Compositional Sequentialization of Periodic Programs

Sagar Chaki[1] Arie Gurfinkel[1] Soonho Kong[1] Ofer Strichman[2]

[1] CMU, Pittsburgh, USA [2] Technion, Haifa, Israel
{chaki,arie}@cmu.edu soonhok@cs.cmu.edu ofers@ie.technion.ac.il

Abstract. We advance the state-of-the-art in verifying periodic programs – a commonly used form of real-time software that consists of a set of asynchronous tasks running periodically and being scheduled preemptively based on their priorities. We focus on an approach based on sequentialization (generating an equivalent sequential program) of a time-bounded periodic program. We present a new compositional form of sequentialization that improves on earlier work in terms of both scalability and completeness (i.e., false warnings) by leveraging temporal separation between jobs in the same hyper-period and across multiple hyper-periods. We also show how the new sequentialization can be further improved in the case of harmonic systems to generate sequential programs of asymptotically smaller size. Experiments indicate that our new sequentialization improves verification time by orders of magnitude compared to competing schemes.

1 Introduction

Real-Time Embedded Software (RTES) controls a wide range of safety-critical systems – ranging from airplanes and cars to infusion pumps and microwaves – that impact our daily lives. Clearly, verification of such systems is an important problem domain. A modern RTES is inherently asynchronous (since it interacts with the real world), concurrent (to increase CPU utilization allowing for smarter, smaller, and more efficient systems), and must adhere to timing constraints. Developing such a RTES is therefore quite challenging.

A common way to address this challenge is to develop the RTES not as an arbitrary concurrent system, but as a *periodic program (PP)*. Indeed, PPs are supported by many real-time OSs, including OSEK [1], vxWorks [3], and RTEMS [2]. A PP $\mathcal{C}$ consists of a set of asynchronous tasks $\{\tau_i\}_i$, where each task $\tau_i = (I_i, T_i, P_i, C_i, A_i)$ is given by a priority I_i (higher number means higher priority), a loop-free body (i.e., code) T_i, a period P_i, a worst case execution time (WCET) C_i and an arrival time A_i. Each execution of T_i is called a *job*. The least common multiple of the periods of all the tasks is called a *hyper-period*.

The execution of $\mathcal{C}$ consists of a number of threads – one per task. A *legal* execution of $\mathcal{C}$ is one in which for every i, T_i is executed by its corresponding thread exactly once between time $A_i + (k-1) \cdot P_i$ and $A_i + k \cdot P_i$, for all natural $k > 0$. A common method for achieving this goal in the real-time systems

literature is to assume a preemptive fixed priority-based scheduler, and assign priorities according to the Rate Monotonic Scheduling (RMS) discipline. In RMS, shorter period implies higher priority. This is why a priority I_i is an element in the definition of τ_i. We assume that $\mathcal{C}$ is *schedulable* (i.e., it only produces legal executions) under RMS.

An example of a PP is the nxt/OSEK-based [1] LEGO Mindstorms controller (described further in Sec. 7) for a robot simulating a Turing machine. It has four periodic tasks: a *TapeMover*, with a 250ms period, that moves the tape; a *Reader*, with a 250ms period, that reads the current symbol; a *Writer*, with a 250ms period, that writes the current symbol; and a *Controller*, with a 500ms period, that issues commands to the other three tasks. Another example is a generic avionic mission system that was described in [18]. 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).

The topic of this paper is verification of logical properties (i.e., user supplied assertions, race conditions, deadlocks, API usage, etc.) of periodic programs. Surprisingly, this verification problem has not received significant research attention. While a PP is a concurrent program with priorities and structured timing constraints, most recent work on concurrent verification does not support priorities or priority-locks used by such systems, which motivates a solution tailored specifically to this domain.

In our previous work [7], we presented an approach for time-bounded verification of PPs, that given a PP $\mathcal{C}$ with assertions and a time-bound W, determines whether the assertions of $\mathcal{C}$ can be violated within the time bound W. The key idea there is to use W to derive an upper bound on the number of jobs that each task can execute within W, sequentialize the resulting job-bounded concurrent program, and use an off-the-shelf sequential verifier such as CBMC [6]. We call the sequentialization used in [7] *monolithic* (monoSeq).

Compositional Sequentialization. In this paper, we develop a new *compositional* sequentialization (compSeq) that improves substantially over monoSeq in terms of both scalability and completeness (i.e., less false warnings). The compositional nature of compSeq emerges from: (i) its use of information about tasks to deduce that certain jobs are *temporally separated*, i.e., one cannot preempt the other; and (ii) using this information to restrict legal thread interleavings. In particular, compSeq leverages two types of temporal separation between jobs: (i) among jobs in the same hyper-period (intra-HP); and (ii) between jobs from different hyper-periods (inter-HP). We illustrate these concepts – and thus the key difference between compSeq and monoSeq – with an example.

Intra-HP Temporal Separation. Consider a PP $\mathcal{C} = \{\tau_0, \tau_1, \tau_2\}$, where

$$\tau_0 = (0, T_0, 100, 50, 0) \qquad \tau_1 = (1, T_1, 50, 1, 0) \qquad \tau_2 = (2, T_2, 25, 1, 0) \qquad (1)$$

That is, task τ_0 has the lowest priority (i.e., 0), body T_0, period 100, WCET 50, and arrival time of 0, and similar for the other tasks. During the time-bound

$W = 100$, there is one execution of T_0, 2 of T_1 and 4 of T_2. Hence, monoSeq constructs sequentialization of the following concurrent program:

$$T_0 \parallel (T_1 \,;T_1) \parallel (T_2 \,;T_2 \,;T_2 \,;T_2) \tag{2}$$

and adds additional constraints to remove interleavings that are infeasible due to timing and priority considerations. monoSeq ignores the arrival time information and assumes that the lowest priority task τ_0 can be preempted by any execution of any higher priority task. This leads to large number of interleavings negatively affecting both scalability and completeness. In contrast, the new approach compSeq, notes that τ_2 arrives at the same time as τ_1 and τ_0 and is therefore given the CPU first. Hence the first instance of τ_2 does not interleave with any other task. Similarly, it notes that because of the WCET, the last execution of τ_2 does not interleave with any other task either. Thus, it uses the following concurrent program

$$T_2 \,;T_1 \,;(T_0 \parallel (T_2 \,;T_2 \,;T_1)) \,;T_2 \tag{3}$$

That is, the single repetition of the lowest priority task τ_0 is interleaved only with some of the executions of higher-priority tasks. In this example, it is easy to see that both (2) and (3) over-approximate $\mathcal{C}$, but (3) is more sequential, hence it leads to a sequentialization that has fewer spurious interleavings and is easier to analyze.

