Efficient Verification of Periodic Programs using Sequential Consistency and Snapshots

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

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

Nishant Sinha
IBM Research
Email: nishant.sinha@in.ibm.com

***Abstract*—We verify safety properties of periodic programs, consisting of periodically activated threads scheduled preemptively based on their priorities. We develop an approach based on generating, and solving, a provably correct verification condition (VC). The VC is generated by adapting Lamport's sequential consistency to the semantics of periodic programs. Our approach is able to handle periodic programs that synchronize via two commonly used types of locks – priority ceiling protocol (PCP) locks, and CPU locks. To improve the scalability of our approach, we develop a strategy called snapshotting, which leads to VCs containing fewer redundant sub-formulas, and are therefore more easily solved by current SMT engines. We develop two types of snapshotting – SS-ALL snapshots all shared variables aggressively, while SS-MOD snapshots only modified variables. We have implemented our approach in a tool. Experiments on a benchmark of robot controllers indicate that SS-MOD is the best overall strategy, and even outperforms significantly the state-of-the art periodic program verifier prior to this work.**

I. Introduction

Periodic programs (PPs) are used frequently to control safety-critical systems. Thus, verifying safety (i.e., reachability) properties of PPs is an important problem [1]. They are inherently concurrent, and model checking them is difficult to scale. In recent years, a number of projects [2], [3], [4], [5], [6] have explored symbolic bounded model checking of multi-threaded programs (MTPs), i.e., concurrent programs with shared memory communication. Specifically, given a MTP $\mathcal{P}$ and a safety property ϕ, the approach is to verify $\mathcal{P} \models \phi$ using two steps: (i) VCGEN: generate a verification condition (VC), a formula $VC(P, \phi)$ that is satisfiable iff $\mathcal{P} \not\models \phi$; (ii) SAT: check if $VC(P, \phi)$ is satisfiable using an SMT solver. We call this approach "memory consistency based BMC" (BMC-MC), since the construction of $VC(P, \phi)$ is based on a specific memory consistency model.

A PP consists of a finite set of *tasks*, each executing in its own thread. However, a PP differs from a MTP in several verification-relevant ways. *First*, each task consists of an infinite sequence of *jobs*, activated periodically. A task's thread remains inactive between the completion of a job and the activation of the next one. *Second*, each task has a priority, that is inherited by its thread. Among all active threads, the one with the highest priority is scheduled – thus, scheduling is deterministic. Scheduling is also preemptive, a newly activated thread with higher priority preempts the currently executing one. *Third*, each task has a worst-case-execution-time (or, WCET) i.e., the maximum time between the arrival and completion of any job of the task, assuming it is not preempted. *Finally*, each task has an arrival time, i.e., the activation time of its first job.

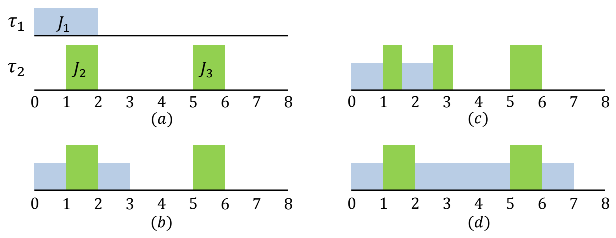

Fig. 1. (a) Example periodic program; (b) legal execution; (c,d) illegal executions; x-axis denotes time; y-axis denotes priority of executing job.

Note that, even though scheduling of a PP is deterministic, its overall behavior is non-deterministic, for two reasons. First, WCET is only an *upper-bound* on execution time. Whether a job J is preempted or not by another job J', depends on the *actual execution time* of J, which is non-deterministic. Second, we abstract away individual statement execution times, and only require that the job's WCET is not exceeded. Therefore, statements execute for a non-deterministic amount of time, and the exact *preemption location* in the control flow of J at which it is preempted by J' is non-deterministic.

We focus on "time-bounded verification" of PPs, i.e., verifying a safety property of a PP assuming it executes for time T. The time-bound fixes the number of jobs for each task, and makes the verification amenable to BMC-MC. Assuming a bound on the execution time is a useful restriction since it occurs naturally in safety-critical systems. For example, once a crash is perceived, an air bag must deploy within a time bound. Figure 1(a) shows a time-bounded PP $\mathcal{P}$ with two tasks – τ_1 and τ_2 – with priorities 1 and 2, periods 8 and 4, WCETs 2 and 1, and arrival times 0 and 1, respectively, and a time bound $T = 8$. Figure 1(b) shows a legal execution of $\mathcal{P}$. In this paper, we develop a BMC-MC approach for time-bounded verification of PPs. We address two challenges – correctness and efficiency – and perform an empirical evaluation, as discussed next.

Correctness of VCs. In current BMC-MC approaches, the construction of $VC(P, \phi)$ is based on Lamport's notion of sequential consistency [7], which we call SC-MT. However, SC-MT is *imprecise* for PPs, and cannot be used for VC generation. This imprecision arises from the combination of priority-based scheduling, WCETs, and arrival times. Consider the PP $\mathcal{P}$ shown in Figure 1(a). Note that if J_2 preempts J_1, then J_2 must complete before J_1 can resume. Recall that SC-MT assumes a non-deterministic scheduler, i.e., any active non-blocked thread is allowed to execute. Thus the execution in Figure 1(c) is impossible for the $\mathcal{P}$, while it is allowed by SC-MT. Similarly, due the arrival and WCETs of τ_1 and τ_2, it is impossible for J_3 to preempt J_1. Therefore, the execution in Figure 1(d) is illegal for $\mathcal{P}$, while it is allowed by SC-MT.

Our **first contribution** is a new method to construct $VC(P, \phi)$ based on a PP-specific notion of sequential consistency. A satisfying assignment to $VC(P, \phi)$ induces an event order corresponding to a legal execution of P. Previous works [3], [5], [6] on memory-consistency based VC generation for MTPs leverage the concept of Lamport clocks [8], which are symbolic integer-valued *timestamps* associated with each program event (i.e., an access to a shared variable). These timestamps order program events in a sequentially consistent logical timeline. However, they are not sufficient to capture all legal executions of PPs. To solve this problem, we propose *hierarchical* timestamps, which not only capture the program order and the write-read ordering as before, but also take into account the priority-based preemption semantics of PPs.

Like MTPs, PPs protect access to shared variables via locking. However, unlike MTPs, locks in PPs are implemented by *altering* thread priorities. Our **second contribution** to deal with two variants of such locks – Priority Ceiling Protocol (PCP) locks [9], and CPU locks (another variant, the Priority Inheritance lock [9], is beyond the scope of this paper). When a thread acquires such a lock, its priority is raised, which disables scheduling of other threads from which the shared resource must be protected. When a thread releases a lock, its priority is reduced correspondingly. To encode such locks, we introduce priority-test-and-set (PTAS) operations, which atomically test and update the set of acquired locks. We formalize the semantics of PTAS operations, and show how to implement PCP and CPU locks using them. We also update $VC(P, \phi)$ to handle PTAS operations in a provably correct manner. Further details are presented in Section IV.

Efficiency of Encoding. As observed in the BMC-MC literature [3], [4], [5], [10], verification conditions, if constructed naively, are intractable for even state-of-the-art SMT solvers. An effective strategy for generating tractable VCs is to reduce the set of writes to a shared variable g that could be "observed" by a read of g, where a read r observes a write w if w is the most recent write to g prior to r. For PPs, we note that the observable write sets for reads in successive jobs contain many common write events from previous job instances, which leads to a severely redundant encoding. Our **third contribution** is an efficient encoding scheme for PPs which reduces the size of observation sets via the idea of *snapshots*.

