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 consisting of periodically activated threads scheduled preemptively, while SS-MOD snapshots only modified variables. We have implemented our approach in a tool. Experiments on a benchmark of robot controllers indicate that our approach significantly outperforms 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 \( P \) and a safety property \( \phi \), the approach is to verify \( P \models \phi \) using two steps: (i) \( \text{VCGEN} \): generate a verification condition (VC), a formula \( VC(P, \phi) \) that is satisfiable iff \( P \not \models \phi \); (ii) \( \text{SAT} \): 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.

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 \( P \) with two tasks \( \tau_1 \) and \( \tau_2 \) – 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 \( P \). 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 to the arrival and WCETs of $\tau_1$ and $\tau_2$, it is impossible for $J_2$ 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 $\tau$, reads the current value of $g$ in $\tau$ 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 $\tau$ 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.


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
Note that we use different fonts for \( J \) to denote different things. In general, \( J \) (or \( J_2 \)) denotes a specific job, \( J \) (or \( J_2 \)) denotes a set of jobs, while \( J(\cdot) \) is a function that maps actions and events to their corresponding jobs.

**Job Alphabet and Program Order:** Each job \( J \) has an alphabet of real actions \( \Sigma_J \subseteq \{ J \} \times \mathbb{Z} \times \{ \tau \} \times \{ r \} \times G \), and write actions \( \Sigma_w(J) \subseteq \{ J \} \times \mathbb{Z} \times \{ w \} \times G \). Let \( \Sigma(J) = \Sigma_r(J) \cup \Sigma_w(J) \cup \{(J, \triangleright) \} \). Let \( PO(J) \) be a partial order over \( \Sigma(J) \), representing control flow. We write \( \alpha \rightarrow_a \alpha' \) to mean \((\alpha, \alpha') \in PO(J)\). Thus, \( \forall \alpha \in \Sigma(J) \cup \Sigma_w(J) \), \((J, \triangleright) \rightarrow \alpha \rightarrow_a \alpha \rightarrow_a \). Let \( J \) be a linearization of \( \Sigma(J) \) consistent with \( PO(J) \), and \( v(\alpha) \) be the index of \( \alpha \) in \( J \). In particular, \( v(J, \triangleright) = 1 \), and \( \alpha(J, \triangleright) = \Sigma(J) \).

**Timed Event Sequences.** The valid executions of periodic programs are characterized by timed event sequences (TES). Formally, a TES is a sequence \( ((e_1, t_1), \ldots, (e_k, t_k)) \) where \( e_i \) is an event, and \( t_i \) is a real-valued timestamp. For TESs \( e_1 \) and \( e_2 \), \( e_1 \sqcup e_2 \) is the set of TESs obtained via their arbitrary interleaving, and \( e_1 \circ e_2 \) is their concatenation. Operations \( \sqcup \) and \( \circ \) extend naturally to sets of TESs. Let \( PriorWr(e, i) \) be the indices of events \( e \) prior to \( e_i \) that write to \( g(e_i) \), i.e., \( PriorWr(e, i) = \{ j \in [1, i) \mid g(e_i) = w \land g(e_j) = g(e_i) \} \). Then, \( LastWr(e, i) \) is the last event written to \( g(e_i) \) prior to \( e_i \), or \( I(g(e_i)) \) if no such write exists, i.e., if \( PriorWr(e, i) = \emptyset \). Then \( LastWr(e, i) = I(g(e_i)) \) if \( e \) is a final event sequence, i.e., \( e \in I(g(e_i)) \) then \( LastWr(e, i) = I(g(e_i)) \).