Inter-HP Temporal Separation. Next, suppose the time-bound W is increased to 200 (i.e., to two hyper-periods). In this case, the concurrent program constructed by monoSeq becomes:

$$(T_0 \,;T_0) \parallel (T_1 \,;T_1 \,;T_1 \,;T_1) \parallel (T_2 \,;T_2 \,;T_2 \,;T_2 \,;T_2 \,;T_2 \,;T_2 \,;T_2) \tag{4}$$

However, note that 100 is the hyper-period of $\mathcal{C}$. It is easy to see that if all tasks initially arrive at time 0, then any execution that starts within a hyper-period ends before the end of the hyper-period. Thus, instead of monolithically encoding all execution in a given time bound, it is sufficient to encode repetitions of a hyper-period. In particular, in this example, compSeq sequentializes the following program:

$$\underbrace{T_2 \,;T_1 \,;(T_0 \parallel (T_2 \,;T_2 \,;T_1)) \,;T_2)}_{\text{Sequentialization of HP\#1}} \;;\; \underbrace{(T_2 \,;T_1 \,;(T_0 \parallel (T_2 \,;T_2 \,;T_1)) \,;T_2)}_{\text{Sequentialization of HP\#2}} \tag{5}$$

Note that verifying the sequentialization of one of the two hyperperiods (HP#1 or HP#2) in isolation is not sound since they communicate via global variables.

Our experimental results show that the difference in the encoding has a dramatic effect on performance. We show that monoSeq does not scale at all on a small (but realistic) robotics controller, but compSeq is able to solve many verification problems in the order of minutes.

Contributions. The paper makes the following contributions. First, we present compSeq when the time bound is a single hyper-period of $\mathcal{C}$, focusing on its use of intra-HP separation. Interestingly, we show that assuming that all tasks start together – a common assumption in schedulability analysis [17] – is unsound for verification (see Theorem 2). That is, a program is schedulable iff it is schedulable assuming that all tasks start at time 0. But, the program might be safe when all tasks start together, but not safe if they start at different times.

Second, we improve compSeq for the case of *harmonic* PPs – a class of PPs in which for every pair of tasks τ_1 and τ_2, $\text{lcm}(P_1, P_2) = \max(P_1, P_2)$, i.e., P_1 is a multiple of P_2. In practice, PPs are often designed to be harmonic, for example, to achieve 100% CPU utilization [13] and more predictable battery usage [20]. The improved version, called harmonicSeq, uses intra-HP separation just like compSeq, but generates sequential programs of asymptotically smaller size (both theoretically and empirically).

Third, we extend compSeq (and harmonicSeq) to multiple hyper-periods of $\mathcal{C}$ by applying it individually to each hyper-period and composing the results sequentially, thereby leveraging inter-HP separation. We show that while this is *unsound* in general (see the discussion after Theorem 3), i.e., a periodic program is not always logically equivalent to repeating the sequentialization of its hyper-period ad infinitum, it is sound under the specific restrictions on arrival times already imposed by [7].

We have implemented our approach and validated it by verifying several RTES for controlling two flavors of LEGO Mindstorms robots – one that self-balances, avoids obstacles and responds to remote controls, and another that simulates a Turing machine. We observe that verification with compSeq is much faster (in one case by a factor of 480x) than that with monoSeq. The improvement is more pronounced with increasing number of hyper-periods. In many cases, verification with compSeq completes while monoSeq runs out of resources. Further details are presented in Sec. 7.

The rest of this paper is organized as follows. Sec. 2 discusses preliminary concepts. Sec. 3 and Sec. 4 present compSeq and harmonicSeq, respectively, but for one hyper-period. Sec. 5 extends them to multiple hyper-periods. In Sec. 6, we survey related work. Sec. 7 presents experimental results, and Sec. 8 concludes.

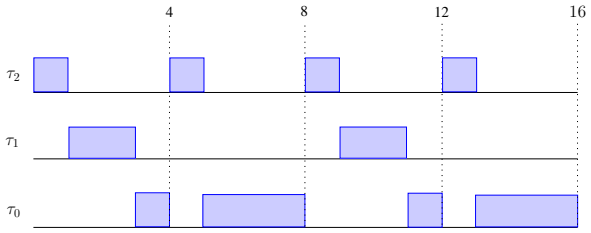
2 Preliminaries

A *task* τ is a tuple $\langle I, T, P, C, A\rangle$, where I is the priority, T – a bounded procedure (i.e., no unbounded loops or recursion) called the task body, P – the period, C – the worst case execution time (WCET) of T, and A, called the release time, is the time at which the task is first enabled[1]. A *periodic program* (PP) is a set of tasks. In this paper, we consider a N-task PP $\mathcal{C} = \{\tau_0, \ldots, \tau_{N-1}\}$, where $\tau_i = \langle I_i, T_i, P_i, C_i, A_i\rangle$. We assume that: (i) for simplicity, $I_i = i$; (ii) execution

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

Task	C_i	P_i
τ_2	1	4
τ_1	2	8
τ_0	8	16

(a)



(b)

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

times are positive, i.e., $C_i > 0$; and (iii) priorities are *rate-monotonic* and distinct – tasks with smaller period have higher priority.

Semantics. A PP 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, preemptive, and priority-sensitive – the CPU is always given to the enabled task with the highest priority, preempting the currently executing task if necessary. Formally, the semantics of $\mathcal{C}$ is the asynchronous concurrent program:

$$\|_{i=0}^{N-1}\ k_i := 0\ ;\ \textbf{while}(\textsc{Wait}(\tau_i, k_i))\ (T_i\ ;\ k_i := k_i + 1) \tag{6}$$

where $\|$ is priority-sensitive interleaving, $k_i \in \mathbb{N}$ is a counter and $\textsc{Wait}(\tau_i, k_i)$ returns FALSE if the current time is greater than $A_i^{k_i}$, and otherwise blocks until time $A_i^{k_i}$ and then returns TRUE.

Schedulability. An execution of each task body T_i in (6) is called a *job*. A job's *arrival* is the time when it becomes enabled (i.e., $\textsc{Wait}(\tau_i, k)$ in (6) 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 task τ_i, denoted by RT_i, is the maximum response times of all of its jobs over all possible executions. Since tasks have positive execution times, their response times are also positive, i.e., $RT_i > 0$.

Note that WAIT in (6) returns TRUE if a job has finished before its next period. A periodic program is *schedulable* iff there is no run of (6) (legal with respect to priorities) in which WAIT returns FALSE. That is, a program is schedulable iff in every run every task starts and finishes within its period.

There are well-known techniques [17] to decide schedulability of periodic programs. In this paper, we are interested in logical properties of periodic programs, assuming that they meet their timing constraints. Thus, we assume that $\mathcal{C}$ is a schedulable periodic program.

Example 1. Consider the task set in Fig. 1(a). Suppose that $RT_2 = 1$, $RT_1 = 3$ and $RT_0 = 16$. A schedule demonstrating these values is shown in Fig. 1(b).