A snapshot ss of g, at a location l inside a task τ, reads the current value of g in τ and then writes the same value back atomically. Thus, by introducing a new atomic *read/write pair* for g at l, ss prevents the reads in τ following l from directly observing the writes to g prior to l. Snapshotting is useful if multiple reads following l may observe the same (or largely similar) set of prior writes: multiple write events prior to l are effectively *merged* into a single write event at l. This reduces the large (quadratic) number of write-read data flows into a small (linear) number of flows, improving efficiency of the encoding. To be beneficial, snapshots must be performed for *selective* shared variables and locations. We explore two snapshotting strategies: (i) SS-ALL: all shared variables are snapshotted at the end of every job; (ii) SS-MOD: only shared variables that could be modified by a job are snapshotted at its end. Further details are presented in Section V.

Empirical Evaluation. Our **final contribution** is an implementation of our approach in a tool called LLREK, and empirical evaluation on a benchmark comprising of PPs that implement controllers for LEGO Mindstorms robots. Our results indicate that both SS-MOD and SS-ALL outperform SS-NONE, with SS-MOD being the best overall strategy. In some cases, SS-MOD is five times faster than SS-NONE. In other cases, SS-MOD completes verification successfully while SS-ALL and SS-NONE run out of memory. This work is part of an ongoing project on developing efficient software model checkers for periodic programs. We also compared LLREK with REKH [11], the most advanced PP verifier developed by the project prior to LLREK. On our benchmark, LLREK outperforms REKH significantly (in some cases by a factor of seven), and also solves many instances for which REKH runs out of memory. Further details are presented in Section VI.

Related Work. There is a large body of work in verification of logical properties of both sequential and concurrent software (see [12] for a survey). However, these techniques abstract away time completely, by assuming a non-deterministic scheduler model. In contrast, we focus on periodic programs where scheduling is non-deterministic, and influenced by both thread priorities and timing.

A number of projects [13], [14] verify timed properties of systems using discrete-time [15] or real-time [16] semantics by abstracting away data- and control-flow completely. Instead, we focus on the verification of real implementations of periodic programs, and do not abstract data- and control-flow.

Verification of multi-threaded programs via BMC-MC [3], [4], [5], [6] has also been studied by several researchers. However, previous methods focus on constructing VCs for MTPs. These methods are incorrect for PPs, as argued earlier. The purpose of snapshotting is orthogonal to that of interference abstraction (IA) [5], commonly used in BMC-MC. IA assigns symbolic values to existing reads to decouple them from writes, while snapshotting introduces new symbolic reads to *merge* data flows arising from multiple writes on a shared variable into a single read/write unit. Merging allows the reads in the following program fragment to observe a single data source as opposed to a large number, thus improving the efficiency of the symbolic encoding significantly.

Florian et al. [1] extend the explicit-state model checker SPIN to verify periodic programs written in PROMELA. Our focus is on the verification of periodic programs at the source code level using BMC-MC, which is a symbolic approach.

Time-bounded verification of PPs via sequentialization was proposed by Chaki et al. [17], and later extended to be compositional [11]. However, sequentialization-based methods for MTPs [18], [19], [20] typically rely on modeling context switches (preemptions) for thread interleavings instead of exploiting memory consistency of read/writes. Sequentialization has also been applied iteratively to verify PPs with priority inheritance locks [21]. It is possible to extend the approach in this paper in a similar manner, but this requires a non-trivial modification to the encoding. Kidd et al. [22] have applied sequentialization to verify PPs, by using function calls to model preemptions. Our encoding relies on memory consistency, and does not model preemptions explicitly. Finally, applying naive concurrency (i.e., MTP) verification to PPs result in virtually 100% of false positives, as explored in prior work [17], [11].

The rest of the paper is organized as follows. In Section II, we present basic concepts and notation. In Section III we present our basic construction of $VC(P,\phi)$. In Section IV, we show how to augment $VC(P,\phi)$ to encode PCP and CPU locks. In Section V we present snapshotting, and its two variants. In Section VI we present our empirical evaluation, and in Section VII, we conclude.

II. Preliminaries

We assume an universe bounded by time T. A task τ is a 5-tuple (J,π,P,C,A) where: (i) π is its priority; (ii) P is its period; (iii) J is a sequence of $\frac{T}{P}$ jobs; (iv) $C>0$ is its WCET; and (v) $A\geq 0$ is its arrival time. A periodic program $\mathcal{P}$ is a finite sequence of tasks. Consider a PP $\mathcal{P}=\langle\tau_1,\ldots,\tau_n\rangle$ such that $\tau_i=(\mathsf{J}_i,\pi_i,P_i,C_i,A_i)$. We write $J_{i,j}$ to mean the the j-th job of the i-th task, i.e., $\mathsf{J}_i=\langle J_{i,1},\ldots,J_{i,|\mathsf{J}_i|}\rangle$. We assume that tasks have: (i) distinct and mutually disjoint jobs, i.e., $(i,j)\neq(i',j') \implies J_{i,j}\neq J_{i',j'}$; and (ii) distinct priorities $i\neq i' \implies \pi_i\neq\pi_{i'}$. Let RT_i be the response time of τ_i, i.e., the time required by any job of τ_i to complete, assuming maximal preemption by other tasks. Note that RT_i is statically computable via Rate-Monotonic Analysis [23]. We assume that the first job of τ_i always completes before time P_i, i.e., $A_i+RT_i\leq P_i$. It can be shown that $RT_i\geq C_i$, which implies that $RT_i>0$ and $P_i>0$.

Job Orderings. Let $\mathcal{J}$ be the set of all jobs. We define two relations $\sqsubset$ (*finishes-before*) and $\uparrow$ (*may preempt*) over $\mathcal{J}$ to characterize the order between jobs. Each job $J_{i,j}$ has a priority $\pi(J_{i,j})=\pi_i$, arrival time $A(J_{i,j})=A_i+(j-1)\times P_i$, and departure time $D(J_{i,j})=A(J_{i,j})+RT_i$. Then:

$$J_1\sqsubset J_2 \iff \begin{array}{c}(\pi(J_1)\leq\pi(J_2)\wedge D(J_1)\leq A(J_2))\vee\\(\pi(J_1)>\pi(J_2)\wedge A(J_1)\leq A(J_2))\end{array} \quad (1)$$

$$J_1\uparrow J_2 \iff \pi(J_1)<\pi(J_2)\wedge A(J_1)<A(J_2)<D(J_1) \quad (2)$$

Note that $J_1\sqsubset J_2$ means that J_1 always completes before J_2 starts, and $J_1\uparrow J_2$ means that it is possible for J_1 to be preempted by J_2. Since $RT_i\leq P_i$, earlier jobs of a task always finish before later jobs of the same task, i.e., $\forall i\in[1,n]\centerdot\forall 1\leq j<j'\leq|\mathsf{J}_i|\centerdot J_{i,j}\sqsubset J_{i,j'}$. Also $A(J)<D(J)$.

States and Events of PPs. We assume a denumerable set G of $\mathbb{D}$-valued shared variables; $\mathbb{D}$ contains a distinguished value $\perp$. Function $\mathcal{I}:G\mapsto\mathbb{D}$ maps shared variables to their initial values. Let $\mathbb{Z}$ be the set of integers. An action α is a 4-tuple (J,pc,η,g) and an event ϵ is a pair (α,v) such that $J\in\mathcal{J}$, $pc\in\mathbb{Z}$, $\eta\in\{r,w\}$, $g\in G$ and $v\in\mathbb{D}$. Let $\mathsf{J}(\alpha)=\mathsf{J}(\epsilon)=J$, $\pi(\alpha)=\pi(\epsilon)=\pi(J)$, $\eta(\alpha)=\eta(\epsilon)=\eta$, $g(\alpha)=g(\epsilon)=g$, and $v(\epsilon)=v$. Events $((J,pc,r,g),v)$ and $((J,pc,w,g),v)$ denote, respectively, that value v is read from and written to variable g by job J at location pc.