**Job Semantics.** The semantics of \( J \), denoted \( [J] \), is a set of TESs. Formally, \( \{(e_1, t_1), \ldots, (e_k, t_k)\} \subseteq [J] \) if: (i) \( \forall i \in [1, k] \cdot J(e_i) = J \); (ii) \( A(J) \leq t_1 < t_2 < \cdots < t_k < D(J) \); and (iii) \( \forall i \in [1, k] \cdot (e_i = (\alpha_i, t_i)) \), then the sequence of actions \( (\alpha_1, \ldots, \alpha_k) \) respects the program order \( PO(J) \), i.e., \( \alpha_1 = (J, \triangleright) \), \( \alpha_k = (J, \triangleright) \), and \( \forall i \in [1, k] \cdot (\alpha_i \rightarrow \alpha_{i+1}) \). 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 \(((J_2, \triangleright), \triangleright), ((J_2, 1, r, g), v_1), (J_2, 2), ((J_2, pc, w, g), v_2), (J_2, 3), ((J_2, \triangleright), \triangleright)) \) such that: (i) \( 1 \leq t_1 < t_2 < t_3 < t_4 < 2 \); (ii) \( v_1 < 0 \land \text{pc} = 3 \land v_2 = v_1 + 7 \); (iii) \( v_1 \geq 0 \land \text{pc} = 2 \land v_2 = v_1 \times 5 \).

**Task Semantics.** The semantics of \( \tau_i \), denoted \( [\tau_i] \), is the set of TESs: \( \bigcup_{j=1}^{n}[J_{i,j}] \). Thus, each execution of \( \tau_i \) is a concatenation of an execution from each of its jobs. The

**II. PRELIMINARIES**

We assume an universe bounded by time \( T \). A task \( \pi \) is a 5-tuple \((J, \pi, P, C, A)\) where: (i) \( \pi \) 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) \( \pi > 0 \) is its arrival time. A periodic program \( P \) is a finite sequence of tasks. Consider a PP \( P = (\tau_1, \ldots, \tau_n) \) such that \( \tau_i = (J_i, \pi_i, P_i, C_i, A_i) \). We write \( J_i, j \) to mean the \( j \)-th job of the \( i \)-th task, i.e., \( J_i = (\langle J_i, 1 \rangle, \ldots, \langle J_i, |J_i| \rangle) \). We assume that tasks have: (i) distinct and mutually disjoint jobs, i.e., \( (i, j) \neq (i', j') \implies \tau_i, j \neq \tau_{i'}, j' \); and (ii) distinct priorities \( i \neq i' \implies \pi_i \neq \pi_{i'} \). Let \( RT_i \) be the response time of \( \tau_i \), i.e., the time required by any job of \( \tau_i \) 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 \( \tau_i \) 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 \( J \) be the set of all jobs. We define two relations \( \sqsubseteq \) (finishes-before) and \( \triangleright \) (may preempt) over \( J \) to characterize the order between jobs. Each job \( J_i, j \) has a priority \( \pi(J_i, j) = \pi_i \), arrival time \( A(J_i, j) = A_i + (j - 1) \times P \), and departure time \( D(J_i, j) = A(J_i, j) + RT_i \). Then:

\[
\begin{align*}
J_1 \sqsubseteq J_2 & \iff (\pi(J_1) \leq \pi(J_2) \wedge D(J_1) \leq D(J_2)) \lor \\
& (\pi(J_1) > \pi(J_2) \wedge A(J_1) \leq A(J_2)) \quad (1) \\
J_1 \triangleright J_2 & \iff \pi(J_1) < \pi(J_2) \wedge A(J_1) < A(J_2) < D(J_1) \quad (2)
\end{align*}
\]

Note that \( J_1 \sqsubseteq J_2 \) means that \( J_1 \) always completes before \( J_2 \) starts, and \( J_1 \triangleright J_2 \) 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., \( \forall i \in [1, n] \cdot \forall 1 \leq j < j' \leq |J_i| \cdot J_i, j \sqsubseteq J_i, j' \). Also \( A(J) < D(J) \).