Time-Bounded Verification. Initially, in Sec. 3 and 4, we assume that $\mathcal{C}$ executes for one "hyper-period" $\mathcal{H}$. The hyper-period [17] of $\mathcal{C}$ is the least common

multiple of $\{P_0, \ldots, P_{n-1}\}$. Thus, we verify the time-bounded periodic program $\mathcal{C}_\mathcal{H}$ that executes like $\mathcal{C}$ for time $\mathcal{H}$ and then terminates. Subsequently, in Sec. 5, we show how to extend verification of $\mathcal{C}_\mathcal{H}$ to multiple hyper-periods.

Throughout the paper, we assume that the first job of each task finishes before its period, i.e.,

$$\forall 0 \leq i < N \centerdot A_i + RT_i \leq P_i \ . \tag{7}$$

Under this restriction, the number of jobs of task τ_i that executes in $\mathcal{C}_\mathcal{H}$ is:

$$J_i = \frac{\mathcal{H}}{P_i} \ . \tag{8}$$

The semantics of $\mathcal{C}_\mathcal{H}$ is the asynchronous concurrent program:

$$\|_{i=0}^{N-1}\ k_i := 0 \,;\, \textbf{while}(k_i < J_i \wedge \textsc{Wait}(\tau_i, k_i))\ (T_i \,;\, k_i := k_i + 1) \ . \tag{9}$$

This is analogous to the semantics of $\mathcal{C}$ in (6) except that each task τ_i executes J_i jobs. We write $\mathsf{J}(\tau, k)$ to denote the k-th job (i.e., the job at the k-th position) of task τ. Thus, the set of all jobs of $\mathcal{C}_\mathcal{H}$ is:

$$\mathsf{J} = \bigcup_{0 \leq i < N} \{\mathsf{J}(\tau_i, k) \mid 0 \leq k < J_i\} \ . \tag{10}$$

3 Job-Bounded Verification

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 $\mathcal{H}$. The first step, sequentialization, outputs a non-deterministic sequential program $\mathcal{S}$ with assume statements, as shown in Algorithm 1. The second step is the verification of $\mathcal{S}$ with an off-the-shelf program verifier. In the rest of this section, we present our first sequentialization algorithm compSeq.

Sequentialization: Intuition. compSeq uses the idea that any execution π of $\mathcal{C}_\mathcal{H}$ 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(b) 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 $R = |\mathsf{J}|$ jobs start and end in π, and thus π has R rounds. Therefore, compSeq reduces the bounded concurrent execution of $\mathcal{C}_\mathcal{H}$ into a sequential execution with R rounds. Initially, jobs are allocated (or scheduled) to rounds. Then, each job is executed independently, in lexicographic order of increasing priority and job position. That is, lower priority jobs are executed first, and jobs of the same task are ordered by their position in the task.

In addition, compSeq leverages arrival time of each job in two important ways. First, by observing that if in every execution a job j completes before another job j' arrives, then j can precede j' in the sequentialization, independently of the priorities of the jobs. Second, by only exploring job schedules that do not violate arrival constraints. This has several benefits. First, compSeq is more complete – it generates fewer false warnings. Second, it enables eager checking for user-specified assertions incrementally.

We now present compSeq in detail. We first describe the job ordering used by compSeq, and then the sequential program $\mathcal{S}$ that compSeq generates.

Job Ordering. Consider a job $j = \mathsf{J}(\tau, k)$. Let A be the arrival time of the first job of τ and P be the period of τ. Then, the arrival time of j is $A(j) = A + k \times P$. Similarly, let RT be the response time of τ. Then the departure time of j is $D(j) = A(j) + RT$. Since we assume that $RT > 0$, we know that $A(j) < D(j)$. Let $\pi(j)$ denote the priority of its task τ. We first present three ordering relations $\lhd$, $\uparrow$ and $\sqsubset$ on jobs. Informally, $j_1 \lhd j_2$ means that j_1 always completes before j_2 begins, $j_1 \uparrow j_2$ means that it is possible for j_1 to be preempted by j_2, and $\sqsubset$ is the union of $\lhd$ and $\uparrow$.

Definition 1. *The ordering relations $\lhd$, $\uparrow$ and $\sqsubset$ are defined as follows:*

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

Lemma 1 relates $\sqsubset$ with $\lhd$ and $\uparrow$.

Lemma 1. *For any two jobs j_1 and j_2, we have:*

$$j_1 \sqsubset j_2 \iff j_1 \lhd j_2 \vee j_1 \uparrow j_2 \tag{11}$$

Note that $j_1 \sqsubset j_2$ means that either j_1 always completes before j_2, or it is possible for j_1 to be preempted by j_2. Also, $\sqsubset$ is a total strict ordering since it is a lexicographic ordering by (arrival time, priority). Moreover, $\sqsubset$ is computable in $\mathcal{O}(R \cdot \log(R))$ time, where R is the total number of jobs.

Construction of $\mathcal{S}$. The structure of $\mathcal{S}$ is given by the pseudo-code in Alg. 1. The top-level function is main. It sets (line 4) the global variables at the beginning of the first round to their initial values, and then calls hyperPeriod to execute the sequential program corresponding to a time-bound of $\mathcal{H}$.

hyperPeriod first calls scheduleJobs to create a legal job schedule – i.e., assign a starting round $start[j]$ and an ending round $end[j]$ to each job j. It then executes, in the order induced by $\sqsubset$, each job j by invoking runJob(j).

In scheduleJobs, line 12 ensures that $start[j]$ and $end[j]$ are sequential and within legal bounds; lines 13–14 ensure that jobs are properly separated; line 15 ensures that jobs are well-nested – if j_2 preempts j_1, then it finishes before j_1.

Algorithm 1 The sequentialization $\mathcal{S}$ of the time-bounded periodic program $\mathcal{C}_{\mathcal{H}}$. Notation: J is the set of all jobs; G is the set of global variables of $\mathcal{C}$; i_g is the initial value of g according to $\mathcal{C}$; '$*$' is a non-deterministic value.

```
 1: var rnd, start[], end[], localAssert[]
 2: ∀g ∈ G . var g[], v_g[]

 3: function MAIN( )
 4:     ∀g ∈ G . g[0] := i_g
 5:     HYPERPERIOD()

 6: function HYPERPERIOD( )
 7:     SCHEDULEJOBS()
 8:     ∀g ∈ G . ∀r ∈ [1, R).
            v_g[r] := *; g[r] := v_g[r]
        let the ordering of jobs by ⊏ be
        j_0 ⊏ j_1 ⊏ ... j_{R-1}
 9:     RUNJOB(j_0); ...; RUNJOB(j_{R-1})

10: function SCHEDULEJOBS( )
11:     ∀j ∈ J . start[j] = *; end[j] = *
    // Jobs are sequential
12:     ∀i ∈ [0, N) . ∀k ∈ [0, J_i) . assume
          (0 ≤ start[J(i, k)] ≤ end[J(i, k)] < R)
    // Jobs are well-separated
13:     ∀j_1 ◁ j_2 . assume(end[j_1] < start[j_2])
14:     ∀j_1 ↑ j_2 . assume(start[j_1] ≤ start[j_2])
    // Jobs are well-nested
15:     ∀j_1 ↑ j_2 . assume(start[j_2] ≤ end[j_1]
            ⟹ (start[j_2] ≤ end[j_2] < end[j_1]))

16: function RUNJOB(Job j)
17:     localAssert[j] := 1
18:     rnd := start[j]
19:     T̂(j)
20:     assume(rnd = end[j])
21:     if rnd < R − 1 then
22:         ∀g ∈ G . assume
              (g[rnd] = v_g[rnd + 1])
23:     X := {j' | (j' = j ∨ j' ↑ j) ∧
          (∀j'' ≠ j . j' ↑ j'' ⇒ j'' ⊏ j)}
24:     ∀j' ∈ X . assert(localAssert[j'])

25: function T̂(Job j)
    Obtained from T_t by replacing
      each statement 'st' with:
26:     CS(j) ; st[g ← g[rnd]]
      and each 'assert(e)' with:
27:     localAssert[j] := e

28: function CS(Job j)
29:     if (*) then return FALSE
30:     o := rnd ; rnd := *
31:     assume(o < rnd ≤ end[j])
        ∀j' ∈ J . j ↑ j' ⟹
32:         assume(rnd ≤ start[j'] ∨
              rnd > end[j'])
33:     return TRUE
```

