Time-Bounded Analysis of Real-Time Systems

Sagar Chaki\textsuperscript{1}, Arie Gurfinkel\textsuperscript{1},
Ofer Strichman\textsuperscript{2}, Soonho Kong\textsuperscript{1}

\textsuperscript{1}Software Engineering Institute, CMU
\textsuperscript{2}Technion, Israel Institute of Technology
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.

Use of any trademarks in this presentation is not intended in any way to infringe on the rights of the trademark holder.

This Presentation may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.

This work was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.
Software Engineering Institute (SEI)

Department of Defense R&D Laboratory (FFRDC)
Created in 1984
Under contract to Carnegie Mellon University
Offices in Pittsburgh, PA; Washington, DC; and Frankfurt, Germany

**SEI Mission:** advance software engineering and related disciplines to ensure the development and operation of systems with predictable and improved cost, schedule, and quality.
Motivation: Real-Time Embedded Systems

Avionics Mission System*
Rate Monotonic Scheduling (RMS)

<table>
<thead>
<tr>
<th>Task</th>
<th>Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>weapon release</td>
<td>10ms</td>
</tr>
<tr>
<td>radar tracking</td>
<td>40ms</td>
</tr>
<tr>
<td>target tracking</td>
<td>40ms</td>
</tr>
<tr>
<td>aircraft flight data</td>
<td>50ms</td>
</tr>
<tr>
<td>display</td>
<td>50ms</td>
</tr>
<tr>
<td>steering</td>
<td>80ms</td>
</tr>
</tbody>
</table>

Case Study: A Metal Stamping Robot

a.k.a. LEGO Turing Machine

Image courtesy of Taras Kowaliw
LEGO Turing Machine

BEGIN:
  READ
  CJUMP0 CASE_0
CASE_1:
  WRITE 0
  MOVE R
  JUMP BEGIN
CASE_0:
  WRITE 1
  MOVE R
  JUMP BEGIN

Turing Machine (Video)

http://www.youtube.com/watch?v=teDyd0d5M4o
Turing Machine: Task Structure

**Controller Task**
- Priority: 1 (Lowest)
- Period: 500 ms | WCET: 440 ms
  1. Calibrate Sensor
  2. Read Program via USB
  3. Command other tasks

**Writer Task**
- Priority: 3
- Period: 50 ms | WCET: < 1 ms
- Flip bits

**Background Task**
- Priority: 5 (Highest)
- Period: 1 ms | WCET: < 1 ms
  1. Call USB background-process
  2. Send log-buffer via USB

**Reader Task**
- Priority: 4
- Period: 25 ms | WCET: < 1 ms
- Read bits using NXT-colorsensor

**TapeMover Task**
- Priority: 2
- Period: 100 ms | WCET: < 1 ms
- Move the tape (left or right)
Turing Machine: Properties

Tape does not move when a bit is read or written

Read sensor and Write arm can move concurrently but must not interfere with one another

Read sensor’s light is off when not in use

Read task WCET is less than 25ms
  • reduced to checking API usage rules

No log messages are lost during USB communication
  • each message is delivered to the server before a new one is produced
Time-Bounded Verification of Periodic Programs

Time-Bounded Verification
- Is an assertion \( A \) violated within \( X \) milliseconds of a system’s execution from initial state \( I \)
  - \( A, X, I \) are user specified

Periodic Program
- Collection of periodic tasks
  - Execute concurrently with fixed-priority scheduling
  - Priorities respect RMS
  - Communicate through shared memory
  - Synchronize through preemption and priority ceiling locks

Assumptions
- System is schedulable
- WCET of each task is given
Overall Approach

Supports C programs w/ tasks, priorities, priority ceiling protocol, shared variables

Works in two stages:

1. **Sequentialization** – reduction to sequential program w/ *prophecy* variables
2. **Bounded program analysis**: bounded C model checker (CBMC, HAVOC, …)

Periodic Program in C

Sequentialization

Sequential Program

Periods, WCETs, Initial Condition, Time bound

OK

BUG + CEX

Analysis
Periodic Program

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
- \( T \) 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 is given by an asynchronous concurrent program:

\[
\begin{align*}
  k_i &= 0; \\
  \text{while} \ (\text{Wait}(\tau_i, k_i)) \\
  &\quad T_i(); \\
  &\quad k_i = k_i + 1;
\end{align*}
\]
Periodic Programs