**States and Events of PPs.** We assume a denumerable set \( G \) of real-valued shared variables; \( \mathbb{D} \) contains a distinguished value \( \perp \). Function \( I : G \mapsto \mathbb{D} \) maps shared variables to their initial values. Let \( \mathbb{Z} \) be the set of integers. An action \( \alpha \) is a 4-tuple \((J, pc, \eta, g)\) and an event \( \epsilon \) is a pair \((\alpha, v)\) such that \( J \in J, \ pc \in \mathbb{Z}, \ \eta \in \{ r, w \}, \ g \in G \) and \( v \in \mathbb{D} \). Let \( J(\alpha) = J(\epsilon) = J, \pi(\alpha) = \pi(\epsilon) = \pi(J), \eta(\alpha) = \eta(\epsilon) = \eta, g(\alpha) = g(\epsilon) = g, \) and \( v(\alpha) = v(\epsilon) = v \). 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 \( \epsilon = ((J, \triangleright), \perp)\), \( J(\alpha) = J(\epsilon) = J, \pi(\alpha) = \pi(\epsilon) = \pi(J), \eta(\alpha) = \eta(\epsilon) = \eta = \perp \). Similarly, action \((J, \triangleright)\) and event \((J, \triangleright), \perp \) denote termination of job \( J \). For \( \alpha = (J, \triangleright) \) and \( \epsilon = ((J, \triangleright), \perp)\), \( J(\alpha) = J(\epsilon) = J, \pi(\alpha) = \pi(\epsilon) = \pi(J), \eta(\alpha) = \eta(\epsilon) = \eta = \perp \).
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 $V(J) = \{V(\alpha) \mid \alpha \in \Sigma_r(J) \cup TTES(S)\}$ such that the following holds.

**Fact 1** (Job Verification Condition). For any $\alpha_1, \ldots, \alpha_k \subseteq \Sigma(J)$, and sequence $(v_1, \ldots, v_k) \in \mathbb{D}^k$, the formula $VC(J) \land \bigwedge_{i=1}^k En(\alpha_i) \land V(\alpha_i) = v_i$ is satisfiable iff $\forall \alpha(J) \leq i < k \leq D(J) \cdot (\{\alpha_1, v_1, t_1\}, \ldots, \{\alpha_k, v_k, t_k\}) \in J$.

Thus, every satisfying assignment of $VC(J) \land \bigwedge_{i=1}^k En(\alpha_i) \land V(\alpha_i) = v_i$ 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 $P$ 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, $VC(P) = VC_{seq} \land VC_{clk} \land VC_{obs}$, where $VC_{seq} = \bigwedge_{J \in \mathcal{J}} VC(J)$ and $VC_{clk}$ and $VC_{obs}$ are presented below. In the following, $\Sigma_r$ denotes $\bigcup_{J \in \mathcal{J}} \Sigma_r(J)$, $\Sigma_w$ denotes $\bigcup_{J \in \mathcal{J}} \Sigma_w(J)$, and $\Sigma$ 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 $\alpha$. Following our intuition, we write $\kappa(\alpha)$ to mean $(R(\alpha), \pi(\alpha), i(\alpha))$, i.e., the symbolic timestamp of $\alpha$. 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:

$$a) \bigwedge_{J \in \mathcal{J}} \bigwedge_{\alpha \in \Sigma(J)} (SR(J) \leq R(\alpha) \leq ER(J))$$

$$b) \bigwedge_{J, J' \in \mathcal{J}} ER(J) < ER(J')$$

The observation implies 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 $(R(e, i), \pi(\alpha_i), i(\alpha_i))$. Thus, $(R(e, i), \pi(\alpha_i), i(\alpha_i))$ is a logical representation of the timestamp $t_i$ of event $(\alpha_i, v_i)$. 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.