We assume, without loss of generality, that a job j contains at most one `assert`, and use the variable $localAssert[j]$ to represent the argument to this assertion. RUNJOB(j) first initializes $localAssert[j]$ to 1. It then ensures that j starts in round $start[j]$ (line 18). Next, it executes the body of j (via $\hat{T}(j)$), ensures that j terminates in round $end[j]$, and ensures consistency of round $end[j]$ by checking that the final value of g in it equals its guessed initial value in round $end[j] + 1$. Finally, it checks whether j caused an assertion violation.

Function $\hat{T}(j)$ is identical to the body of j's task, except that it uses variable $g[rnd]$ instead of g and records the argument to its assertion instead of checking it. This is important because assertions must be checked only after ensuring consistency of all relevant rounds. Function $\hat{T}(j)$ also increases value of rnd non-deterministically (by invoking function CS) to model preemption by higher priority jobs. As in other work [14], preemption is only allowed before access of global variables, without losing soundness.

Soundness of compSeq. The state-of-the-art sequentialization for periodic programs – which we refer to as monoSeq– was developed and shown to be sound in our prior work [7]. monoSeq first executes all jobs in increasing order of priority and job position, then ensures round consistency, and finally checks for assertion violations. In contrast, in the case of compSeq: (1) jobs are serialized in the order $\sqsubset$; (2) the consistency of round $end[j]$ is checked as soon as job j completes, we call this *eager-check-assumptions*; (3) assertions are also checked as soon as all jobs that affect the outcome of an assertion has completed, we call this *eager-check-assertions*. Theorem 1 states that despite these differences, the soundness of monoSeq to carry over to compSeq.

Theorem 1. *Ordering jobs by* $\sqsubset$*,* eager-check-assumptions*, and* eager-check-assertions *are sound.*

The set of arrival times is called the *phasing*. A special case is *zero-phasing*, when all tasks start together – i.e., $\forall i \in [0, N)$. $A_i = 0$. It depends on the OS whether zero-phasing can be assumed or not. For example, OSEK enforces zero-phasing while RTLinux does not.

Zero-phasing is a sufficient assumption for completeness of schedulability analysis, i.e., a system is schedulable for all phasings iff it is schedulable for the zero-phasing [17]. However, assuming zero-phasing is unsound for verification, as illustrated by Theorem 2. At the same time, assuming arbitrary phasing (as in monoSeq) leads to many false positives (see Sec. 7). compSeq handles tasks with a given phasing. This makes it sound, yet more complete than monoSeq.

Theorem 2. *The safety of a periodic program under zero-phasing does not imply its safety under all phasings.*

Proof. Consider a periodic program $\mathcal{C}$ with two tasks: t_1 and t_2. The tasks communicate via a shared variable x, initially 0. Task t_1: period 2ms, priority 1, T_1 is `assert`$(x\%2 = 1)$; task t_2: period 1ms, priority 2, T_2 is $x = x + 1$. The WCET of both tasks is 0.1ms. Under zero phasing, there is no preemption, and t_1 always reads an odd value of x. Hence the assertion succeeds. However, if t_1 arrives before t_2, then the value of x read by t_1 is 0, which fails the assertion. Therefore, $\mathcal{C}$ is safe under zero-phasing, but not under all phasings. Since $\mathcal{C}$ is harmonic, the theorem holds for harmonic periodic programs as well. □

4 Verifying Harmonic Periodic Programs

In the worst case, the number of constraints in scheduleJobs() is quadratic in the total number of jobs. This is essentially because relations $\lhd$ and $\uparrow$ collectively can have $\mathcal{O}(R^2)$ job pairs. In this section, we show that for a special class of periodic programs, known as harmonic programs, we are able to implement scheduleJobs using $\mathcal{O}(R \cdot N)$ constraints only, where N is the number of tasks. This leads to our sequentialization algorithm harmonicSeq. Since N is typically exponentially smaller than R, harmonicSeq yields an asymptotically smaller job scheduling function.

Algorithm 2 Procedure to assign legal starting and ending rounds to jobs in a harmonic program.

1: **var** $min[\,], max[\,]$ *//extra variables*

2: **function** SCHEDULEHARMONIC( )

3: $\forall j \in \mathsf{J}\,\centerdot\; start[j] = *; end[j] = *; min[j] = *; max[j] = *$

// Correctness of min and max

4: $\forall n \in \mathcal{T}\,\centerdot\; isleaf(n) \implies \text{assume}(min[n] = start[n] \wedge max[n] = end[n])$

5: $\forall n \in \mathcal{T}\,\centerdot\; \neg isleaf(n) \implies \text{assume}(min[n] = \text{MIN}(start[n], min[first(n)]))$

6: $\forall n \in \mathcal{T}\,\centerdot\; \neg isleaf(n) \implies \text{assume}(max[n] = \text{MAX}(end[n], max[last(n)]))$

// Jobs are sequential

7: $\forall n \in \mathcal{T}\,\centerdot\; \text{assume}(low(n) \leq start[n] \leq end[n] \leq high(n))$

// Jobs are well-separated

8: $\forall n \in \mathcal{T}\,\centerdot\; hasNext(n) \implies \text{assume}(max[n] < min[next(n)])$

9: $\forall j_1 \uparrow j_2\,\centerdot\; \text{assume}(start[j_1] \leq start[j_2])$

// Jobs are well-nested

10: $\forall j_1 \uparrow j_2\,\centerdot\; \text{assume}(start[j_2] \leq end[j_1] \implies (start[j_2] \leq end[j_2] < end[j_1]))$

$\mathcal{T}(n)$	= sub-tree of $\mathcal{T}$ rooted at n	$isleaf(n)$	= true iff n is a leaf node
$level(n)$	= level of node n	$size(n)$	= number of nodes in $\mathcal{T}(n)$
$id(n)$	= position of n in the DFS pre-ordering of $\mathcal{T}$	$hasNext(n)$	= true iff n is *not* the last node at level $level(n)$
$next(n)$	= node after n at level $level(n)$	$first(n)$	= first child of n
$last(n)$	= last child of n	$maxid(n)$	= $id(n) + size(n) - 1$
$low(n)$	= $id(n) - level(n)$	$high(n)$	= $maxid(n)$

Fig. 2. Functions on each node n of the job-graph.

