Efficient Verification of Periodic Programs Using Sequential Consistency and Snapshots

Sagar Chaki, Arie Gurfinkel, Nishant Sinha
October 24, 2014

FMCAD’14, Lausanne, Switzerland
Outline

• Context
  • Periodic Programs
  • Time-Bounded Verification

• Verification Condition Generation
  • Hierarchical Lamport Clocks
  • Locks
  • Snapshotting

• Experimental Results

• Related Work
Periodic Embedded Real-Time Software

<table>
<thead>
<tr>
<th>Task</th>
<th>Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>Engine control</td>
<td>10ms</td>
</tr>
<tr>
<td>Airbag</td>
<td>40ms</td>
</tr>
<tr>
<td>Braking</td>
<td>40ms</td>
</tr>
<tr>
<td>Cruise Control</td>
<td>50ms</td>
</tr>
<tr>
<td>Collision Detection</td>
<td>50ms</td>
</tr>
<tr>
<td>Entertainment</td>
<td>80ms</td>
</tr>
</tbody>
</table>

**Technical Name**
Periodic Fixed-Priority Software with Preemptive Rate Monotonic Scheduling

*Domains: Avionics, Automotive*

*OS: OSEK, VxWorks, RTEMS*

We call them periodic programs
Time-Bounded Verification [FMCAD’11&’14, VMCAI’13]

Input: Periodic Program
- Collection of periodic tasks
  - Execute concurrently with preemptive priority-based scheduling
  - Priorities respect RMS
  - Communicate through shared memory

Problem: Time-Bounded Verification
- Assertion $A$ violated within $X$ ms of a system’s execution from initial state $I$?
  - $A$, $X$, $I$ are user specified
  - Time bounds map naturally to program’s functionality (e.g., air bags)

Solution: Bounded Model Checking
- Generate Verification Condition (SMT Formula over Bit-Vectors)
- Use SMT Solver to check satisfiability
Periodic Program (PP)

An N-task periodic program PP is a set of tasks \( \{\tau_1, \ldots, \tau_N\} \)

A task \( \tau \) is a tuple \( \langle I, T, P, C, A \rangle \), where

- \( I \) is a task identifier = its priority
- \( T \) is a task body (i.e., code)
- \( P \) is a period
- \( C \) is the worst-case execution time
- \( A \) is the release time: the time at which task becomes first enabled

Semantics of PP bounded by time \( X \equiv \) asynchronous concurrent program:
Efficient Verification of Periodic Programs
Sagar Chaki, October 24, 2014

© 2014 Carnegie Mellon University

Periodic Program Example

High-Priority Task

Low-Priority Task

Job1 of \(\tau_2\)

\(\tau_1\) executes for 2 units

\(\tau_2\)

\(\tau_1\) preempts \(\tau_2\)

\(\tau_1 = \langle 1, J_1, 8, 2, 0 \rangle, \quad \tau_2 = \langle 2, J_2 = J_3, 4, 1, 1 \rangle\)

Job2 of \(\tau_2\)

Another Legal Execution – \(\tau_1\) executes for 1 units

Illegal Execution – \(\tau_1\) preempts \(\tau_2\)
Verification Condition

\[ VC = VC_{seq} \land VC_{clk} \land VC_{obs} \]

- Encodes purely job-local computation. Value read/written by each shared variable access represented by a fresh variable.
- Associates each shared variable access with a hierarchical Lamport Clock. Constraints values of Clock components based on timing and priority.
- Connects value read at each “read” to the value written by most recent “write” according to the Lamport Clock.
Verification Condition $V C_{seq}$

\[ V \]
\[ C \]
\[ s e q \]

Same as verification condition for sequential program except that both reads and writes are given fresh variables

\[ J_1() \{ x := x + 1; \} \rightarrow x_2 = x_1 + 1 \]
\[ J_2() \{ x := x + 1; \} \rightarrow x_4 = x_3 + 1 \]
\[ J_3() \{ x := x + 1; \} \rightarrow x_6 = x_5 + 1 \]
Verification Condition $VC_{clk}$

Observe: $x_i$ is accessed before $x_j$ iff $(R_i, \pi_i, \iota_i) < (R_j, \pi_j, \iota_j)$
where $<$ is lexicographic ordering

Claim/Intuition: This holds for all legal executions, not just this one.