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

**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 $\alpha_1 \rightarrow \alpha_2, \alpha_3 \rightarrow \alpha_4, \alpha_5 \rightarrow \alpha_6$. Now consider a legal execution of the PP shown in Figure 2(b), where $\forall i \in [1, 6] \cdot x_i = (\{\alpha_1, \alpha_4\}, t_i)$. Let $R(e, i)$, be the number of jobs ending before $x_i$. Let $\in \subseteq \{<, >\}$. Then, we observe for each $(x_i, x_j)$:

1. If $R(e, i) \gg R(e, j)$, then $t_i \gg t_j$. Example pairs are $(x_4, x_2)$ and $(x_2, x_5)$.
2. If $R(e, i) = R(e, j) \land \pi(\alpha_i) \gg \pi(\alpha_j)$, then $t_i \gg t_j$. An example is $(x_1, x_3)$.
3. If $R(e, i) = R(e, j) \land \pi(\alpha_i) = \pi(\alpha_j)$ (note this implies $J(\alpha_i) = J(\alpha_j)$), but $i(\alpha_i) \gg i(\alpha_j)$, then $t_i \gg t_j$. 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 $(R(e, i), \pi(\alpha_i), i(\alpha_i))$. Thus, $(R(e, i), \pi(\alpha_i), i(\alpha_i))$ is a logical representation of the timestamp $t_i$ of event $(\alpha_i, v_i)$. 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.
denotes the symbolic clock of the write action observed by \( \alpha_r \), and is denoted by \( \tilde{\kappa}(\alpha_r) \). Let \( \alpha \prec \alpha' \) denote \( \alpha \) happens before \( \alpha' \), 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 \( \alpha_r \in \Sigma_r \):

\[
En(\alpha_r) \Rightarrow \left( \bigwedge_{\alpha_w \in W(\alpha_r)} \alpha_w \prec \alpha_r \Rightarrow \kappa(\alpha_w) \leq \tilde{\kappa}(\alpha_r) \right) \tag{12}
\]

\[
En(\alpha_r) \Rightarrow \left( VC_{obs}^1 \vee \bigvee_{\alpha_w \in W(\alpha_r)} \left( \bigwedge_{\alpha \in W(\alpha_r)} \alpha \prec \alpha_r \wedge (I(g(\alpha)) = V(\alpha_r)) \right) \right) \tag{13}
\]

\[
VC_{obs}^1(\alpha) = \alpha \prec \alpha_r \wedge \kappa(\alpha) = \tilde{\kappa}(\alpha_r) \wedge V(\alpha) = V(\alpha_r) \tag{14}
\]

\[
VC_{obs}^2(\alpha) = \alpha \prec \alpha_r \wedge \kappa(\alpha) = \tilde{\kappa}(\alpha_r) \wedge V(\alpha) = V(\alpha_r) \tag{15}
\]

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

**Correctness.** The correctness of \( VC(P) \) is expressed by Theorem 1, which states essentially that every satisfying assignment of \( VC(P) \wedge \bigwedge_{i=1}^k (En(\alpha_i) \wedge V(\alpha_i) = v_i) \) corresponds to a legal execution of \( P \). 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 \( (v_1, \ldots, v_k) \in \mathbb{D}^k \), the formula \( VC(P) \wedge \bigwedge_{i=1}^k (En(\alpha_i) \wedge V(\alpha_i) = v_i) \) is satisfiable iff \( \exists 1, \ldots, t \cdot ((\alpha_1, v_1), t_1), \ldots, ((\alpha_k, v_k), t_k) \in [P] \).

