, we modified the original design to separate handling of each sensor by a separate task. Our design, called 'aso' has 3 tasks: *balancer*, *observer*, and *bluetooth*. The first two are the same as before, and the *bluetooth* is responsible for the bluetooth communication. We wanted a design in which the *balancer* task is lock-free, which was challenging. During our design, we unintentionally introduced subtle concurrency errors which were detected by REK.

TABLE I
Experimental results. OL and SL = # lines of code in the original C program and the generated sequentialization $\mathcal{S}$, respectively; GL = size of the GOTO program produced by CBMC; Var and Clause = # variables and clauses in the SAT instance, respectively; S = verification result – 'Y' for SAFE and 'N' for UNSAFE; Time = verification time in sec.

Name	Program Size		SAT Size			S	Time (s)
	OL	SL	GL	Var	Clause		
nxt.ok1	377	2,265	7,848	136K	426K	Y	22.16
nxt.bug1	378	2,265	7,848	136K	426K	N	9.95
nxt.ok2	368	2,322	8,572	141K	439K	Y	13.92
nxt.bug2	385	2,497	10,921	144K	451K	N	17.48
nxt.ok3	385	2,497	10,905	144K	449K	Y	18.32
aso.bug1	401	2680	13106	178K	572K	N	16.32
aso.bug2	400	2,682	13060	176K	566K	N	15.01
aso.ok1	398	2,684	13,026	175K	560K	Y	66.43
aso.bug3	426	3,263	19,211	373K	1,187K	N	59.66
aso.bug4	424	3,250	18,503	347K	1,099K	N	31.51
aso.ok2	421	3,251	18,589	348K	1,101K	Y	328.32
RW1	190	3,428	5,860	42K	125K	Y	20.74
RW1-PO	190	5,021	7,626	45K	134K	Y	14.71
RW2	239	4,814	8,121	52K	152K	Y	165.89
RW2-PO	239	7,356	10,388	56K	164K	Y	162.20
RW3	285	7,338	21,163	139K	419K	Y	436.86
RW3-PO	285	12,002	26,283	153K	467K	Y	199.13
RW4	244	7,255	19,745	117K	350K	Y	321.25
RW4-PO	244	12,272	24,261	130K	392K	Y	59.66
RW5	188	3,198	5,208	41K	119K	Y	47.83
RW5-PO	188	4,791	7,138	45K	131K	Y	20.35
RW6	257	5,231	7,634	54K	157K	Y	165.33
RW6-PO	257	8,235	10,119	59K	173K	Y	157.43

The results of these experiments are shown in the second part of Table I. We checked consistency of communication between the tasks. The experiments are: aso.bug1 and aso.bug2 – initial versions with inadequate locking leading to race conditions. aso.ok1 is a correct design with preemption locks. aso.bug3 is our first attempt at a lock-free implementation that was fundamentally flawed and had to be abandoned. aso.bug4 and aso.ok2 are a buggy and a correct version of the final design in which *obstacle* and *bluetooth* synchronize via priority locks and *balancer* is lock-free.

During the case study, we found it very convenient to increase the period of the highest priority task (thus decreasing its frequency). In many cases, this dramatically reduced verification time, while allowing us to draw meaningful conclusions from the counterexamples. Of course, this approach is not sound in general. Fig. 3 shows the relationship between the analysis time and the frequency of *balancer* for a correct and an incorrect version of the controller. In case the design is buggy, the time increased monotonically with the frequency. However, for a safe design, the time behaves erratically: e.g., increasing the frequency from 20 to 26 made it easier to verify. Such erratic behavior is common with SAT.

Reader-Writer. Reader-Writer (RW) is a common communication pattern in concurrent programs. We implemented three

lock-free flavors of a RW protocol (RW1, RW3, and RW5), and their counterparts with locks (RW2, RW4, and RW6). We checked consistency of communication between the tasks. Each protocol was analyzed with 3 to 6 tasks (depending on the protocol), with $\mathcal{W}$ such that every task executes once, and with an increasing number of shared variables. The results are shown in Table I. For each protocol, we only report on the hardest instance solved in under 10 mins. In these examples, semantic POR yields significant reduction in verification time. The results with POR are shown in Table I in rows named "PO". Note that in all cases, the number of variables and clauses with POR is larger, yet the verification time is smaller.

VIII. Related work

There is a large body of work in verification of logical properties of both sequential and concurrent programs (see [23] for a recent survey). However, these techniques abstract away time completely, by assuming a non-deterministic scheduler model. In contrast, we use a priority-sensitive scheduler model, and abstract time partially via our job-bounded abstraction.

A number of projects [5], [6] verify timed properties of systems using discrete-time [24] or real-time [7] semantics. They abstract away data- and control-flow, and verify models only. We focus on the verification of implementations of periodic programs, and do not abstract data- and control-flow.

Recently, Kidd et al. [4] showed a number of decidability results for reachability in finite-state periodic programs with recursion and locks. They apply sequentialization as well. The key idea is to share a single stack between all tasks and model preemptions by function calls. However, they not report on an implementation. In contrast, we focus on a practical solution to a bounded version of this problem.

In the context of concurrent software verification, several flavors of sequentialization have been proposed and evaluated (e.g., [9], [10], [11], [12]). Our procedure is closest to the LR [9] style. However, it differs from LR significantly, as discussed earlier (see Sec. IV-A), and provides a crucial advantage over LR for periodic programs, as discussed next.