A periodic program $\mathcal{C} = \{\tau_0, \ldots, \tau_{N-1}\}$ is harmonic if $\forall 0 < i < N\,\centerdot\; P_{i-1} | P_i$, where $x|y$ means that x is divisible by y. For $0 \leq i < N-1$, let $r(\tau_i) = P_i/P_{i+1}$. Note that $\mathcal{H} = P_0$ and thus $J_0 = 1$. Also, the taskset from Example 1 defines a harmonic program. Harmonicity is a common restriction imposed by real-time system designers, especially in the safety-critical domain. For example, it is possible to achieve 100% CPU utilization [13] for a harmonic program with rate monotonic scheduling.

We begin by defining the *job-tree* $\mathcal{T}$. The nodes of $\mathcal{T}$ are the jobs of $\mathcal{C}_\mathcal{H}$, and there is an edge from $j_1 = \mathsf{J}(\tau_1, p_1)$ to $j_2 = \mathsf{J}(\tau_2, p_2)$ iff $\pi(j_2) = \pi(j_1) + 1 \wedge p_2/r(\tau_1) = p_1$. Thus, the job-tree is a balanced tree of depth N rooted at $\mathsf{J}(\tau_0, 0)$ and for $0 \leq i < N-1$, each node at level i (the root is at level 0) has r_i children.

Note that since $\mathcal{C}$ is harmonic, $\uparrow$ contains $\mathcal{O}(R \cdot N)$ job pairs. This is because if $j_1 \uparrow j_2$, then j_1 must be an ancestor of j_2 in $\mathcal{T}$, and there are $\mathcal{O}(R \cdot N)$ such pairs. Moreover, all elements of $\uparrow$ can be enumerated in $\mathcal{O}(R \cdot N)$ by checking for each node j_2 of $\mathcal{T}$, and each ancestor j_1 of j_2, whether $j_1 \uparrow j_2$.

Let nodes at the same level of $\mathcal{T}$ be ordered by increasing arrival time. For each node $n \in \mathcal{T}$, we define $size(n)$, $first(n)$, $last(n)$, $id(n)$, $maxid(n)$, $level(n)$, $low(n)$ and $high(n)$ as in Fig. 2. Note that these are statically computable from

$\mathcal{T}$. Also, $maxid(n) = \text{MAX}_{k \in \mathcal{T}(n)} id(n)$, $low(n)$ is the earliest round in which job n can start, and $high(n)$ is the latest round in which job n can finish.

Since each job is a node of $\mathcal{T}$, an assignment to $start[\,]$ and $end[\,]$ is equivalent to two functions $start$ and end from nodes of $\mathcal{T}$ to values in the range $[0, R)$. This, in turn, induces the following two additional functions from $\mathcal{T}$ to $[0, R)$:

$$min(n) = \text{MIN}_{k \in \mathcal{T}(n)} start(k) \qquad max(n) = \text{MAX}_{k \in \mathcal{T}(n)} end(k)$$

The difference between HARMONICSEQ and COMPSEQ is that HARMONICSEQ uses function SCHEDULEHARMONIC – shown in Algorithm 2 – instead of SCHEDULEJOBS. The key features of SCHEDULEHARMONIC are:

- It uses two additional arrays (defined at line 1) to represent functions min and max. It adds constraints (lines 4–6) to ensure that these arrays contain appropriate values. Note that these constraints are based on the following recursive definition of min and max:

$$min(n) = \begin{cases} start(n) & \text{if } isleaf(n) \\ \text{MIN}(start(n), min(first(n))) & \text{otherwise} \end{cases}$$

$$max(n) = \begin{cases} end(n) & \text{if } isleaf(n) \\ \text{MAX}(end(n), max(last(n))) & \text{otherwise} \end{cases}$$

- It imposes stricter constraints (line 7) over $start[\,]$ and $end[\,]$, compared to SCHEDULEJOBS. Specially, it ensures that $low(n) \leq start[n] \leq end[n] \leq high(h)$ instead of $0 \leq start[n] \leq end[n] < R$.
- It uses min, max and $\uparrow$ (lines 8–9) to ensure separation. Function SCHEDULEJOBS uses $\lhd$ and $\uparrow$ instead for this purpose.
- The relation $\uparrow$ is used (line 10), as in SCHEDULEJOBS, to ensure that jobs are well-nested.

Note that the number of constraints in SCHEDULEHARMONIC is $\mathcal{O}(R \cdot N)$. Specifically, to ensure that jobs are sequential, we require $\mathcal{O}(R)$ constraints. Also, since $\uparrow$ contains $\mathcal{O}(R \cdot N)$ job pairs, specifying that jobs are well-separated and well-nested requires $\mathcal{O}(R \cdot N)$ constraints each.

5 Verification Over Multiple Hyper-Periods

In this section, we present an approach to extend job-bounded verification to the case where the time-bound is a multiple of its hyper-period. Let $\mathcal{C}$ be a schedulable periodic program with hyper-period $\mathcal{H}$ and let the time-bound for verification be $(m \times \mathcal{H})$. From (9), it follows that the semantics of $\mathcal{C}_{(m \times \mathcal{H})}$ is given by the following asynchronous concurrent program:

$$\|_{i=0}^{N-1}\ k_i := 0\,;\, \textbf{while}(k_i < m \times J_i \wedge \text{WAIT}(\tau_i, k_i))\ (T_i\,;\, k_i := k_i + 1)\,. \qquad (12)$$

Let $\mathcal{C}_{\mathcal{H}}^m$ be the program that invokes function MULTIHYPER in Fig. 3(a) with argument m. In other words, $\mathcal{C}_{\mathcal{H}}^m$ executes $\mathcal{C}_{\mathcal{H}}$ sequentially m times. Since the

1: **var** $rnd, start[\,], end[\,]$
2: **var** $localAssert[\,]$
3: $\forall g \in$ `G.` **var** $g[\,], v_g[\,]$

4: **function** MULTIHYPER(k)
5: $\quad \forall g \in$ `G.` $g[0] := i_g$
6: $\quad$ **for** $i = 1$ to k **do**
7: $\quad\quad$ HYPERPERIOD()
8: $\quad\quad \forall g \in$ `G.` $g[0] := g[R-1]$

(a)

Task	A_i	C_i	P_i
τ_1	1.9	0.5	2
τ_2	0	0.1	1

```
int x=0; int y=0;
void T1() {
  int t=y; assert(x == t+2); y=x;
}
void T2() {x++;}
```

(b)

Fig. 3. (a) Sequentialization for multiple hyper-periods; function HYPERPERIOD is shown in Alg. 1; (b) A periodic program $\mathcal{C}$ such that $\mathcal{C}_{2\mathcal{H}}$ is not equivalent to $\mathcal{C}^2_{\mathcal{H}}$.

arrival-pattern of jobs repeats every hyper-period, it is tempting to conclude that the semantics of $\mathcal{C}_{(m\times\mathcal{H})}$ is equivalent to $\mathcal{C}^m_{\mathcal{H}}$. We show that this is true under our assumption (7) from Sec. 2 on job arrivals, but is not true in general.