Task $\tau = (I, T, P, C, A)$

Priority
Periodic Programs

Task $\tau = (I, T, P, C, A)$
Periodic Programs

Task $\tau = (I, T, P, C, A)$
Periodic Programs

Task $\tau = (I, T, P, C, A)$
Periodic Programs

Task $\tau = (I, T, P, C, A)$
Preemptive Fixed Priority Scheduling
Preemptive Fixed Priority Scheduling

High Priority Task

Low Priority Task

Time
Preemptive Fixed Priority Scheduling

High Priority Task

Low Priority Task

High & low priority jobs arrived together
Preemptive Fixed Priority Scheduling

High priority job is executed first

Time-Bounded Analysis of Real-Time Sys.  
Chaki, Gurfinkel, Strichman  
© 2012 Carnegie Mellon University
Preemptive Fixed Priority Scheduling

High Priority Task

Low Priority Task

Low priority job is executed later
Preemptive Fixed Priority Scheduling

High Priority Task

Low Priority Task

High priority job arrived

Time
Preemptive Fixed Priority Scheduling

High Priority Task

Low Priority Task

Preempts low priority job
Preemptive Fixed Priority Scheduling

High Priority Task

Low Priority Task

Time

Lower priority job resumes later
Preemptive Fixed Priority Scheduling

High Priority Task

Low Priority Task

Time
Preemptive Fixed Priority Scheduling

High Priority Task

Low Priority Task

Time
Preemptive Fixed Priority Scheduling
Example: Task Schedule

<table>
<thead>
<tr>
<th>Task</th>
<th>WCET ($C_i$)</th>
<th>Period ($P_i$)</th>
<th>Arrival Time ($A_i$)</th>
<th>Response Time ($RT_i$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_2$</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$\tau_1$</td>
<td>2</td>
<td>8</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>$\tau_0$</td>
<td>8</td>
<td>16</td>
<td>0</td>
<td>16</td>
</tr>
</tbody>
</table>

Maximum difference between arrival time and completion time of a job

Computed via Rate Monotonic Analysis
Time Bounded Semantics of Periodic Program

Assumptions

• Time window $W$ is divisible by the period of each task (i.e., $W | P_i$)
• Each task arrives in time to complete in 1st period (i.e., $A_i + RT_i \leq P_i$)

The time bound imposes a natural bound on # of jobs: $J_i = W / P_i$

Time-Bounded Semantics of PP is

```plaintext
k_i = 0;
while (k_i < J_i && Wait(\tau_i, k_i))
  T_i();
  k_i = k_i + 1;
```

Job-Bounded Abstraction

• Abstracts away time
• Approximates $\text{Wait}()$ by a non-deterministic delay
• Preserves logical (time-independent) properties!
C as a Modeling Language

Extend C programming language with 3 modeling features

Assertions

- `assert(e)` – aborts an execution when `e` is false, no-op otherwise

```c
void assert (_Bool b) { if (!b) exit(); }
```

Non-determinism

- `nondet_int()` – returns a non-deterministic integer value

```c
int nondet_int () { int x; return x; }
```

Assumptions

- `assume(e)` – “ignores” execution when `e` is false, no-op otherwise

```c
void assume (_Bool e) { while (!e); }
```
Example of Using Assume/Nondet/Assert

```c
int x, y;

void main (void) {
  x = nondet_int ();
  assume (x > 10);
  y = x + 1;
  assert (y > x);
}
```
Example: Modeling with Prophecy Variables

```c
int x, y, v;
void main (void) {
    v = nondet_int ();
    x = v;
    x = x + 1;
    y = nondet_int ();
    assume (v == y);
}
```

- `v` is a *prophecy* variable. It guesses the future value of `y`.
- Syntactically: `x` is changed *before* `y`.
- Semantically: `x` *depends on* `y`.
- Assume blocks executions with a wrong guess.
Partition Execution into Rounds

Execution starts in round 0
A round ends, and a new one begins, each time a job finishes
• # rounds == # of jobs
Sequentialization: Visually

Guess initial value of each global in each round (g[0] … g[6])
Remember initial values in prophecy variables
Execute task bodies

• τ₀
• τ₁
• τ₂

Check that final value of round i is the initial value of round i +1 (using the remembered prophecy values)
Sequentialization: Visually

- $g[0]$: $\tau_2$
- $g[1]$: $\tau_1$
- $g[2]$: $\tau_0$
- $g[3]$: $\tau_0$
- $g[4]$: $\tau_1$
- $g[5]$: $\tau_0$
- $g[6]$: $\tau_0$