In both cases, ours and LR, the number of variables is proportional to the number of rounds, R. However, LR allows more computations since it does not enforce priority constraints. In addition to yielding fewer false warnings, our approach guarantees better coverage for the same number of variables, as shown below:

Take two programs: (i) an N-task periodic program $\mathcal{C}$ with a time bound that permits exactly one job per task; (ii) the analogue of $\mathcal{C}$, called $\mathcal{C}'$, in a non-real-time setting, i.e., N threads scheduled non-deterministically, each executing one task of $\mathcal{C}$. Consider the value of R required to cover all reachable states, in our encoding $\mathcal{S}$ vs. the LR encoding of $\mathcal{C}'$. For LR, R is the number of possible context switches, which by itself is proportional to the number of statements over shared variables in $\mathcal{C}'$. This value of R is needed to explore a pathological path in $\mathcal{C}$ where, in each round, a single thread is executed and the threads are picked in reverse-round-robin order. In contrast, our approach only needs $R = N$, a much smaller value in most practical cases. In fact, the pathological path above is illegal, since scheduling tasks in reverse-round-robin order violates priority constraints.

IX. Conclusion

Periodic programs, i.e., periodic RTES with rate-monotonic scheduling, are an important sub-class of embedded real-time software. In this paper, we address the time-bounded verification of safety properties of periodic program implementations. We present a solution involving two steps – convert the target periodic program to a non-deterministic sequential program, and then verify it with an off-the-shelf verification tool. Our approach is sound, preserves both data- and control-flow, and abstracts the effect of time via preemption-bounds. Some of our techniques are applicable to other types of systems. Specifically, note that we only used the assumption that the verified system follows RMS in order to compute preemption bounds. Other periodic RTES can be modeled with our method by either supplying these bounds as part of the input, or by removing line 26 in Alg. 1 (this may hinder completeness, however). In addition, our partial order reduction is not restricted to periodic RTES, and is applicable when analyzing general multi-threaded programs.

We have implemented our approach in a tool, and used it to identify subtle concurrency errors in a robot controller. We believe that our work opens up several avenues for future work in real-time software verification, notably unbounded verification of periodic programs and the use of automated abstraction refinement techniques.

References

[1] C. L. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *J. of ACM*, vol. 20, no. 1, 1973.

[2] "nxtOSEK/JSP Open Source Platform for LEGO MINDSTORMS NXT," http://lejos-osek.sf.net.

[3] D. C. Locke, D. R. Vogel, L. Lucas, and J. B. Goodenough, "Generic Avionics Software Specification," SEI/CMU, Tech. Rep. CMU/SEI-90-TR-8-ESD-TR-90-209, 1990.

[4] N. Kidd, S. Jagannathan, and J. Vitek, "One Stack to Run Them All," in *Proc. of SPIN'11*, 2011.

[5] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a Nutshell," *STTT*, vol. 1, no. 1-2, pp. 134–152, 1997.

[6] V. A. Braberman and M. Felder, "Verification of Real-Time Designs: Combining Scheduling Theory with Automatic Formal Verification," in *Proc. of ESEC/SIGSOFT FSE'99*, 1999, pp. 494–510.

[7] R. Alur and D. Dill, "A Theory of Timed Automata," *Theor. Comp. Sci.*, vol. 126, pp. 183–235, 1994.

[8] S. Qadeer and D. Wu, "KISS: Keep It Simple and Sequential," in *Proc. of PLDI'04*, 2004, pp. 14–24.

[9] A. Lal and T. W. Reps, "Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis," in *Proc. of CAV'08*, 2008, pp. 37–51.

[10] S. L. Torre, P. Madhusudan, and G. Parlato, "Reducing Context-Bounded Concurrent Reachability to Sequential Reachability," in *Proc. of CAV'09*, 2009, pp. 477–492.

[11] N. Ghafari, A. J. Hu, and Z. Rakamaric, "Context-Bounded Translations for Concurrent Software: An Empirical Evaluation," in *Proc. of SPIN'10*, 2010, pp. 227–244.

[12] M. Emmi, S. Qadeer, and Z. Rakamaric, "Delay-Bounded Scheduling," in *Proc. of POPL'2011*, 2011, pp. 411–422.

[13] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zue, *Bounded Model Checking*, ser. Advances in Computers. Academic Press, 2003, vol. 58.

[14] N. Audsley, A. Burns, K. Tindell, and A. Wellings, "Applying New Scheduling Theory to Static Priority Preemptive Scheduling," *Software Eng. J.*, 1993.

[15] "VxWorks Programmer's Guide."
[16] R. Mall, *Real-Time Systems: Theory and Practice*. Prentice Hall, 2009.
[17] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, ser. Lecture Notes in Computer Science. Springer, 1996, vol. 1032.
[18] D. Peled, "All from One, One for All: on Model Checking Using Representatives," in *Proc. of CAV'93*, 1993, pp. 409–423.
[19] D. Bosnacki and G. J. Holzmann, "Improving Spin's Partial-Order Reduction for Breadth-First Search," in *Proc. of SPIN'05*, 2005.
[20] V. Kahlon, C. Wang, and A. Gupta, "Monotonic Partial Order Reduction: An Optimal Symbolic Partial Order Reduction Technique," in *Proc. of CAV'09*, 2009, pp. 398–413.
[21] G. C. Necula, S. McPeak, S. Rahul, and W. Weimer, "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs," in *Proc. of CC'02*, 2002.
[22] E. M. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs," in *Proc. of TACAS'04*, 2004, pp. 168–176.
[23] V. D'Silva, D. Kroening, and G. Weissenbacher, "A Survey of Automated Techniques for Formal Software Verification," *IEEE Trans. on CAD*, vol. 27, no. 7, pp. 1165–1178, 2008.
[24] F. Laroussinie, N. Markey, and P. Schnoebelen, "Efficient Timed Model Checking For Discrete-Time Systems," *Theor. Comput. Sci.*, vol. 353, no. 1-3, pp. 249–271, 2006.