Therefore: Associate $x_i$ with hierarchical Lamport clock $\kappa_i = (R_i, \pi_i, \iota_i)$

- $\pi_i = priority\ of\ job\ accessing\ x_i$
  - $\pi_1 = \pi_2 = 1, \pi_3 = \cdots = \pi_6 = 2$
- $R_i = \# of\ jobs\ finished\ before\ x_i\ accessed$
  - $R_1 = R_3 = R_4 = 0, R_2 = 1, R_5 = R_6 = 2$
- $\iota_i = index\ of\ instruction\ accessing\ x_i\ in\ topological\ ordering\ of\ CFG$
  - $\iota_1 = \iota_3 = \iota_5 = 1, \iota_2 = \iota_4 = \iota_6 = 2$
Verification Condition $VC_{obs}$

Let $J_i$ = job in which $x_i$ is accessed

Compute: $J \sqsubseteq J'$ if $J$ always completes before $J'$ starts

Recall $\kappa_i = (R_i, \pi_i, \iota_i)$. For each read $x_i$, let

$W_i = \{x_j | x_j \text{ is a write } \land \neg (J_i \sqsubseteq J_j)\}$, i.e., the set of all writes that

$x_i$ “may observe”

$VC_{obs} \equiv$

The value of each $x_i$ accessed by a read equals the value of $x_j$

such that $\kappa_j = max\{\kappa_k | \kappa_k < \kappa_i \text{ and } x_k \in W_i\}$, where $max\{} =$

initial value of $x$. 
Verification Condition $V C_{obs}$

For each read $x_i$ introduce $\tilde{\kappa}_i = \text{clock of write action observed}$

$$V C_{obs} \equiv$$

$$\land_{x_j \in W_i} \kappa_j < \kappa_i \Rightarrow \kappa_j \leq \tilde{\kappa}_i$$

$$\land$$

$$((V C^1_{obs}) \lor (\lor_{x_j \in W_i} V C^2_{obs}(j)))$$

$$V C^1_{obs} \equiv (\land_{x_j \in W_i} \kappa_j \geq \kappa_i) \land (x_i = x_{\text{Init}})$$

$$V C^2_{obs}(j) \equiv (\kappa_j < \kappa_i \land \kappa_j = \tilde{\kappa}_i) \land x_i = x_j$$

$x_i$ observes initial value $x_{\text{Init}}$ of $x$

$x_i$ observes $x_j$

In the paper, we handle multiple shared variables.
Snapshotting: Problem

\[ w_i = \text{write}, r_i = \text{read} \]

Sequence of jobs. Each job writes to a variable multiple times.

\[
\begin{align*}
\tau_1 & \quad J_1 & \quad J_2 & \quad J_3 & \quad J_4 & \quad J_5 \\
0 & w_1 & w_3 & w_5 & w_7 & w_9 \\
1 & r_1 & r_2 & r_3 & r_4 & r_5 \\
2 & w_2 & w_4 & w_6 & w_8 & w_{10} \\
\end{align*}
\]

Series-Parallel Structure

Observe: \( W(r_1) = \{w_1, w_2\} \), \( W(r_2) = \{w_1, w_2, w_3, w_4\} \), \( W(r_3) = \{w_1, w_2, w_3, w_4, w_5, w_6\}, \ldots \)

Result: Problem for \( r_{<i} \) gets re-encoded (and resolved) as part of problem for \( r_i \)

Empirically: SMT solvers do not scale beyond small number of jobs
**Snapshotting: Solution**

Now: \( W(r_1) = W(s_1) = \{w_1, w_2\}, W(r_2) = W(s_2) = \{s_1, w_3, w_4\}, W(r_3) = W(s_3) = \{s_2, w_5, w_6\}, \ldots \)