Theorem 3. *Let $\mathcal{C}$ be a schedulable periodic program with hyper-period $\mathcal{H}$ satisfying assumption* (7) *from Sec. 2. Let $\mathcal{C}_{(m\times\mathcal{H})}$ and $\mathcal{C}^m_{\mathcal{H}}$ be programs as defined above. Then, $\mathcal{C}_{(m\times\mathcal{H})}$ and $\mathcal{C}^m_{\mathcal{H}}$ are semantically equivalent with respect to safety properties for any natural number m.*

Proof. We show, by induction on m, that every execution of $\mathcal{C}_{(m\times\mathcal{H})}$ is matched by an execution of $\mathcal{C}^m_{\mathcal{H}}$, and vice versa. The base case ($m = 1$) is trivial since, by definition, $\mathcal{C}_{\mathcal{H}} = \mathcal{C}^1_{\mathcal{H}}$. For the inductive case, let $m = k + 1$ and assume that the theorem holds for $m = k$. By assumption (7), every execution of $\mathcal{C}_{k\mathcal{H}}$ terminates within time $k\mathcal{H}$. Thus, every execution t of $\mathcal{C}_{(m\times\mathcal{H})}$ is of the form $t_1 \bullet t_2$ – where $\bullet$ denotes concatenation – such that t_1 is an execution of $\mathcal{C}_{k\mathcal{H}}$ and t_2 is an execution of $\mathcal{C}_{\mathcal{H}}$. By induction, t_1 is also an execution of $\mathcal{C}^k_{\mathcal{H}}$. Therefore, t is an execution of $\mathcal{C}^{k+1}_{\mathcal{H}} = \mathcal{C}^m_{\mathcal{H}}$. The converse is proven analogously. □

To see that assumption (7) is necessary for Theorem 3, consider the periodic program $\mathcal{C}$ shown in Fig. 3(b). The tasks communicate via two shared variables `x` and `y`. Let $m = 2$. Note that $\mathcal{C}_{2\mathcal{H}}$ violates the assertion, whereas $\mathcal{C}^2_{\mathcal{H}}$ does not. In $\mathcal{C}^2_{\mathcal{H}}$, `x == y` is an inductive loop invariant: it holds initially, and is maintained since the single job of τ_1 starts after both jobs of τ_2. Therefore, the assertion is never violated since `x == y` is always true at the beginning of each hyper-period, and `x` is incremented twice by the jobs of τ_2 before the job of τ_1 begins.

However, consider the following execution in $\mathcal{C}_{2\mathcal{H}}$: (i) the τ_2 jobs from the first hyper-period set `x` to 2; (ii) the τ_1 job from the first hyper-period set `t` to 0; (iii) the second hyper-period begins; (iv) the first job of τ_2 in the second hyper-period arrives, preempts the τ_1 job and sets `x` to 3; (v) the τ_1 jobs resumes and reads the value 3 for `x`; since the value of `t` is 0, the assertion is violated. Therefore, $\mathcal{C}_{2\mathcal{H}}$ and $\mathcal{C}^2_{\mathcal{H}}$ are not equivalent with respect to safety properties. Note that $\mathcal{C}$ is harmonic, so this is true for harmonic periodic programs as well.

In summary, Theorem 3 captures the essence of inter-HP temporal separation between jobs. It allows us to verify $\mathcal{C}_{(m\times\mathcal{H})}$ by verifying $\mathcal{C}^m_{\mathcal{H}}$ instead. Since $\mathcal{C}^m_{\mathcal{H}}$ is a

sequential non-deterministic program, it can be verified by any existing software model checker. Experimental results (see Sec. 7) indicate that this new way of verifying a periodic program over multiple hyper-periods is orders of magnitude faster than the state-of-the-art.

6 Related Work

There is a large body of work in verification of logical properties of both sequential and concurrent programs (see [9] 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 [16, 5] verify timed properties of systems using discrete-time [15] or real-time [4] 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. [12] have applied sequentialization as well to verify periodic programs. Their key idea is to share a single stack between all tasks and model preemptions by function calls. They do not report on an implementation. In contrast, we use a sequentialization based on rounds, and present an efficient implementation and empirical evaluation. They also use the idea that a periodic program is equivalent to unbounded repetition of its hyper-period. However, we show that this is unsound in general and provide sufficient conditions under which this is sound.

In the context of concurrent software verification, several flavors of sequentialization have been proposed and evaluated (e.g., [14, 21, 11, 10]). Our procedure is closest to the LR [14] style. However, it differs from LR significantly, and provides a crucial advantage over LR for periodic programs [7] because it only considers schedules that respect the priorities. The key difference are in the notion of rounds, and the number of rounds required by the two techniques.

The sequentialization in this paper extends and advances the one presented in our earlier work [7]. We already pointed out the syntactic differences in Sect. 3. Semantically, compSeq is more complete than monoSeq. The reason is that compSeq imposes stronger restrictions on possible preemptions between jobs. Consider again our taskset from Example 1. compSeq ensures that $\mathsf{J}(\tau_1, 0)$ cannot be preempted by either $\mathsf{J}(\tau_2, 2)$ or $\mathsf{J}(\tau_2, 3)$. However, monoSeq only ensures that $\mathsf{J}(\tau_1, 0)$ is preempted by at most two jobs of τ_2. It allows, for example, an execution in which $\mathsf{J}(\tau_1, 0)$ is preempted by both $\mathsf{J}(\tau_2, 2)$ and $\mathsf{J}(\tau_2, 3)$. Indeed, we encountered a false warning eliminated by compSeq during our experiments (see Sec. 7). The ordering used by compSeq enables us to prune out infeasible executions and check for assertions violations more eagerly than in monoSeq. The early check of assertions leads to faster run-times, because CBMC creates straight-line programs up to the assertion, which means that they are now shorter in length.

A further advancement over [7] is HARMONICSEQ – a specialized version of sequentialization for harmonic programs, which extends naturally to multiple hyper-periods, allowing the reuse of variables across different hyper-periods.

7 Experiments

We have developed a tool called REKH which implements HARMONICSEQ and supports multiple hyper-periods. REKH is built on the same framework as REK. CIL [19] is used to parse and transform C programs and CBMC [8] is the sequential verifier. REKH takes as input C programs annotated with entry points of each task, their periods, worst case execution times, arrival times, and the time bound $\mathcal{W}$. The output is a sequential C program $\mathcal{S}$ that is then verified by CBMC. Our implementation of HARMONICSEQ includes support for locks, in exactly the same way as in MONOSEQ, as described in [7].