Action $(J,\triangleright)$ and event $((J,\triangleright),\perp)$ denote start of job J. For $\alpha=(J,\triangleright)$, and $\epsilon=((J,\triangleright),\perp)$, $\mathsf{J}(\alpha)=\mathsf{J}(\epsilon)=J$, $\pi(\alpha)=\pi(\epsilon)=\pi(J)$, $\eta(\alpha)=\eta(\epsilon)=\triangleright$. Similarly, action $(J,\triangleleft)$ and event $((J,\triangleleft),\perp)$ denote termination of job J. For $\alpha=(J,\triangleleft)$, and $\epsilon=((J,\triangleleft),\perp)$, $\mathsf{J}(\alpha)=\mathsf{J}(\epsilon)=J$, $\pi(\alpha)=\pi(\epsilon)=\pi(J)$, $\eta(\alpha)=\eta(\epsilon)=\triangleleft$.

Note that we use different fonts for J to denote different things. In general, J (or J_x) denotes a specific job, J (or J_x) denotes a set of jobs, while $\mathsf{J}(\cdot)$ is a function that maps actions and events to their corresponding jobs.

Job Alphabet and Program Order. Each job J has an alphabet of read actions $\Sigma_r(J)\subseteq\{J\}\times\mathbb{Z}\times\{r\}\times G$, and write actions $\Sigma_w(J)\subseteq\{J\}\times\mathbb{Z}\times\{w\}\times G$. Let $\Sigma(J)=\Sigma_r(J)\cup\Sigma_w(J)\cup\{(J,\triangleright),(J,\triangleleft)\}$. Let $PO(J)$ be a partial order over $\Sigma(J)$, representing control flow. We write $\alpha\xrightarrow{J}\alpha'$ to mean $(\alpha,\alpha')\in PO(J)$. Thus, $\forall\alpha\in\Sigma_r(J)\cup\Sigma_w(J)\centerdot(J,\triangleright)\xrightarrow{J}\alpha\xrightarrow{J}(J,\triangleleft)$. Let $\mathbb{J}$ be a linearization of $\Sigma(J)$ consistent with $PO(J)$, and $\iota(\alpha)$ be the index of α in $\mathbb{J}$. In particular, $\iota(J,\triangleright)=1$, and $\iota(J,\triangleleft)=|\Sigma(J)|$.

Timed Event Sequences. The valid executions of periodic programs are characterized by timed event sequences (TES). Formally, a TES is a sequence $\langle(\epsilon_1,t_1),\ldots,(\epsilon_k,t_k)\rangle$ where ϵ_i is an event, and t_i is a real-valued timestamp. For TESs e_1 and e_2, $e_1\oplus e_2$ is the set of TESs obtained via their arbitrary interleaving, and $e_1\odot e_2$ is their concatenation. Operations $\oplus$ and $\odot$ extend naturally to sets of TESs. Let $PriorWr(e,i)$ be the indices of events in e prior to ϵ_i that write to $g(\epsilon_i)$, i.e., $PriorWr(e,i) = \{j\in[1,i)\mid\eta(\epsilon_j)=w\wedge g(\epsilon_j)=g(\epsilon_i)\}$. Then, $LastWr(e,i)$ is last value written to $g(\epsilon_i)$ prior to ϵ_i, or $\mathcal{I}(g(\epsilon_i))$ if no such write exists, i.e., if $PriorWr(e,i)=\emptyset$ then $LastWr(e,i)=\mathcal{I}(g(\epsilon_i))$ else $LastWr(e,i)=v(\epsilon_m)$ where $m=\max(PriorWr(e,i))$.

Job Semantics. The semantics of J, denoted $[\![J]\!]$, is a set of TESs. Formally, $\langle(\epsilon_1,t_1),\ldots,(\epsilon_k,t_k)\rangle\in[\![J]\!]$ if: (i) $\forall i\in[1,k]\centerdot\mathsf{J}(\epsilon_i)=J$; (ii) $A(J)\leq t_1<t_2<\cdots<t_k\leq D(J)$; and (iii) if $\forall i\in[1,k]\centerdot\epsilon_i=(\alpha_i,v_i)$, then the sequence of actions $\langle\alpha_1,\ldots,\alpha_k\rangle$ respects the program order $PO(J)$, i.e., $\alpha_1=(J,\triangleright)$, $\alpha_k=(J,\triangleleft)$, and $\forall i\in[1,k)\centerdot\alpha_i\xrightarrow{J}\alpha_{i+1}$. For example, suppose the body of job J_2 from our running example is described by the control-flow-graph shown in Figure 2(c). Then $[\![J_2]\!]$ contains all TESs of the form $\langle(((J_2,\triangleright),\perp),t_1),(((J_2,1,r,g),v_1),t_2),(((J_2,pc,w,g),v_2),t_3),(((J_2,\triangleleft),\perp),t_4)\rangle$ such that: (i) $1\leq t_1<t_2<t_3<t_4\leq 2$; (ii) $(v_1<0\wedge pc=3\wedge v_2=v_1+7)\vee(v_1\geq 0\wedge pc=2\wedge v_2=v_1\times 5)$.

Task Semantics. The semantics of τ_i, denoted $[\![\tau_i]\!]$, is the set of TESs: $\bigodot_{j=1}^{|\mathsf{J}_i|}[\![J_{i,j}]\!]$. Thus, each execution of τ_i is a concatenation of an execution from each of its jobs. The

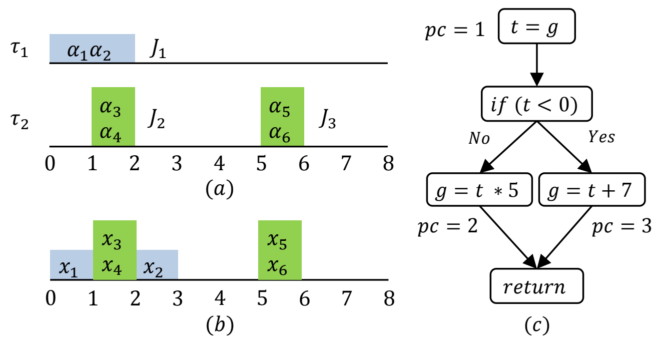

Fig. 2. (a) Periodic program; (b) Execution; (c) Control-Flow Graph.

semantics of $\mathcal{P}$, denoted $[\![\mathcal{P}]\!]$, is also a set of TESs. Formally, $e = \langle(\epsilon_1, t_1), \dots, (\epsilon_k, t_k)\rangle \in [\![\mathcal{P}]\!]$ if:

$$(a)\ e \in \bigoplus_{i=1}^{n} [\![\tau_i]\!] \qquad (b)\ \forall i \in [1,k) \centerdot t_i < t_{i+1} \tag{3}$$

$$\forall 1 \le i < j \le k \centerdot \neg(\mathsf{J}(\epsilon_j) \sqsubset \mathsf{J}(\epsilon_i)) \tag{4}$$

$$\forall 1 \le i \le j \le h \le k \centerdot \mathsf{J}(\epsilon_i) = \mathsf{J}(\epsilon_h) \implies \pi(\epsilon_i) \le \pi(\epsilon_j) \tag{5}$$

$$\forall i \in [1,k] \centerdot \eta(\epsilon_i) = r \implies v(\epsilon_i) = \mathit{LastWr}(e,i) \tag{6}$$

Informally, (3) states that e is an interleaving of executions of tasks in $\mathcal{P}$ with non-decreasing timestamps; (4) enforces job ordering; (5) enforces priority based preemptive scheduling; and (6) states that the last written value is always read.