Result: Solving \( VC_{obs} \) involves fewer redundant computation

Empirically: SMT solvers scale beyond small number of jobs

Choice of variables to snapshot: (i) all variables (ii) only written by the job

\( w_i = \text{write}, r_i = \text{read} \)

\( s_i = \text{snapshot} \)
Verification Condition $VC_{obs}$ with Snapshotting

Input: $Snaps(J) =$ set of variables snapshotted by $J$

Compute: Relation $J \uparrow J'$ iff $J$ can be preempted by $J'$

Let $\Psi_{\subset}(J, g) =$ maximal jobs less that $J$ that snapshot $g$

Let $\Psi_{\uparrow}(J, g) = \{J' | J \uparrow J' \land g \in Snaps(J')\}$

Let $\Psi_{\downarrow}(J) = \{J' | J' = J \lor J' \uparrow J\}$

These relations capture the series-parallel structure

$$W_i = \{x_j | x_j \text{ is a snapshot } \land J_j \in \Psi_{\uparrow}(J_i, g)\} \cup \{x_j | x_j \text{ is a snapshot } \land J_j \in \Psi_{\subset}(J_i, g)\} \cup \{x_j | x_j \text{ is a write } \land J_j \in \Psi_{\downarrow}(J_i, g)\}$$

$VC_{obs} \equiv$ same as before with the new definition of $W_i$ above
Handling Locks

We handle two types of locks (both involve changing priorities)

- Each thread has a base priority = priority of task it executes
- Each PCP lock $l$ is associated with priority $\pi(l)$
  - A CPU lock is a PCP lock such that $\pi(l) = \infty$
  - Thread’s priority = max (its base priority, priorities of all PCP locks it holds)

Lock operation encoded by “priority-test-and-set” action $(j, pc, \pi_t, L_r, L_a)$

- Guard: All held locks must have priority less than $\pi_t$
- Command: Locks in $L_r$ are released; Locks in $L_a$ are acquired
- Encode by updating $VC_{clk}$ and $VC_{obs}$ appropriately

Note: To handle locks, we generalize VC-Gen to support operations that read and write program state (in this case held locks) atomically

- Atomic operations handled similarly to snapshots
### Results (Time in seconds)

<table>
<thead>
<tr>
<th></th>
<th>NONE</th>
<th>ALL</th>
<th>MOD</th>
<th>REKH</th>
</tr>
</thead>
<tbody>
<tr>
<td>nxt.bug1:H1</td>
<td>33</td>
<td>9</td>
<td>7</td>
<td>18</td>
</tr>
<tr>
<td>nxt.bug2:H1</td>
<td>32</td>
<td>10</td>
<td>7</td>
<td>31</td>
</tr>
<tr>
<td>nxt.ok1:H1</td>
<td>19</td>
<td>7</td>
<td>8</td>
<td>17</td>
</tr>
<tr>
<td>nxt.ok2:H1</td>
<td>20</td>
<td>7</td>
<td>6</td>
<td>29</td>
</tr>
<tr>
<td>nxt.ok3:H1</td>
<td>30</td>
<td>8</td>
<td>6</td>
<td>31</td>
</tr>
<tr>
<td>aso.bug1:H1</td>
<td>29</td>
<td>9</td>
<td>9</td>
<td>34</td>
</tr>
<tr>
<td>aso.bug2:H1</td>
<td>28</td>
<td>10</td>
<td>9</td>
<td>32</td>
</tr>
<tr>
<td>aso.bug3:H1</td>
<td>29</td>
<td>13</td>
<td>11</td>
<td>80</td>
</tr>
<tr>
<td>aso.bug4:H1</td>
<td>32</td>
<td>17</td>
<td>9</td>
<td>66</td>
</tr>
<tr>
<td>aso.ok1:H1</td>
<td>32</td>
<td>11</td>
<td>10</td>
<td>32</td>
</tr>
<tr>
<td>aso.ok2:H1</td>
<td>38</td>
<td>29</td>
<td>17</td>
<td>67</td>
</tr>
<tr>
<td>nxt.bug1:H4</td>
<td>*</td>
<td>119</td>
<td>74</td>
<td>*</td>
</tr>
<tr>
<td>nxt.bug2:H4</td>
<td>*</td>
<td>172</td>
<td>92</td>
<td>*</td>
</tr>
<tr>
<td>nxt.ok1:H4</td>
<td>*</td>
<td>89</td>
<td>49</td>
<td>*</td>
</tr>
</tbody>
</table>