To compare between MONOSEQ and HARMONICSEQ, we have evaluated REKH on a set of benchmarks from prior work, and have conducted an additional case study by building and verifying a robotics controller simulating a Turing machine. In the rest of this section, we report on this experience. The tool, the experiments, and additional information about the case study, including the video of the robot and explanation of the properties verified, are available at: `http://www.andrew.cmu.edu/user/arieg/Rek`. All experiments have been performed on a AMD Opteron 2.3 GHz processor, 94 GB of main memory running Linux.

NXTway-GS controller. The NXTway-GS controller, nxt for short, runs on nxtOSEK [1] – a real-time operating system ported to the LEGO Mindstorms 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 a period of 48ms, that monitors a sonar sensor for obstacles, and a 96ms background task that prints debug information on an LCD screen. Note that this system is harmonic. (In [7], we used a non-harmonic variant of the system). Arrival time of all tasks were set to 0 – to model the semantics of the OSEK operating system.

We verified several versions of this controller. All of the properties (i.e., assertions) verified were in 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 for analyzing a single hyper-period (96ms) are shown in the top part of Table 1. Experiments nxt.ok1 (nxt.bug1) check that the balancer

Table 1. 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, 'N' for UNSAFE, and 'U' for timeout (12,000s); Time = verification time in sec.

		monoSeq						harmonicSeq					
Name			SAT Size			S	Time		SAT Size			S	Time
	OL	SL	GL	Var	Clause		(sec)	SL	GL	Var	Clause		(sec)
1 hyper-period													
nxt.ok1	396	2,158	12K	128K	399K	Y	21.22	2,378	17K	110K	354K	Y	4.22
nxt.bug1	398	2,158	12K	128K	399K	N	6.22	2,378	17K	110K	354K	N	4.36
nxt.ok2	388	2,215	12K	132K	410K	Y	11.16	2,432	18K	111K	356K	Y	4.69
nxt.bug2	405	2,389	15K	135K	422K	N	8.66	2,704	23K	114K	372K	N	5.81
nxt.ok3	405	2,389	15K	135K	425K	Y	14.46	2,704	23K	109K	358K	Y	5.71
aso.bug1	421	2,557	17K	167K	541K	N	12.05	3,094	29K	173K	568K	N	6.67
aso.bug2	421	2,627	17K	167K	539K	N	11.61	3,184	29K	165K	549K	N	6.71
aso.ok1	418	2,561	17K	164K	525K	Y	22.20	3,098	28K	147K	486K	Y	6.51
aso.bug3	445	3,118	24K	350K	1,117K	N	22.15	4,131	41K	341K	1,108K	Y	19.27
aso.bug4	444	3,105	23K	325K	1,027K	N	16.32	4,118	40K	307K	1,001K	N	10.83
aso.ok2	443	3,106	23K	326K	1,035K	Y	601.59	4,119	40K	311K	1,006K	Y	21.94
4 hyper-periods													
nxt.ok1	396	14,014	57K	1,825K	5,816K	Y	1,305	2,393	71K	471K	1,610K	Y	70.59
nxt.bug1	398	14,014	57K	1,825K	5,816K	N	1,406	2,393	71K	471K	1,610K	N	73.27
nxt.ok2	388	14,156	60K	1,850K	5,849K	Y	1,382	2,447	73K	475K	1,618K	Y	67.08
nxt.bug2	405	14,573	71K	1,887K	5,978K	N	362	2,722	94K	485K	1,667K	N	77.39
nxt.ok3	405	14,573	71K	1,884K	5,964K	U	—	2,722	93K	466K	1,723K	Y	101.01
aso.bug1	421	14,942	81K	2,359K	7,699K	N	894	3,115	115K	726K	2,741K	N	143.52
aso.bug2	421	15,097	81K	2,359K	7,689K	N	773	3,205	116K	692K	2,438K	N	107.66
aso.ok1	418	14,946	80K	2,331K	7,590K	U	—	3,119	114K	620K	2,188K	Y	110.21
aso.bug3	445	16,024	113K	5,016K	16,162K	N	9,034	4,161	167K	1,406K	4,774K	Y	215.02
aso.bug4	444	16,055	108K	4,729K	15,141K	N	6,016	4,148	161K	1,271K	4,295K	N	168.22
aso.ok2	443	16,056	109K	4,734K	15,159K	U	—	4,149	162K	1,289K	4,360K	Y	200.25

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. In all cases, harmonicSeq dramatically outperforms monoSeq. Furthermore, harmonicSeq declares the program safe for aso.bug3. Indeed, the program is safe under zero-phasing of OSEK. Note that it was flagged unsafe by monoSeq, a false warning, because monoSeq assumes arbitrary phasing.

We have also experimented with analyzing multiple hyper-periods. In the bottom part of Table 1, we show the result for 4 hyper-periods. Results for hyper-periods 2 and 3 are similar. In the case of aso.bug3 the performance improves by a factor of 40x. Furthermore, harmonicSeq solves all cases, while monoSeq times

out in 3 instances. We conclude that HARMONICSEQ scales better to multiple hyper-periods than MONOSEQ does.

Turing Machine Case Study. We built a robot simulating the Turing Machine (TM) using LEGO Mindstorms. While our robot is a toy, it is quite similar to many industrial robots such as ones used for metal stamping. Physically, the robot consists of a conveyer belt (the tape) and levers (for reading and writing the tape). The tape is built out of 16 L-shaped black bricks. Each brick represents a bit. A bit is flipped by the write lever. The light sensor, which is attached to the read head, approaches the tape and determines the value of the current bit by emitting green light and measuring the intensity of its reflection. Due to our design of the TM, it is possible for the write lever and the read head to collide. It is the controller's responsibility to avoid this collision (i.e., read head and write lever should never approach the tape together). The tape is placed on a rail and is moved left and right by the tape motor.

The implementation has four periodic tasks – `Controller`, `TapeMover`, `Reader`, and `Writer` in order of ascending priority. The `Controller` task has 500ms period and 440ms WCET. The other three tasks each have 250ms period and 10ms WCET respectively. The `Controller` task looks up a transition table, determines next operations to execute, and gives commands to the other tasks. The `TapeMover` task moves the tape to the left (or right). The `Reader` task moves the read head back and forth by rotating the read motor and reads the current bit of the tape. The `Writer` task rotates the write lever to flip a bit.

We model the motors and the color sensor to abstract away unnecessary complexity and verify properties of interest as follows:

- *Motor.* The speed of a motor is only accessed through the API functions. Motor's counter is modeled non-deterministically but respects the current speed, i.e., if the speed is positive, consecutive samplings of the counter increase monotonically. Effectively, we abstract away the acceleration.
- *Color sensor.* The model of the color sensor returns a non-deterministic intensity value. Additionally, it maintains two variables for the mode of the sensor – one for the current mode and one for the requested one. This reflects the actual API of the sensor. The API function `set_nxtcolorsensor()` is used to request to switch the mode. The actual transition takes relatively long time (around 440ms) and is triggered by a call to `bg_nxtcolorsensor()`.