III. VC Generation for Periodic Programs

Hierarchical Clock. The concept of a hierarchical Lamport clock is fundamental to our VCGen algorithm. To understand this idea, consider the PP shown in Figure 2(a). It is the same as in Figure 1(a), except that we have added actions, with program ordering, to the jobs. Specifically $\Sigma(J_1) = \{\alpha_1, \alpha_2\}$, $\Sigma(J_2) = \{\alpha_3, \alpha_4\}$, and $\Sigma(J_3) = \{\alpha_5, \alpha_6\}$, with program order $\alpha_1 \overset{J_1}{\to} \alpha_2$, $\alpha_3 \overset{J_2}{\to} \alpha_4$, and $\alpha_5 \overset{J_3}{\to} \alpha_6$. Now consider a legal execution of the PP shown in Figure 2(b), where $\forall i \in [1,6] \centerdot x_i = ((\alpha_i, v_i), t_i)$. Let $\mathsf{R}(e,i)$, be the number of jobs ending before x_i. Let $\bowtie \in \{<,>\}$. Then, we observe for each (x_i, x_j):

1) If $\mathsf{R}(e,i) \bowtie \mathsf{R}(e,j)$, then $t_i \bowtie t_j$. Example pairs are (x_4, x_2) and (x_2, x_5).
2) If $\mathsf{R}(e,i) = \mathsf{R}(e,j) \wedge \pi(\alpha_i) \bowtie \pi(\alpha_j)$, then $t_i \bowtie t_j$. An example is (x_1, x_3).
3) If $\mathsf{R}(e,i) = \mathsf{R}(e,j) \wedge \pi(\alpha_i) = \pi(\alpha_j)$ (note this implies $\mathsf{J}(\alpha_i) = \mathsf{J}(\alpha_j)$), but $\iota(\alpha_i) \bowtie \iota(\alpha_j)$, then $t_i \bowtie t_j$. Example pairs are (x_3, x_4) and (x_5, x_6).

The above observations imply that, for the TES in Figure 2(b), the ordering of x_i's by their timestamps t_i's equals their lexicographic ordering by the tuple $(\mathsf{R}(e,i), \pi(\alpha_i), \iota(\alpha_i))$. Thus, $(\mathsf{R}(e,i), \pi(\alpha_i), \iota(\alpha_i))$ is a logical representation of the timestamp t_i of event (α_i, v_i). Our key insight is that this holds for arbitrary PPs and their legal executions. In the rest of this section, we formalize this insight, use it to construct the VC for an arbitrary PP, and prove its correctness.

VCGen for Jobs. We assume that for any job J, there exists a bit-vector logic formula $VC(J)$ over the set of predicates $En(J) = \{En(\alpha) \mid \alpha \in \Sigma(J)\}$, and terms $V(J) = \{V(\alpha) \mid \alpha \in \Sigma_r(J) \cup \Sigma_w(J)\}$ such that the following holds.

Fact 1 (Job Verification Condition)**.** *For any* $\{\alpha_1, \dots, \alpha_k\} \subseteq \Sigma(J)$*, and sequence* $\langle v_1, \dots, v_k\rangle \in \mathbb{D}^k$*, the formula* $VC(J) \wedge \bigwedge_{i=1}^{k}(En(\alpha_i) \wedge V(\alpha_i) = v_i)$ *is satisfiable iff* $\forall A(J) \le t_1 < \dots < t_k \le D(J) \centerdot \langle((\alpha_1, v_1), t_1), \dots, ((\alpha_k, v_k), t_k)\rangle \in [\![J]\!]$.

Thus, every satisfying assignment of $VC(J) \wedge \bigwedge_{i=1}^{k}(En(\alpha_i) \wedge V(\alpha_i) = v_i)$ corresponds to a legal execution of J. If J is a C function – without unbounded loops, recursion and dynamic memory – $VC(J)$ can be constructed polynomially [24]. The VC of $\mathcal{P}$ is also a bit-vector formula, and consists of three sub-VCs: (i) VC_{seq} captures the thread local behavior of each task; (ii) VC_{clk} orders events into a total order along a logical timeline; and (iii) VC_{obs} relates the read and write events on shared variables so that they are sequentially consistent. Formally,

$$VC(\mathcal{P}) = VC_{seq} \wedge VC_{clk} \wedge VC_{obs} \quad \text{, where} \tag{7}$$

$$VC_{seq} = \bigwedge_{J \in \mathcal{J}} VC(J) \tag{8}$$

and VC_{clk} and VC_{obs} are presented below. In the following, Σ_r denotes $\bigcup_{J \in \mathcal{J}} \Sigma_r(J)$, Σ_w denotes $\bigcup_{J \in \mathcal{J}} \Sigma_w(J)$, and Σ denotes $\bigcup_{J \in \mathcal{J}} \Sigma(J)$. All terms have bit-vector type.

The Clock VC: VC_{clk}**.** For each $\alpha \in \Sigma$, let term $R(\alpha)$ denote the round of α. Following our intuition, we write $\kappa(\alpha)$ to mean $(R(\alpha), \pi(\alpha), \iota(\alpha))$, i.e., the symbolic timestamp of α. During VC construction, we can now use the predicate $\kappa(\cdot)$ to order events in a periodic program, akin to the way *happens-before* predicate is used for non-periodic, multi-threaded programs [7].

For each job J, we introduce two terms: $SR(J)$ and $ER(J)$, to represent, respectively, the earliest (i.e., start) and latest (i.e., end) round of J's execution, in which any action in $\Sigma(J)$ may occur. Then, VC_{clk} is a conjunction of the following:

$$\begin{array}{l} (a)\ \bigwedge_{J \in \mathcal{J}} \bigwedge_{\alpha \in \Sigma(J)} (SR(J) \le R(\alpha) \le ER(J)) \\ (b)\ \bigwedge_{J_1 \sqsubset J_2} ER(J_1) < SR(J_2) \end{array} \tag{9}$$

$$\bigwedge_{J_1 \uparrow J_2} \bigwedge_{\alpha \in \Sigma(J_1)} R(\alpha) \le SR(J_2) \vee R(\alpha) > ER(J_2) \tag{10}$$

Informally, (9)(a) asserts that actions respect starting and ending rounds; (9)(b) asserts that if J_1 finishes before J_2 starts then the ending round of J_1 must be less than the starting round of J_2; (10) asserts that if J_1 could be preempted by J_2, then it cannot execute while J_2 is active.

The Observation VC: VC_{obs}**.** For a read action $\alpha_r \in \Sigma_r$, let $\mathcal{W}(\alpha_r)$ be the set of write actions that α_r may observe, i.e., the set of writes to variable $g(\alpha_r)$ belonging to jobs that do not start after $\mathsf{J}(\alpha_r)$ ends. Formally:

$$\mathcal{W}(\alpha_r) = \{\alpha_w \in \Sigma_w \mid g(\alpha_w) = g(\alpha_r) \wedge \neg(\mathsf{J}(\alpha_r) \sqsubset \mathsf{J}(\alpha_w))\} \tag{11}$$

For each $\alpha_r \in \Sigma_r$, we introduce three additional variables $\tilde{R}(\alpha_r)$, $\tilde{\pi}(\alpha_r)$, and $\tilde{\iota}(\alpha_r)$. In essence, $(\tilde{R}(\alpha_r), \tilde{\pi}(\alpha_r), \tilde{\iota}(\alpha_r))$

denotes the symbolic clock of the write action observed by α_r, and is denoted by $\tilde{\kappa}(\alpha_r)$. Let $\alpha \prec \alpha'$ denote α *happens before* α', i.e., $\alpha \prec \alpha' = En(\alpha) \wedge \kappa(\alpha) < \kappa(\alpha')$. Then VC_{obs} is a conjunction of the following for each read action $\alpha_r \in \Sigma_r$:

$$En(\alpha_r) \Rightarrow \left(\bigwedge_{\alpha_w \in \mathcal{W}(\alpha_r)} \alpha_w \prec \alpha_r \Rightarrow \kappa(\alpha_w) \le \tilde{\kappa}(\alpha_r) \right) \quad (12)$$

$$En(\alpha_r) \Rightarrow \left(VC^1_{obs} \vee \bigvee_{\alpha_w \in \mathcal{W}(\alpha_r)} VC^2_{obs}(\alpha_w) \right), \text{ where} \quad (13)$$

$$VC^1_{obs} = \left(\bigwedge_{\alpha \in \mathcal{W}(\alpha_r)} \alpha \not\prec \alpha_r \right) \wedge (\mathcal{I}(g(\alpha_r)) = V(\alpha_r)) \quad (14)$$

$$VC^2_{obs}(\alpha) = \alpha \prec \alpha_r \wedge \kappa(\alpha) = \tilde{\kappa}(\alpha_r) \wedge V(\alpha) = V(\alpha_r) \quad (15)$$

Note that (12) asserts that write action observed by α_r must have executed prior to α_r and no later than any write action to the same shared variable; (13)–(15) asserts that α_r reads the value written by the write action its observes. Thus, VC_{obs} is essentially an encoding of (6).