Rounds: 0, 1, 2, 3, 4, 5, 6

Time Bounded Analysis of Real-Time Sys. Chaki, Gurfinkel, Strichman

© 2012 Carnegie Mellon University
Sequentialization: Overview

Sequential Program for execution of $R$ rounds:

1. for each global variable $g$
   - let $i_g[i]$ be the prophesied initial value of $g$ in round $i$
     - Initialize $i_g[i]$ with a non-deterministic value
   - let $g[i]$ be the value of $g$ in round $i$
     - Initialize $g[i]$ to be equal to $i_g[i]$

2. non-deterministically choose for each task $t$ and job $j$
   - start round: $\text{start}[t][j]$
   - end round: $\text{end}[t][j]$

3. execute task bodies sequentially
   - in ascending order of priorities
   - for global variables, use $g[i]$ instead of $g$ when running in round $i$
   - non-deterministically decide where to context switch
   - at a context switch jump to a new round (cannot preempt a higher task)

4. check that initial value of round $i+1$ is the final value of round $i$

5. check user assertions

must be well-nested
Sequentialization

```plaintext
var
  int round;       // current round
  int job;         // current job
  int endRound;    // end round of the current job
  int g[R], i_g[R]; // global and initial global
  int start[N][J], end[N][J]; // start/end round of every job
  Bool localAssert[N][J] = {1..1}; // local assertions

void main()
  initShared();

initShared ()
  for each global g: g[0] = init_value (g);
```

User-supplied initial value of g
Sequentialization

var
int round;  // current round
int job;    // current job
int endRound;  // end round of the current job
int g[R], i_g[R];  // global and initial global
int start[N][J], end[N][J];  // start/end round of every job
Bool localAssert[N][J] = {1..1};  // local assertions

void main ()
initShared();
initGlobals();

initGlobals ()
  for r in [1,R):  // for each round
    for each global g: g[r] = i_g[r] = nondet();
## Sequentialization

```plaintext
var
    int round; // current round
    int job; // current job
    int endRound; // end round of the current job
    int g[R], i_g[R]; // global and initial global
    int start[N][J], end[N][J]; // start/end round of every job
    Bool localAssert[N][J] = {1..1}; // local assertions
```

```plaintext
void main()
    initShared();
    initGlobals();
    scheduleJobs();
```

Will look at this in more details later, but it will essentially assign appropriate values to `start[x][y]` and `end[x][y]`
Sequentialization

```c
var
int round; // current round
int job; // current job
int endRound; // end round of the current job
int g[R], i_g[R]; // global and initial global
int start[N][J], end[N][J]; // start/end round of every job
Bool localAssert[N][J] = {1..1}; // local assertions

void main ()
initShared();
initGlobals();
scheduleJobs();

for t in [0,N) : // for each task
  for j in [0,J_t) : // for each job
    job = j;
    round = start[t][job];
    endRound = end[t][job];
    T_t();
    assume (round == endRound);
```
Sequentialization

void $T'_t()$
  Same as $T_t$, but each statement ‘st’ is replaced with:
  contextSwitch (t); st[g ← g[round]];
  and each ‘assert(e)’ is replaced with:
  localAssert[t][job] = e;

void contextSwitch (task t)
  int oldRound;

  if (nondet ()) return; // non-det do not context switch

  oldRound = round;
  round = nondet_int ();
  assume (oldRound < round <= endRound);

  // for each higher priority job, ensure that t does not preempt it
  for t1 in [t+1, N):
    for j1 in [0,$J_{t1}$):
      assume(round <= start[t1][j1] || round > end[t1][j1]);
void main ()
    scheduleJobs();
    initShared();
    initGlobals();
    for t in [0,N) : // for each thread
        for j in [0,J] : // for each job
            job = j;
            round = start[t][job];
            endRound = end[t][job];
            T'();
            assume (round == endRound);
    checkAssumptions ();