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

**IV. HANDLING LOCKS**

In this section, we extend VC generation to handle acquiring and releasing of locks. We consider PP.s 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 \( L \) be the set of all PCP locks. For \( L \subseteq L \subseteq \pi(L) = \{\pi(l) \mid l \in L\} \). Formally, a PTAS action is a 5-tuple \( (J, pc, \pi_t, L_r, L_a) \) such that \( J \in J, pc \in \mathbb{Z}, \pi_t \) is a priority value, \( L_r \subseteq L \), and \( L_a \subseteq L \). A PTAS event \( e \) is a pair \( (\alpha, L_h) \) such that \( \alpha \) is a PTAS action, and \( L_h \subseteq L \). Informally, \( L_h \) denotes the set of locks held after \( \epsilon \) occurs. PTAS actions restrict the set of legal executions of a PP. Specifically, whenever, a PTAS action \( (J, pc, \pi_t, L_r, L_a) \) appears on an execution, the following holds: (i) test: all currently held PCP locks have priority less than \( \pi_t \); 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, \( \Sigma_p(J) = \{sched(J) \cup \bigcup_{L \subseteq L} (lock(J, l) \cup unlock(J, l)) \mid \) where: \( \) sched\( (J) = (J, 0, \pi(J), 0, 0), \) lock\( (J, l) \subseteq \{(J, pc, max(\pi(L))) + 1, \emptyset, \{l\} \mid pc \in \mathbb{Z}\}, \) and \( \) unlock\( (J, l) \subseteq \{(J, pc, max(\pi(L))) + 1, \{l\}, \emptyset \mid pc \in \mathbb{Z}\}. \) 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. Program order \( PO(J) \) satisfies:

\[
\forall \alpha \in \Sigma(J) \setminus \{sched(J), (J, \emptyset)\} \cdot (J, \emptyset) \rightarrow sched(J) \rightarrow \alpha \tag{16}
\]

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

\[
\begin{align*}
L^1_h = L^i_r & \bigwedge_{i \in (1, k]} \forall i \in (1, k] \cdot L^i_h = L^i_h \setminus L^i_r \cup L^i_a \tag{17}
\end{align*}
\]

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 priority has max\((\pi(L) \mid L \in J)\) + 1.

**Updated Construction of \( VC(P) \).** Let \( \Sigma_p = \bigcup_{J \in \Sigma} \Sigma_p(J) \). When constructing \( VC_{seq} \), we treat each \( \alpha \in \Sigma_p \) as a NOP. The construction of \( VC_{clh} \) uses the augmented \( \Sigma(J) \) containing the additional PTAS actions. The construction of \( VC_{obs} \) is updated as follows. For each \( \alpha \in \Sigma(J) \), we add the following terms: \( R(\alpha), R(\alpha), \bar{R}(\alpha), \bar{R}(\alpha), \) 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 \( \alpha \) occurs. Also, we define \( W(\alpha) \), i.e., the set of actions that \( \alpha \) may observe, to contain all other PTAS actions belonging to jobs that do not start after \( J(\alpha) \) finishes. Formally:

\[
W(\alpha) = \{\alpha' \in \Sigma_p \mid \alpha' \not\prec \alpha \wedge -(J(\alpha) \sqsubseteq J(\alpha'))\} \tag{19}
\]

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

\[
En(\alpha) \Rightarrow \left( \bigwedge_{\alpha' \in W(\alpha)} \alpha' \prec \alpha \Rightarrow \kappa(\alpha') \leq \tilde{\kappa}(\alpha) \right) \tag{20}
\]

\[
En(\alpha) \Rightarrow \left( VC_{obs}^3 \vee \bigvee_{\alpha' \in W(\alpha)} \left( \bigwedge_{\alpha \in W(\alpha)} \alpha \not\prec \alpha' \wedge (V(\alpha) = \pi(L_a)) \right) \right) \tag{21}
\]

\[
VC_{obs}^3 = \left( \bigwedge_{\alpha' \in W(\alpha)} \alpha' \not\prec \alpha \wedge (V(\alpha) = \pi(L_a)) \right) \tag{22}
\]