Correctness. The correctness of $VC(\mathcal{P})$ is expressed by Theorem 1, which states essentially that every satisfying assignment of $VC(\mathcal{P}) \wedge \bigwedge_{i=1}^{k}(En(\alpha_i) \wedge V(\alpha_i) = v_i)$ corresponds to a legal execution of $\mathcal{P}$. For brevity, we defer the proof to the extended version [25] of the paper.

Theorem 1. *For any set of actions $\{\alpha_1, \ldots, \alpha_k\} \subseteq \Sigma$, and sequence of values $\langle v_1, \ldots, v_k \rangle \in \mathbb{D}^k$, the formula $VC(\mathcal{P}) \wedge \bigwedge_{i=1}^{k}(En(\alpha_i) \wedge V(\alpha_i) = v_i)$ is satisfiable iff $\exists t_1, \ldots, t_k \centerdot \langle ((\alpha_1, v_1), t_1), \ldots, ((\alpha_k, v_k), t_k) \rangle \in [\![\mathcal{P}]\!]$.*

Constructing $VC(\mathcal{P}, \phi)$. To check a property ϕ for $\mathcal{P}$, let us assume that $\mathcal{P}$ is augmented with an action $\alpha(\phi)$ such that $\mathcal{P} \models \phi$ iff no TES in $[\![\mathcal{P}]\!]$ contains the event $(\alpha(\phi), v)$ for some value v. Then, from Theorem 1, $\mathcal{P} \models \phi \iff VC(\mathcal{P}) \wedge En(\alpha(\phi))$ is unsatisfiable. Thus, $VC(\mathcal{P}, \phi) = VC(\mathcal{P}) \wedge En(\alpha(\phi))$.

IV. Handling Locks

In this section, we extend VC generation to handle acquiring and releasing of locks. We consider PPs with two kinds of locks – priority ceiling protocol (PCP) locks and CPU locks. Each PCP lock l is associated with a priority level $\pi(l)$. Acquiring l disables scheduling any task whose priority is less than $\pi(l)$. Thus, a job is executed iff it is active and its priority is higher than all other active jobs, as well as those of all PCP locks held. A CPU lock disables scheduling altogether. In the rest, we only deal with PCP locks since a CPU lock is equivalent to a PCP lock l such that $\pi(l)$ is greater than the largest task priority.

To formalize PCP locks, we introduce atomic *priority-test-and-set* (PTAS) actions. Let $\mathcal{L}$ be the set of all PCP locks. For $L \subseteq \mathcal{L}$, let $\pi(L) = \{\pi(l) \mid l \in L\}$. Formally, a PTAS action is a 5-tuple (J, pc, π_t, L_r, L_a) such that $J \in \mathcal{J}$, $pc \in \mathbb{Z}$, π_t is a priority value, $L_r \subseteq \mathcal{L}$, and $L_a \subseteq \mathcal{L}$. A PTAS event ϵ is a pair (α, L_h) such that α is a PTAS action, and $L_h \subseteq \mathcal{L}$. Informally, L_h denotes the set of locks held after ϵ occurs. PTAS actions restrict the set of legal executions of a PP. Specifically, whenever, a PTAS action (J, pc, π_t, L_r, L_a) appears on an execution, the following holds: (i) *test*: all currently held PCP locks have priority less than π_t; and (ii) *set*: locks in L_r are released, locks in L_a are acquired.

Modeling Locks. Let $\Sigma_p(J)$ be the set of PTAS actions in $\Sigma(J)$. Formally, $\Sigma_p(J) = \{sched(J)\} \cup \bigcup_{l \in \mathcal{L}}(lock(J, l) \cup unlock(J, l))$, where: $sched(J) = (J, 0, \pi(J), \emptyset, \emptyset)$, $lock(J, l) \subseteq \{(J, pc, \max(\pi(\mathcal{L})) + 1, \emptyset, \{l\}) \mid pc \in \mathbb{Z}\}$, and $unlock(J, l) \subseteq \{(J, pc, \max(\pi(\mathcal{L})) + 1, \{l\}, \emptyset) \mid pc \in \mathbb{Z}\}$. Action $sched(J)$ denotes the scheduling of J for the first time. Actions in $lock(J, l)$ and $unlock(J, l)$ are used, respectively, to acquire and release lock l. Program order $PO(J)$ satisfies:

$$\forall \alpha \in \Sigma(J) \setminus \{sched(J), (J, \triangleright)\} \centerdot (J, \triangleright) \xrightarrow{J} sched(J) \xrightarrow{J} \alpha \quad (16)$$

Note that this means on any execution of J, $sched(J)$ appears before every other action in $\Sigma(J)$, except for $(J, \triangleright)$. Every TES $e \in [\![\mathcal{P}]\!]$ also satisfies the following condition. Let there be k PTAS events in e, and $\tilde{\epsilon}_i = ((J^i, pc^i, \pi^i_t, L^i_r, L^i_a), L^i_h)$ be the i-th PTAS event in e. Then:

$$L^1_h = L^1_a \quad \bigwedge \quad \forall i \in (1, k] \centerdot L^i_h = L^{i-1}_h \setminus L^i_r \cup L^i_a \quad (17)$$

$$\forall i \in (1, k] \centerdot \max(\pi(L^{i-1}_h)) < \pi^i_t \quad (18)$$

Note that (16)–(18) imply that J is scheduled only if the priority of J is higher than all PCP locks held. The CPU lock has priority $\max(\{\pi(J) \mid J \in \mathcal{J}\}) + 1$.

Updated Construction of $VC(\mathcal{P})$. Let $\Sigma_p = \bigcup_{J \in \mathcal{J}} \Sigma_p(J)$. When constructing VC_{seq}, we treat each $\alpha \in \Sigma_p$ as a *NOP*. The construction of VC_{clk} uses the augmented $\Sigma(J)$ containing the additional PTAS actions. The construction of VC_{obs} is updated as follows. For each $\alpha \in \Sigma_p$, we add the following terms: $R(\alpha)$, $\tilde{R}(\alpha)$, $\tilde{\pi}(\alpha)$, $\tilde{\iota}(\alpha)$, and $V(\alpha)$. Their meaning is the same as for other events, except that $V(\alpha)$ now represents the set of PCP locks held after α occurs. Also, we define $\mathcal{W}(\alpha)$, i.e., the set of actions that α may observe, to contain all other PTAS actions belonging to jobs that do not start after $\mathtt{J}(\alpha)$ finishes. Formally:

$$\mathcal{W}(\alpha) = \{\alpha' \in \Sigma_p \mid \alpha' \neq \alpha \wedge \neg(\mathtt{J}(\alpha) \sqsubset \mathtt{J}(\alpha'))\} \quad (19)$$