bool localAssert[N][J] = {1..1}; // local assertions

for r in [0,R-1):
    for each global g:
        assume (g[r] == i_g[r+1]);
Sequentialization

```c
var
int round; // current round
int job; // current job
int endRound; // end round of the current job
int g[R], i_g[R]; // global and initial global
int start[N][J], end[N][J]; // start/end round of every job
Bool localAssert[N][J] = {1..1}; // local assertions

void main ()
scheduleJobs();
initShared();
initGlobals();

for t in [0,N) : // for each thread
  for j in [0,J_t) : // for each job
    job = j;
    round = start[t][job];
    endRound = end[t][job];
    T_t();
    assume (round == endRound);

checkAssumptions ();
checkAssertions ();
```

```c
checkAssumptions ()
  for r in [0,R-1):
    for each global g:
      assume (g[r] == i_g[r+1]);

checkAssertions ()
  for t in [0,N-1):
    for j in [0,J_t):
      assert (localAssert[t][j]);
```
Sequentialization

```c
var
  int round;        // current round
  int job;          // current job
  int endRound;     // end round of the current job
  int g[R], i_g[R]; // global and initial global
  int start[N][J], end[N][J]; // start/end round of every job
  Bool localAssert[N][J] = {1...1}; // local assertions

void main ()
  scheduleJobs();
  initShared();
  initGlobals();

  for t in [0,N) : // for each task
    for j in [0,Jt) : // for each job
      job = j;
      round = start[t][job];
      endRound = end[t][job];
      T'_t();
      assume (round == endRound);

  checkAssumptions ();
  checkAssertions ();
```

Full Sequentialization
Sequentialization: Job Scheduling

```c
void scheduleJobs ()
for t in [0,N):
    for j in [0, J_t):
        start[t][j] = nondet_int ();
        end[t][j] = nondet_int ();
        assume (0 <= start[t][j]);  // start in a legal round
        assume (end[t][j] <= R);     // end in a legal round
        assume (start[t][j] <= end[t][j]); // start before end
        assume (end[t][j] < start[t][j+1]); // jobs are run in order

    // jobs are well-nested (low priority job does not preempt a high priority job)
for t1 in [0,N-1):
    for t2 in [t1 + 1,N):
        for j1 in [0, J_{t1}):
            for j2 in [0, J_{t2}):
                if (start[t1][j1] <= end[t2][j2] && start[t2][j2] <= end[t1][j1])
                    assume (start[t1][j1] <= start[t2][j2] <= end[t2][j2] <= end[t1][j1])
```
Additional Parts

Partial Order Reduction
• allow for context switches ONLY at statements that access shared variables
• ensure that read statements are preempted by write statements…

Preemption bounds
• we use RMA to compute an upper bound on the number of times one task can preempt another
• scheduleJobs() enforces this bound with additional constraints

Locks
• preemption locks
  – do not allow context switch when a task holds a lock
• priority ceiling locks
  – extend the model with dynamic priorities (see details in the paper)

Assertions
• jump to the end of the execution as soon as a local assertion is violated
Implementation: REK

Sequentialization (CIL)

Periodic Program in C

OK

BUG + CEX

Analysis (CBMC)

Sequential Program

Periods, WCETs, Initial Condition, Time bound

http://www.andrew.cmu.edu/~arieg/Rek
NXTway-GS: a 2 wheeled self-balancing robot

Original: nxt (2 tasks)
• balancer (4ms)
  – Keeps the robot upright and responds to BT commands
• obstacle (50ms)
  – monitors sonar sensor for obstacle and communicates with balancer to back up the robot

Ours: aso (3 tasks)
• balancer as above but no BT
• obstacle as above
• bluetooth (100ms)
  – responds to BT commands and communicates with the balancer

Verified consistency of communication between tasks
## Experimental Results

<table>
<thead>
<tr>
<th>Name</th>
<th>Program Size</th>
<th>SAT Size</th>
<th>Safe</th>
<th>Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OL</td>
<td>SL</td>
<td>GL</td>
<td>Var</td>
</tr>
<tr>
<td>nxt.ok1</td>
<td>377</td>
<td>2,265</td>
<td>6,541</td>
<td>136,944</td>
</tr>
<tr>
<td>nxt.bug1</td>
<td>378</td>
<td>2,265</td>
<td>6,541</td>
<td>136,944</td>
</tr>
<tr>
<td>nxt.ok2</td>
<td>368</td>
<td>2,322</td>
<td>6,646</td>
<td>141,305</td>
</tr>
<tr>
<td>nxt.bug2</td>
<td>385</td>
<td>2,497</td>
<td>7,398</td>
<td>144,800</td>
</tr>
<tr>
<td>nxt.ok3</td>
<td>385</td>
<td>2,497</td>
<td>7,386</td>
<td>144,234</td>
</tr>
<tr>
<td>aso.bug1</td>
<td>401</td>
<td>2,680</td>
<td>7,835</td>
<td>178,579</td>
</tr>
<tr>
<td>aso.bug2</td>
<td>400</td>
<td>2,682</td>
<td>7,785</td>
<td>176,925</td>
</tr>
<tr>
<td>aso.ok1</td>
<td>398</td>
<td>2,684</td>
<td>7,771</td>
<td>175,221</td>
</tr>
<tr>
<td>aso.bug3</td>
<td>426</td>
<td>3,263</td>
<td>10,387</td>
<td>373,426</td>
</tr>
<tr>
<td>aso.bug4</td>
<td>424</td>
<td>3,250</td>
<td>9,918</td>
<td>347,628</td>
</tr>
<tr>
<td>aso.ok2</td>
<td>421</td>
<td>3,251</td>
<td>9,932</td>
<td>348,252</td>
</tr>
</tbody>
</table>