\[
VC_{obs}^4(\alpha') = \alpha' \prec \alpha \wedge \kappa(\alpha') = \tilde{\kappa}(\alpha) \wedge \max(\pi(V(\alpha'))) < \pi_t \wedge V(\alpha) = V(\alpha') \setminus L_r \cup L_a \tag{23}
\]
Note that (20)–(23) assert that the PTAS action observed by \( \alpha \) must be the last PTAS action that executed prior to \( \alpha \); (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 \( VC(P) \). 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 \( t_1 \) with 5 jobs \( J_1, \ldots, J_5 \). Consider initially only the read and write actions \( \alpha_1, \ldots, \alpha_5 \). \( \alpha_1 \) must be the last PTAS action that executed prior to \( \alpha_1 \). Thus, every execution in \( J \) snapshots all variables in \( \text{Snaps}(J) \), and snapshot events appear after all other events, except for \((J, \varnothing)\). Both reads and snapshots observe the last written values. Formally (6) is replaced by:

\[
\forall i \in [1, k] \cdot \eta(\epsilon_i) \in \{r, s\} \implies v(\epsilon_i) = \text{LastWr}(\epsilon_i, i)
\]  (25)

Updated Construction of \( VC(P) \). Let \( \Sigma_s \) be the set of snapshot actions, i.e., \( \Sigma_s = \bigcup_{J \in P} \text{Snaps}(J) \). 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 \( \alpha \in \Sigma_s \) we define \( \Psi(\alpha) \), i.e., the set of actions that \( \alpha \) may observe, as follows. For every job \( J \), and shared variable \( g \), let \( \Psi(J, g) \) be the set of \( g \)-snapshotting jobs less than \( J \) according to the \( \sqsubseteq \) order, i.e.,

\[
\Psi(J, g) = \{J' \in J | g \in \text{Snaps}(J') \land J' \sqsubseteq J \land \forall J'' \in J \cdot g \in \text{Snaps}(J'') \land J'' \sqsubseteq J \implies (J'' \sqsubset J')\}
\]  (26)

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

\[
\Psi_{\perp}(J, g) = \{J' \in J | g \in \text{Snaps}(J') \land J' \sqsubseteq J \land \forall J'' \in J \cdot g \in \text{Snaps}(J'') \land J'' \sqsubseteq J \implies (J'' \sqsubset J')\}
\]  (27)

\[
\Psi_{\perp}(J) = \{J' \in J | J' = J \lor J' \sqsubseteq J\}
\]  (28)

Let \( \alpha = (J, \eta, g) \). Then \( \Psi(\alpha) \) consists of: (i) snapshots by jobs in \( \Psi_{\perp}(J, g) \) and \( \Psi_{\perp}(J, g) \); and (ii) writes by jobs in \( \Psi_{\perp}(J) \). Formally:

\[
\Psi(\alpha) = \{(J', s, g) | J' \in \Psi_{\perp}(J, g) \land \Psi_{\perp}(J, g) \lor \{(J', w, g) | J' \in \Psi_{\perp}(J)\}
\]  (29)