Then VC_{obs} contains the following additional constraints for each $\alpha = (J, pc, \pi_t, L_r, L_a) \in \Sigma_p$:

$$En(\alpha) \implies \left(\bigwedge_{\alpha' \in \mathcal{W}(\alpha)} \alpha' \prec \alpha \implies \kappa(\alpha') \le \tilde{\kappa}(\alpha) \right) \quad (20)$$

$$En(\alpha) \implies \left(VC^3_{obs} \vee \bigvee_{\alpha' \in \mathcal{W}(\alpha)} VC^4_{obs}(\alpha') \right), \text{ where} \quad (21)$$

$$VC^3_{obs} = \left(\bigwedge_{\alpha' \in \mathcal{W}(\alpha)} \alpha' \not\prec \alpha \right) \wedge (V(\alpha) = \pi(L_a)) \quad (22)$$

$$VC^4_{obs}(\alpha') = \begin{pmatrix} \alpha' \prec \alpha \wedge \kappa(\alpha') = \tilde{\kappa}(\alpha) \wedge \\ \max(\pi(V(\alpha'))) < \pi_t \wedge \\ V(\alpha) = V(\alpha') \setminus L_r \cup L_a \end{pmatrix} \quad (23)$$

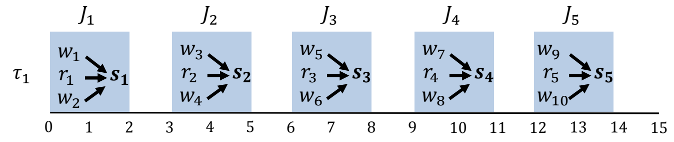

Fig. 3. Example periodic program to illustrate snapshotting.

Note that (20)–(23) assert that the PTAS action observed by α must be the last PTAS action that executed prior to α; (21)–(23) further asserts the semantics of PTAS actions is respected. Thus, (20)–(23) encode (18). We claim that Theorem 1 is valid even for the new $VC(\mathcal{P})$. The proof of this claim is in the extended version [25] of the paper.

V. Snapshotting Shared Variables

In this section, we present snapshotting of shared variables. To understand what snapshotting is, and why it is important, consider the PP in Figure 3. It consists of 1 task τ_1 with 5 jobs $J_1, \ldots, J_5$. Consider initially only the read and write actions $r_1, \ldots, r_5, w_1, \ldots, w_{10}$, and for each read, the set of writes it may observe. Then, we have: $\mathcal{W}(r_1) = \{w_1, w_2\}, \mathcal{W}(r_2) = \{w_1, \ldots, w_4\}, \ldots, \mathcal{W}(r_5) = \{w_1, \ldots, w_{10}\}$. In general, $\mathcal{W}(r_i) = \{w_1, \ldots, w_{2\times i}\}$. Recall – from (12)–(15) – that VC_{obs} encodes, for each r_i, the most recent write in $\mathcal{W}(r_i)$ prior to r_i. However, since $\mathcal{W}(r_{i-1}) \subseteq \mathcal{W}(r_i)$, the problem for r_{i-1} (and indeed for all $j < i$) is re-encoded (and resolved by the SMT solver) as part of the problem for r_i.

Snapshotting eliminates much of this redundant encoding and solving. Semantically, a snapshot of shared variable g in job J appears after every write to g in the program order of J, and *atomically* reads the value of g and writes the same value back to g. In Figure 3, these actions are shown as $s_1, \ldots, s_5$[1]. A snapshot *dominates* every other write to g in its job, and therefore eliminates them from being observed by future reads. At the same time, it may observe these writes, and snapshots in other jobs. With the snapshots added to Figure 3, we now have: $\mathcal{W}(s_1) = \mathcal{W}(r_1) = \{w_1, w_2\}, \mathcal{W}(s_2) = \mathcal{W}(r_2) = \{s_1, w_3, w_4\}, \ldots, \mathcal{W}(s_5) = \mathcal{W}(r_5) = \{s_4, w_9, w_{10}\}$. Note how the problem for r_i is solved only once (for s_i), and then the solution for s_i is reused for all $j > i$. Empirically, snapshotting leads to significantly improved (see Section VI) verification time.

Formalism. We define a function $\mathit{Snaps} : \mathcal{J} \mapsto 2^G$. Informally, $\mathit{Snaps}(J)$ is the set of shared variables snapshotted by job J. The alphabet $\Sigma(J)$ of J is augmented with snapshot actions: $\Sigma_s(J) = \{(J, s, g) \mid g \in \mathit{Snaps}(J)\}$. Let $\Sigma_s(J) = \langle \alpha_s^1, \ldots, \alpha_s^k \rangle$. The program order $PO(J)$ is augmented with:

$$\forall \alpha \in \Sigma(J) \backslash (\Sigma_s(J) \cup \{(J, \lhd)\}) \centerdot \alpha \xrightarrow{J} \alpha_s^1 \ldots \xrightarrow{J} \ldots \alpha_s^k \xrightarrow{J} (J, \lhd) \tag{24}$$

Thus, every execution in $[\![J]\!]$ snapshots all variables in $\mathit{Snaps}(J)$, and snapshot events appear after all other events, except for $(J, \lhd)$. Both reads and snapshots observe the last written values. Formally (6) is replaced by:

$$\forall i \in [1, k] \centerdot \eta(\epsilon_i) \in \{r, s\} \implies v(\epsilon_i) = \mathit{LastWr}(e, i) \tag{25}$$

Updated Construction of $VC(\mathcal{P})$. Let Σ_s be the set of snapshot actions, i.e., $\Sigma_s = \bigcup_{J \in \mathcal{J}} \Sigma_s(J)$. When constructing VC_{seq}, we treat each $\alpha \in \Sigma_s$ as a *NOP*. The construction of VC_{clk} uses the augmented $\Sigma(J)$ containing the additional snapshot actions. The construction of VC_{obs} is updated as follows. For every action $\alpha_r \in \Sigma_r \cup \Sigma_s$, we define $\mathcal{W}(\alpha_r)$, i.e., the set of actions that α_r may observe, as follows. For every job J, and shared variable g, let $\Psi_{\sqsubset}(J, g)$ be the maximal set of g-snapshotting jobs less than J according to the $\sqsubset$ order, i.e.,

$$\begin{aligned} &\Psi_{\sqsubset}(J, g) = \{J' \in \mathcal{J} \mid g \in \mathit{Snaps}(J') \wedge J' \sqsubset J \bigwedge \\ &\forall J'' \in \mathcal{J} \centerdot g \in \mathit{Snaps}(J'') \wedge J'' \sqsubset J \implies \neg(J' \sqsubset J'')\} \end{aligned} \tag{26}$$

Let $\Psi_{\uparrow}(J, g)$ be the set of jobs that can preempt J and also snapshot g, and $\Psi_{\downarrow}(J)$ be the set of jobs that can be preempted by J, and J itself, i.e.,

$$\Psi_{\uparrow}(J, g) = \{J' \in \mathcal{J} \mid g \in \mathit{Snaps}(J') \wedge J \uparrow J'\} \tag{27}$$

$$\Psi_{\downarrow}(J) = \{J' \in \mathcal{J} \mid J' = J \vee J' \uparrow J\} \tag{28}$$

Let $\alpha_r = (J, \eta, g)$. Then $\mathcal{W}(\alpha_r)$ consists of: (i) snapshots by jobs in $\Psi_{\sqsubset}(J, g)$ and $\Psi_{\uparrow}(J, g)$; and (ii) writes by jobs in $\Psi_{\downarrow}(J)$. Formally:

$$\begin{aligned} \mathcal{W}(\alpha_r) = \{(J', s, g) \mid J' \in \Psi_{\sqsubset}(J, g) \cup \Psi_{\uparrow}(J, g)\} \bigcup \\ \{(J', w, g) \mid J' \in \Psi_{\downarrow}(J)\} \end{aligned} \tag{29}$$

Finally, VC_{obs} contains the constraints defined in (12)–(15) for each $\alpha_r \in \Sigma_r \cup \Sigma_s$. Note that this means that a read or snapshot action α_r observes the last write or snapshot action to $g(\alpha_r)$ that executed prior to α_r. We claim that Theorem 1 also holds for the new $VC(\mathcal{P})$. The proof of this claim is in the extended version [25] of the paper.