During the case study, we developed and verified the code together. We found REKH to be scalable enough for the task and useful to find many subtle errors in early development stages. Some of the more interesting properties are summarized below:

- `ctm.ok1`: When a bit is read, all the motors are stopped to avoid mismeasurement. We added `assert(R_speed==0 && W_speed==0 && T_speed==0)` in the `Reader` task to specify this property, where `R_speed`, `W_speed`, and `T_speed` represent the speed of read, write, and tape motor respectively.

Table 2. Experimental results of concurrent Turing Machine. H = # of hyper-periods, 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, 'N' for UNSAFE, and 'U' for timeout (85,000s); Time = verification time in sec.

			monoSeq						harmonicSeq					
Name			SAT Size				S	Time	SAT Size				S	Time
	H	OL	SL	GL	Var	Clause		(sec)	SL	GL	Var	Clause		(sec)
ctm.ok1	4	613	13K	121K	2,737K	8,774K	Y	44,781	7K	111K	1,063K	3,497K	Y	93.39
ctm.ok2	4	610	13K	119K	2,728K	8,738K	Y	21,804	7K	109K	1,055K	3,467K	Y	87.60
ctm.bug2	4	611	13K	118K	2,707K	8,674K	N	2,281	7K	108K	1,047K	3,441K	N	86.18
ctm.ok3	6	612	20K	222K	6,276K	20,163K	U	—	7K	171K	1,667K	5,566K	Y	243.76
ctm.bug3	6	612	20K	214K	5,914K	19,044K	N	84,625	7K	165K	1,609K	5,383K	N	248.65
ctm.ok4	8	613	29K	333K	10,390K	33,550K	U	—	7K	222K	2,178K	7,417K	Y	534.38

- `ctm.ok2`: When a bit is read, the sensor is on green-light mode. We added `assert(get_nxtcolorsensor_mode(CSENSOR) == GREEN)` in the `Reader` task to specify this property. When we request to switch to green-light mode in the `Reader` task, it sets a flag and waits until the `Controller` task runs the background process to make the transition and clear the flag.
- `ctm.bug2`: In this case, we have the same property as `ctm.ok2`. In this implementation, however, the `Reader` task does not wait for the `Controller` task to clear the flag. Since the `Reader` task has higher priority, the `Controller` task is not able to preempt and run the background process.
- `ctm.ok3`: When the writer flips a bit, the tape motor is stopped and the read head is at the safe position to avoid a collision with the read head. We added `assert(T_speed==0 && get_count(RMOTOR)<=0)` in the `Writer` task to express this property.
- `ctm.bug3`: We have assumed that the read head is stopped as soon as it arrives at the safe position (`get_count(RMOTOR)<=0`), expressed by `assert(T_speed==0 && R_speed==0)`. However, the property does not hold of our implementation due to the sampling granularity.
- `ctm.ok4`: We verified that the writer and read motors are stopped when the tape moves by checking `assert(R_speed == 0 && W_speed == 0)` in the `TapeMover` task.

Table 2 shows the experimental results of the Turing machine. For each case, the minimum hyper-period is selected for the analysis to reach the assertion in the program. For instance, `ctm.ok4` case requires at least 8 hyper-periods to check the assertion. In all cases, compSeq dramatically outperforms monoSeq. In one case - `ctm.ok1` - the performance improves by a factor of 480x.

8 Conclusion

In this paper, we deal with the problem of verifying logical properties, such as user specified assertions, race conditions, and API usage rules, of Real-Time

Embedded Systems (RTESs). We present a technique for time-bounded verification of RTES system implemented by a periodic program in C. The novelty of the technique is in *compositional* sequentialization that takes into account inter- and intra-hyper-period temporal separation between tasks. Tasks in different hyper-periods are sequentialized separately, as well as tasks that can never interleave due to their arrival and response times. This leads to a dramatic increase in scalability of the sequentialization approach while making it more complete (i.e., reducing false positives). We have implemented the approach and illustrate it on a benchmark from [7] and on an additional case study of a robotics system.

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.[2].

References

1. nxtOSEK/JSP Open Source Platform for LEGO MINDSTORMS NXT. http://lejos-osek.sf.net.
2. RTEMS Operating System. http://www.rtems.com.
3. VxWorks Programmer's Guide.
4. R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science (TCS)*, 126(2), 1994.
5. V. A. Braberman and M. Felder. Verification of Real-Time Designs: Combining Scheduling Theory with Automatic Formal Verification. In *Proc. of FSE*, 1999.
6. CBMC website. `http://www.cprover.org/cbmc`.
7. S. Chaki, A. Gurfinkel, and O. Strichman. Time-Bounded Analysis of Real-Time Systems. In *Proc. of FMCAD*, 2011.
8. E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In K. Jensen and A. Podelski, editors, *Proc. of TACAS*, volume 2988, 2004.
9. V. D'Silva, D. Kroening, and G. Weissenbacher. A Survey of Automated Techniques for Formal Software Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(7), 2008.
10. M. Emmi, S. Qadeer, and Z. Rakamaric. Delay-Bounded Scheduling. In *Proc. of POPL*, 2011.
11. N. Ghafari, A. J. Hu, and Z. Rakamaric. Context-Bounded Translations for Concurrent Software: An Empirical Evaluation. In *Proc. of SPIN*, 2010.
12. N. Kidd, S. Jagannathan, and J. Vitek. One Stack to Run Them All - Reducing Concurrent Analysis to Sequential Analysis under Priority Scheduling. In *Proc. of SPIN*, 2010.
13. T.-W. Kuo and A. K. Mok. Load Adjustment in Adaptive Real-Time Systems. In *Proceedings of the Real-Time Systems Symposium (RTSS '91)*, 1991.
14. A. Lal and T. W. Reps. Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis. In *Proc. of CAV*, 2008.

15. F. Laroussinie, N. Markey, and P. Schnoebelen. Efficient timed model checking for discrete-time systems. *Theoretical Computer Science (TCS)*, 353(1-3), 2006.
16. K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1-2), 1997.
17. C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM (JACM)*, 20(1), 1973.
18. D. C. Locke, D. R. Vogel, L. Lucas, and J. B. Goodenough. Generic Avionics Software Specification. Technical report CMU/SEI-90-TR-8-ESD-TR-90-209, Software Engineering Institute, Carnegie Mellon University, 1990.
19. G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proc. of CC*, 2002.
20. A. Rowe, K. Lakshmanan, H. Zhu, and R. Rajkumar. Rate-harmonized scheduling and its applicability to energy management. *IEEE Trans. Industrial Informatics*, 6(3):265–275, 2010.
21. S. L. Torre, P. Madhusudan, and G. Parlato. Reducing Context-Bounded Concurrent Reachability to Sequential Reachability. In *Proc. of CAV*, 2009.