NONE=No snapshotting, ALL=Snapshot all variables, MOD=Snapshot only modified variables, REKH=Previous tool based on sequentialization

2GB Memory Limit
60min Time Limit
Solver=STP
Results (Time in seconds)

<table>
<thead>
<tr>
<th></th>
<th>NONE</th>
<th>ALL</th>
<th>MOD</th>
<th>REKH</th>
</tr>
</thead>
<tbody>
<tr>
<td>ntx.ok2:H4</td>
<td>*</td>
<td>125</td>
<td>49</td>
<td>*</td>
</tr>
<tr>
<td>ntx.ok3:H4</td>
<td>*</td>
<td>358</td>
<td>133</td>
<td>*</td>
</tr>
<tr>
<td>aso.bug1:H4</td>
<td>*</td>
<td>128</td>
<td>92</td>
<td>*</td>
</tr>
<tr>
<td>aso.bug2:H4</td>
<td>*</td>
<td>147</td>
<td>74</td>
<td>*</td>
</tr>
<tr>
<td>aso.bug3:H4</td>
<td>*</td>
<td>209</td>
<td>136</td>
<td>*</td>
</tr>
<tr>
<td>aso.bug4:H4</td>
<td>*</td>
<td>329</td>
<td>152</td>
<td>*</td>
</tr>
<tr>
<td>aso.ok1:H4</td>
<td>*</td>
<td>270</td>
<td>210</td>
<td>*</td>
</tr>
<tr>
<td>aso.ok2:H4</td>
<td>*</td>
<td>1312</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>ctm.bug2</td>
<td>36</td>
<td>29</td>
<td>21</td>
<td>105</td>
</tr>
<tr>
<td>ctm.bug3</td>
<td>*</td>
<td>124</td>
<td>59</td>
<td>258</td>
</tr>
<tr>
<td>ctm.ok1</td>
<td>23</td>
<td>37</td>
<td>21</td>
<td>122</td>
</tr>
<tr>
<td>ctm.ok2</td>
<td>28</td>
<td>26</td>
<td>17</td>
<td>111</td>
</tr>
<tr>
<td>ctm.ok3</td>
<td>*</td>
<td>116</td>
<td>53</td>
<td>275</td>
</tr>
<tr>
<td>ctm.ok4</td>
<td>*</td>
<td>320</td>
<td>143</td>
<td>395</td>
</tr>
</tbody>
</table>

NONE=No snapshotting, ALL=Snapshot all variables, MOD=Snapshot only modified variables, REKH=Previous tool based on sequentialization

2GB Memory Limit
60min Time Limit
Solver=STP
## Observability Sizes

<table>
<thead>
<tr>
<th></th>
<th>$\text{AVGOBS}(\mathcal{P})$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NONE</td>
</tr>
<tr>
<td>nxt.bug1:H1</td>
<td></td>
</tr>
<tr>
<td>nxt.bug2:H1</td>
<td>25.6</td>
</tr>
<tr>
<td>nxt.ok1:H1</td>
<td>26.5</td>
</tr>
<tr>
<td>nxt.ok2:H1</td>
<td>25.6</td>
</tr>
<tr>
<td>nxt.ok3:H1</td>
<td>25.4</td>
</tr>
<tr>
<td>aso.bug1:H1</td>
<td>26.5</td>
</tr>
<tr>
<td>aso.bug2:H1</td>
<td>26.0</td>
</tr>
<tr>
<td>aso.bug3:H1</td>
<td>26.4</td>
</tr>
<tr>
<td>aso.bug4:H1</td>
<td>25.5</td>
</tr>
<tr>
<td>aso.ok1:H1</td>
<td>26.5</td>
</tr>
<tr>
<td>aso.ok2:H1</td>
<td>27.1</td>
</tr>
<tr>
<td>nxt.bug1:H4</td>
<td>99.5</td>
</tr>
<tr>
<td>nxt.bug2:H4</td>
<td>102.9</td>
</tr>
<tr>
<td>nxt.ok1:H4</td>
<td>99.5</td>
</tr>
</tbody>
</table>