We have implemented two variants of snapshotting – SS-ALL and SS-MOD – which differ in the set of variables snapshotted. For SS-ALL, all shared variables are snapshotted at the end of each job, i.e., $\mathit{Snaps}(J) = G$. For SS-MOD, only shared variables that are written by a job are snapshotted by it, i.e., $\mathit{Snaps}(J) = \{g \mid (J, w, g) \in \Sigma(J)\}$. We denote by SS-NONE the strategy of no snapshotting, presented in earlier sections. Next, we evaluate snapshotting empirically.

VI. Empirical Validation

We implemented our approach in a tool called LLREK, on top of UFO [26] and LLVM [27]. The input to LLREK is a PP $\mathcal{P}$ written in C, with jobs implemented via C functions, and periods, priorities etc. specified via macros. The safety property ϕ is expressed as an assertion in the job code. LLREK constructs the verification condition $VC(\mathcal{P}, \phi)$, as described earlier, and solves it using STP [28]. All experiments were performed on a machine running at 2.9GHz with a memory limit of 2GB and a time limit of 60 minutes. Our tools and benchmark are available at andrew.cmu.edu/~schaki/misc/llrek.tgz.

[1]For simplicity, we view a snapshot as either a read or a write, based on the context.

Name	Time (in seconds)				SAT Vars (in 1000s)				SAT Clauses (in 1000s)				AVGOBS($\mathcal{P}$)			$\lvert W(\mathcal{P})\rvert$		
	NONE	ALL	MOD	REKH	NONE	ALL	MOD	REKH	NONE	ALL	MOD	REKH	NONE	ALL	MOD	NONE	ALL	MOD
nxt.bug1:H1	33	9	**7**	18	612	234	223	698	3252	1029	985	2642	25.6	2.9	2.9	298	455	416
nxt.bug2:H1	32	10	**7**	31	642	250	235	710	3394	1091	1030	2684	26.5	3.1	3.2	310	492	429
nxt.ok1:H1	19	**7**	8	17	612	234	223	698	3252	1030	986	2642	25.6	2.9	2.9	298	455	416
nxt.ok2:H1	20	7	**6**	29	611	234	223	699	3246	1029	985	2645	25.4	3.0	2.9	298	454	415
nxt.ok3:H1	30	8	**6**	31	642	250	235	709	3394	1091	1030	2675	26.5	3.1	3.2	310	492	429
aso.bug1:H1	29	**9**	**9**	34	636	274	249	737	3346	1198	1090	2796	26.0	3.6	3.6	304	512	427
aso.bug2:H1	28	10	**9**	32	646	277	251	734	3399	1211	1100	2780	26.4	3.7	3.7	308	516	431
aso.bug3:H1	29	13	**11**	80	690	305	270	958	3608	1324	1171	3660	25.5	3.6	3.5	355	615	504
aso.bug4:H1	32	17	**9**	66	649	306	265	891	3412	1357	1168	3396	26.5	4.6	4.4	309	543	434
aso.ok1:H1	32	11	**10**	32	658	286	261	726	3458	1255	1148	2746	27.1	4.1	4.2	311	519	434
aso.ok2:H1	38	29	**17**	67	651	307	265	893	3421	1360	1170	3406	26.5	4.6	4.4	311	545	436
nxt.bug1:H4	*	119	**74**	*	*	1096	1046	*	*	4897	4681	10696	99.5	3.0	3.0	1192	1835	1676
nxt.bug2:H4	*	172	**92**	*	*	1177	1105	*	*	5214	4916	10877	102.9	3.1	3.2	1240	1989	1731
nxt.ok1:H4	*	89	**49**	*	*	1096	1046	*	*	4898	4682	10696	99.5	3.0	3.0	1192	1835	1676
nxt.ok2:H4	*	125	**49**	*	*	1096	1046	*	*	4897	4682	10708	99.3	3.0	3.0	1192	1834	1675
nxt.ok3:H4	*	358	**133**	*	*	1177	1105	*	*	5213	4916	10830	102.9	3.1	3.2	1240	1989	1731
aso.bug1:H4	*	128	**92**	*	*	1301	1177	*	*	5773	5231	11394	99.9	3.6	3.6	1216	2072	1723
aso.bug2:H4	*	147	**74**	*	*	1316	1189	*	*	5840	5283	11316	101.6	3.7	3.7	1232	2088	1739
aso.bug3:H4	*	209	**136**	*	*	1452	1280	*	*	6408	5647	*	98.3	3.6	3.5	1420	2490	2034
aso.bug4:H4	*	329	**152**	*	*	1465	1261	*	*	6579	5645	*	100.4	4.6	4.4	1236	2199	1751
aso.ok1:H4	*	270	**210**	*	*	1359	1237	*	*	6061	5523	11151	103.2	4.1	4.2	1244	2100	1751
aso.ok2:H4	*	*	**1312**	*	*	1469	1264	*	*	6597	5659	*	100.1	4.6	4.4	1244	2207	1759
ctm.bug2	36	29	**21**	105	656	429	336	719	3253	1822	1448	2801	17.9	4.1	4.5	512	1052	683
ctm.bug3	*	124	**59**	258	*	705	554	1098	*	3066	2439	4389	26.6	4.1	4.5	768	1588	1033
ctm.ok1	23	37	**21**	122	668	434	341	730	3309	1839	1466	2845	18.6	4.1	4.6	512	1052	684
ctm.ok2	28	26	**17**	111	657	431	338	724	3261	1829	1455	2823	18.1	4.1	4.5	512	1052	683
ctm.ok3	*	116	**53**	275	*	714	567	1124	*	3106	2497	4485	27.9	4.1	4.5	780	1600	1057
ctm.ok4	*	320	**143**	395	*	959	760	1410	*	4184	3356	5713	36.4	4.2	4.7	1040	2140	1400

TABLE I
EXPERIMENTAL RESULTS; * = MEMORYOUT OR TIMEOUT; VARS = # OF SAT VARIABLES; CLAUSES = # OF SAT CLAUSES; BEST NUMBERS ARE IN BOLD.

Benchmark. Our benchmark consist of a set of PPs for controlling two LEGO Mindstorms robots – a two-wheel self-balancing robot (http://lejos-osek.sourceforge.net/nxtway_gs.htm), and a metal-stamping robot (http://www.cs.cmu.edu/~soonhok/blog/building-a-lego-turing-machine). The self-balancing robot controllers come in two variants. Some – named `nxt.*` in our tables – have three 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. Others – named `aso.*` – have the functionality for monitoring bluetooth refactored out into the `Background` task.

The Turing Machine examples are named `ctm.*` and have 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. In each case, we have safety properties (whose violations lead to potential collisions between the robot and an obstacle, or between different arms of the robot etc.) encoded as assertions, and both buggy and safe versions – named `*.bug*` and `*.ok*` – of the controller w.r.t. these assertions.

Evaluation of Snapshotting. Our first set of experiments were aimed at evaluating the three snapshotting strategies – SS-NONE, SS-ALL, SS-MOD. Our results are show in Table I. The first column shows the experiment name. For the `nxt.*` and `aso.*` example, `Hk` indicates that the time-bound T was set to equal `k` hyper-periods (i.e., $T = \mathtt{k} \times 96$) of the PP. The next three columns show the verification time $Time$, and the number of variables $Vars$ and clauses $Clauses$ of the final SAT formula solved by STP after simplifying and bit-blasting the SMT formula – for each snapshotting strategy.