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

We have implemented two variants of snapshotting – \( \text{SS-ALL} \) and \( \text{SS-MOD} \) – which differ in the set of variables snapshotted. For \( \text{SS-ALL} \), all shared variables are snapshotted at the end of each job, i.e., \( \text{Snaps}(J) = G \). For \( \text{SS-MOD} \), only shared variables that are written by a job are snapshotted by it, i.e., \( \text{Snaps}(J) = \{g | (J, w, g) \in \Sigma(J)\} \). We denote by \( \text{SS-NONE} \) 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 \textsc{llrek}, on top of \textsc{uf0} [26] and \textsc{llvm} [27]. The input to \textsc{llrek} is a PP written in \textsc{C}, with jobs implemented via \textsc{C} functions, and periods, priorities etc. specified via macros. The safety property \( \phi \) is expressed as an assertion in the job code. \textsc{llrek} constructs the verification condition \( VC(P, \phi) \), as described earlier, and solves it using \textsc{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.
| Name         | Time (in seconds) | SAT Vars (in 1000s) | SAT Clauses (in 1000s) | AvgOBGS(\*^\^) | | [W (\*^\^)] |
|--------------|------------------|---------------------|------------------------|----------------|----------------|
| nxt.bug1.H1  | 3.7              | 19.4                | 57.2                   | 2.7            | 1.4            |
| nxt.bug1.H2  | 4.1              | 28.4                | 134                    | 2.4            | 1.3            |
| nxt.bug1.H3  | 3.5              | 21.2                | 101                    | 2.3            | 1.1            |
| nxt.bug1.H4  | 3.8              | 24.1                | 110                    | 2.5            | 1.3            |
| nxt.ok1.H1   | 4.2              | 27.3                | 138                    | 2.4            | 1.3            |
| nxt.ok1.H2   | 3.9              | 29.2                | 122                    | 2.5            | 1.3            |
| nxt.ok1.H3   | 3.7              | 26.4                | 111                    | 2.3            | 1.1            |
| nxt.ok1.H4   | 4.1              | 21.1                | 100                    | 2.2            | 1.1            |
| aso.bug1.H1  | 3.8              | 25.9                | 133                    | 2.4            | 1.3            |
| aso.bug1.H2  | 3.7              | 22.2                | 104                    | 2.3            | 1.1            |
| aso.bug1.H3  | 4.1              | 26.3                | 118                    | 2.4            | 1.3            |
| aso.bug1.H4  | 3.6              | 28.1                | 125                    | 2.3            | 1.1            |
| aso.ok1.H1   | 3.9              | 22.3                | 104                    | 2.3            | 1.1            |
| aso.ok1.H2   | 3.8              | 25.9                | 133                    | 2.4            | 1.3            |
| aso.ok1.H3   | 3.7              | 26.4                | 118                    | 2.4            | 1.3            |
| aso.ok1.H4   | 4.1              | 26.3                | 118                    | 2.4            | 1.3            |
| aso.ok2.H1   | 3.7              | 24.7                | 128                    | 2.4            | 1.3            |
| aso.ok2.H2   | 3.8              | 22.2                | 104                    | 2.3            | 1.1            |
| aso.ok2.H3   | 4.1              | 26.3                | 118                    | 2.4            | 1.3            |
| aso.ok2.H4   | 3.6              | 28.1                | 125                    | 2.3            | 1.1            |
| aso.ok3.H1   | 3.9              | 22.3                | 104                    | 2.3            | 1.1            |
| aso.ok3.H2   | 3.8              | 25.9                | 133                    | 2.4            | 1.3            |
| aso.ok3.H3   | 3.7              | 26.4                | 118                    | 2.4            | 1.3            |
| aso.ok3.H4   | 4.1              | 26.3                | 118                    | 2.4            | 1.3            |

**Experimental results:** * = memory out or timeout; Vars = # of SAT variables; Clauses = # of SAT clauses; best numbers are in bold.

**Benchmark.** Our benchmark consists of a set of PPs for controlling two LEGO Mindstorms robots – a two-wheeled self-balancing robot (http://lejos-osak.sourceforge.net/ nxtway_es.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 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 operation 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.* examples, \( k \) indicates that the time-bound \( T \) was set to equal \( k \) hyper-periods (i.e., \( T = k \times 96 \)) of the PP. The next three columns show the verification time \( T \), 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 \( W(P) \) be the set of output (write or snapshot) actions in a PP \( P \). For each \( w \in W(P) \), let \( Obs(w) \) be the set of input (read or snapshot) actions that may observe \( w \). Thus, \( Obs(w) = \{ \alpha \mid w \in W(\alpha) \} \). Let \( AvgOBGS(\*^\^)(P) \) be the mean of the set \( \{ |Obs(w)| \mid w \in W(P)\} \). A smaller value of \( AvgOBGS(\*^\^)(P) \) indicates lower redundancy in the observation sets of \( P \). Here, redundancy means that a single output action may be observed by multiple input actions. Table I shows the values of \( AvgOBGS(\*^\^)(P) \) and \( |W(P)| \) for each \( P \) in our benchmark and for each snapshotting strategy. As expected, \( AvgOBGS(\*^\^)(P) \) 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 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.

**REFERENCES**