$\text{AVGOBS}(\mathcal{P}) = \text{avg. no. of reads observing each write or snapshot}$

$|W(\mathcal{P})| = \text{total no. of snapshot and write variables}$
## Observability Sizes

|          | \( \text{AVGOBS}(\mathcal{P}) \) | \( |W(\mathcal{P})| \) |
|----------|----------------------------------|-------------------------|
|          | NONE | ALL | MOD | NONE | ALL | MOD |
| ntx.ok2:H4 | 99.3 | 3.0 | 3.0 | 1192 | 1834 | 1675 |
| ntx.ok3:H4 | 102.9 | 3.1 | 3.2 | 1240 | 1989 | 1731 |
| aso.bug1:H4 | 99.9 | 3.6 | 3.6 | 1216 | 2072 | 1723 |
| aso.bug2:H4 | 101.6 | 3.7 | 3.7 | 1232 | 2088 | 1739 |
| aso.bug3:H4 | 98.3 | 3.6 | 3.5 | 1420 | 2490 | 2034 |
| aso.bug4:H4 | 100.4 | 4.6 | 4.4 | 1236 | 2199 | 1751 |
| aso.ok1:H4 | 103.2 | 4.1 | 4.2 | 1244 | 2100 | 1751 |
| aso.ok2:H4 | 100.1 | 4.6 | 4.4 | 1244 | 2207 | 1759 |
| ctm.bug2 | 17.9 | 4.1 | 4.5 | 512 | 1052 | 683 |
| ctm.bug3 | 26.6 | 4.1 | 4.5 | 768 | 1588 | 1033 |
| ctm.ok1 | 18.6 | 4.1 | 4.6 | 512 | 1052 | 684 |
| ctm.ok2 | 18.1 | 4.1 | 4.5 | 512 | 1052 | 683 |
| ctm.ok3 | 27.9 | 4.1 | 4.5 | 780 | 1600 | 1057 |
| ctm.ok4 | 36.4 | 4.2 | 4.7 | 1040 | 2140 | 1400 |

\[ \text{AVGOBS}(\mathcal{P}) = \text{avg. no. of reads observing each write or snapshot} \]

\[ |W(\mathcal{P})| = \text{total no. of snapshot and write variables} \]
Related Work and Concluding Thoughts

Generate Verification Condition by Encoding Dataflow between Reads and Writes Using Lamport Clocks

- Nishant Sinha, Chao Wang: Staged concurrent program analysis. SIGSOFT FSE 2010: 47-56
- Jade Alglave, Daniel Kroening, Michael Tautschnig: Partial Orders for Efficient Bounded Model Checking of Concurrent Software. CAV 2013: 141-157

Generate Verification Condition per Scheduling round using prophecy variables, and ensure that output of one round equals input to the next


- Snapshotting combines both ideas
- Interplay between Logical Clocks and Prophecy Variables
  - Both due to Lamport
- We encode both program variables and clocks as bit-vectors
  - Clocks can be encoded as integers, but then we have a mixed theory
QUESTIONS?
Contact Information Slide Format

Sagar Chaki
Principal Researcher
SSD/CSC
Telephone: +1 412-268-1436
Email: chaki@sei.cmu.edu

Web
www.sei.cmu.edu
www.sei.cmu.edu/contact.cfm

U.S. Mail
Software Engineering Institute
Customer Relations
4500 Fifth Avenue
Pittsburgh, PA 15213-2612
USA

Customer Relations
Email: info@sei.cmu.edu
Telephone: +1 412-268-5800
SEI Phone: +1 412-268-5800
SEI Fax: +1 412-268-6257