These results indicate that SS-MOD is the best overall strategy. In all but one instance, it is the fastest. Sometimes it is more than twice as fast as the next best strategy SS-ALL. The worst choice is SS-NONE which runs out of memory in many instances, while both SS-ALL and SS-MOD complete successfully. These trends are mirrored when we consider $Vars$ and $Clauses$, suggesting that snapshotting effectively eliminates a lot of redundancy in the SMT formulas generated by SS-NONE, with SS-MOD producing the most compact SAT formula overall. Next, we present a more direct quantitative evaluation of the effectiveness of snapshotting.

Observation Set Redundancy. Let $W(\mathcal{P})$ be the set of output (write or snapshot) actions in a PP $\mathcal{P}$. For each $w \in W(\mathcal{P})$, let $Obs(w)$ be the set of input (read or snapshot) actions that may observe w. Thus, $Obs(w) = \{\alpha \mid w \in \mathcal{W}(\alpha)\}$. Let AVGOBS($\mathcal{P}$) be the mean of the set $\{|Obs(w)| \mid w \in W(\mathcal{P})\}$. A smaller value of AVGOBS($\mathcal{P}$) indicates lower redundancy in the observation sets of $\mathcal{P}$. Here, redundancy means that a single output action may be observed by multiple input actions. Table I shows the values of AVGOBS($\mathcal{P}$) and $|W(\mathcal{P})|$ for each $\mathcal{P}$ in our benchmark and for each snapshotting strategy. As expected, AVGOBS($\mathcal{P}$) is much smaller for SS-MOD and SS-ALL compared to SS-NONE (sometimes by a factor of over 30), indicating that snapshotting reduces redundancy in observation sets significantly. Both SS-MOD and SS-ALL have similar values of $\textsc{AvgObs}(\mathcal{P})$. However, $|W(\mathcal{P})|$ is smaller for SS-MOD since it snapshots more selectively. This leads to better overall performance of SS-MOD compared to SS-ALL.

Comparison with REKH. We also compare LLREK with REKH [11]. REKH constructs a sequential (but non-deterministic) C program that is semantically equivalent to $\mathcal{P}$, and verifies it using CBMC [29] 4.5. Internally, CBMC constructs a verification condition and solves it using a SAT solver. Thus, REKH and LLREK are similar – both generate and solve verification conditions. However, they construct VCs differently: LLREK generates it directly based on sequential consistency and snapshotting, while REKH generates a C program using rounds and prophecy variables (following Lal and Reps [30]), from which the VC is constructed by CBMC. The results for REKH are also presented in Table I. They indicate that SS-ALL and SS-MOD perform better than REKH, sometimes by a factor of over seven, and often complete verification when REKH runs out of memory. Thus, LLREK is a clear and significant improvement over REKH.

VII. Conclusion

We addressed the problem of verifying safety properties of PPs. Our solution is based on the BMC-MC paradigm and consists of two steps: (i) generate a provably correct VC; (ii) solve the VC using a SMT engine. We generate the VC by adapting Lamport's sequential consistency to the semantics of PPs. Moreover, we handle PPs that synchronize via two commonly used types of locks – PCP locks, and CPU locks. To improve scalability, we develop a strategy called snapshotting, aimed at generating VCs with fewer redundant sub-formulas. We develop two snapshotting strategies – SS-ALL snapshots all shared variables, while SS-MOD only snapshots modified variables. We have implemented our approach in a tool called LLREK. Experiments indicate that snapshotting improves effectiveness of verification significantly. In particular, SS-MOD is the best strategy, and it even outperforms the state-of-art verifier for PPs. An important direction for future work is to handle additional synchronization primitives, such as priority-inheritance locks [9], and to relax the restriction of a time bound.

Acknowledgment

Copyright 2014 Carnegie Mellon University and FMCAD, Inc. [2]

References

[1] M. Florian, E. Gamble, and G. Holzmann, "Logic Model Checking of Time-Periodic Real-Time Systems," in *Proc. of Infotect@Aerospace*, 2012.

[2] I. Rabinovitz and O. Grumberg, "Bounded Model Checking of Concurrent Programs," in *Proc. of CAV*, 2005.

[3] S. Burckhardt, R. Alur, and M. M. K. Martin, "CheckFence: checking consistency of concurrent data types on relaxed memory models," in *Proc. of PLDI*, 2007.

[4] C. Wang, S. Kundu, M. K. Ganai, and A. Gupta, "Symbolic Predictive Analysis for Concurrent Programs," in *Proc. of FM*, 2009.

[5] N. Sinha and C. Wang, "Staged concurrent program analysis," in *Proc. of FSE*, 2010.

[6] J. Alglave, D. Kroening, and M. Tautschnig, "Partial Orders for Efficient Bounded Model Checking of Concurrent Software," in *Proc. of CAV*, 2013.

[7] L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers (TC)*, vol. 28, no. 9, 1979.

[8] ——, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM (CACM)*, vol. 21, no. 7, 1978.

[9] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Trans. on Comp.*, vol. 39, no. 9, 1990.

[10] N. Sinha and C. Wang, "On interference abstractions," in *Proc. of POPL*, 2011.

[11] S. Chaki, A. Gurfinkel, S. Kong, and O. Strichman, "Compositional Sequentialization of Periodic Programs," in *Proc. of VMCAI*, 2013.

[12] V. D'Silva, D. Kroening, and G. Weissenbacher, "A Survey of Automated Techniques for Formal Software Verification," *IEEE Trans. on Comp. Aided Des.*, vol. 27, no. 7, 2008.

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

[14] V. A. Braberman and M. Felder, "Verification of Real-Time Designs: Combining Scheduling Theory with Automatic Formal Verification," in *Proc. of FSE*, 1999.

[15] F. Laroussinie, N. Markey, and P. Schnoebelen, "Efficient timed model checking for discrete-time systems," *Theoretical Computer Science (TCS)*, vol. 353, no. 1-3, 2006.

[16] R. Alur and D. L. Dill, "A Theory of Timed Automata," *Theo. Comp. Sc.*, vol. 126, no. 2, 1994.

[17] S. Chaki, A. Gurfinkel, and O. Strichman, "Time-Bounded Analysis of Real-Time Systems," in *Proc. of FMCAD*, 2011.

[18] S. L. Torre, P. Madhusudan, and G. Parlato, "Reducing Context-Bounded Concurrent Reachability to Sequential Reachability," in *Proc. of CAV*, 2009.

[19] N. Ghafari, A. J. Hu, and Z. Rakamaric, "Context-Bounded Translations for Concurrent Software: An Empirical Evaluation," in *Proc. of SPIN*, 2010.

[20] M. Emmi, S. Qadeer, and Z. Rakamaric, "Delay-Bounded Scheduling," in *Proc. of POPL*, 2011.

[21] S. Chaki, A. Gurfinkel, and O. Strichman, "Verifying Periodic Programs with Priority Inheritance Locks," in *Proc. of FMCAD*, 2013.

[22] 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.

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

[24] A. Gurfinkel, S. Chaki, and S. Sapra, "Efficient Predicate Abstraction of Program Summaries," in *Proc. of NFM*, 2011.

[25] S. Chaki, A. Gurfinkel, and N. Sinha, "Efficient Verification of Periodic Programs using Sequential Consistency and Snapshots," Extended version of paper published in the proceedings of FMCAD'14, andrew.cmu.edu/~schaki/publications/FMCAD-2014-Extended.pdf.

[26] A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik, "Ufo: A Framework for Abstraction- and Interpolation-Based Software Verification," in *Proc. of CAV*, 2012.

[27] C. Lattner and V. S. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proc. of CGO*, 2004.

[28] V. Ganesh and D. L. Dill, "A Decision Procedure for Bit-Vectors and Arrays," in *Proc. of CAV*, 2007.

[29] E. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs," in *Proc. of TACAS*, 2004.

[30] A. Lal and T. W. Reps, "Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis," in *Proc. of CAV*, 2008.

[2] This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN AS-IS BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT. This material has been approved for public release and unlimited distribution. DM-0000981