Time bound: 100ms
No partial order reduction

**OL** = #of original LOC  **Var** = #of SAT vars
**SL** = #of seq LOC  **Clause** = #of SAT clauses
**GL** = #of goto LOC  **Safe** = whether assert valid
## Experimental Results: Partial Order Reduction

Lock-Free Reader-Writer protocols

<table>
<thead>
<tr>
<th>Name</th>
<th>Program Size</th>
<th>SAT Size</th>
<th>Safe</th>
<th>Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OL</td>
<td>SL</td>
<td>GL</td>
<td>Var</td>
</tr>
<tr>
<td>RW1</td>
<td>190</td>
<td>3,428</td>
<td>5,860</td>
<td>42,441</td>
</tr>
<tr>
<td>RW1-PO</td>
<td>190</td>
<td>5,021</td>
<td>7,626</td>
<td>45,493</td>
</tr>
<tr>
<td>RW2</td>
<td>239</td>
<td>4,814</td>
<td>8,121</td>
<td>52,171</td>
</tr>
<tr>
<td>RW2-PO</td>
<td>239</td>
<td>7,356</td>
<td>10,388</td>
<td>56,039</td>
</tr>
<tr>
<td>RW3</td>
<td>285</td>
<td>7,338</td>
<td>21,163</td>
<td>139,542</td>
</tr>
<tr>
<td>RW3-PO</td>
<td>285</td>
<td>12,002</td>
<td>26,283</td>
<td>153,826</td>
</tr>
<tr>
<td>RW4</td>
<td>244</td>
<td>7,255</td>
<td>19,745</td>
<td>117,406</td>
</tr>
<tr>
<td>RW4-PO</td>
<td>244</td>
<td>12,272</td>
<td>24,261</td>
<td>130,229</td>
</tr>
<tr>
<td>RW5</td>
<td>188</td>
<td>3,198</td>
<td>5,208</td>
<td>41,371</td>
</tr>
<tr>
<td>RW5-PO</td>
<td>188</td>
<td>4,791</td>
<td>7,138</td>
<td>45,321</td>
</tr>
<tr>
<td>RW6</td>
<td>257</td>
<td>5,231</td>
<td>7,634</td>
<td>54,829</td>
</tr>
<tr>
<td>RW6-PO</td>
<td>257</td>
<td>8,235</td>
<td>10,119</td>
<td>59,744</td>
</tr>
</tbody>
</table>

| OL = #of original LOC | SL = #of seq LOC | GL = #of goto LOC | Var = #of SAT vars | Clause = #of SAT clauses | Safe = whether assert valid |

© 2012 Carnegie Mellon University
Related Work

Sequentialization of Concurrent Programs (Lal & Reps ‘08, and others)
• Context Bounded Analysis of concurrent programs via sequentialization
• Arbitrary concurrent software
• Non-deterministic round robin scheduler
• Preserve executions with bounded number of thread preemptions
• Allow for arbitrary number of preemptions between tasks

Sequentialization of Periodic Programs (Kidd, Jagannathan, Vitek ’10)
• Same setting as this work
• Alternative sol’n: replace preemptions by non-deterministic function calls
• Additionally, supports recursion and inheritance locks
• No publicly available implementation – would be interesting to compare

Verification of Time Properties of (Models of) Real Time Embedded Systems
Conclusion

Past
- Time Bounded Verification of Periodic C Programs
- Small (but hard) toy programs
- Reader/Writer protocols (with locks and lock-free versions)
- A robot controller for LEGO MINDSTORM from nxtOSEK examples

Present
- Taking into account additional timing constraints for improved scheduling
  – arrival times, harmonicity, etc.
- A Lego Metal Stamping Robot (a.k.a. Turing Machine)
  • http://www.andrew.cmu.edu/~arieg/Rek (look for Turing Machine demo)

Future
- Verification without the time bound
- Abstraction / Refinement
- Additional communication and synchronization
  – Priority-inheritance locks, message passing
- Modeling physical aspects (i.e., environment) more faithfully
- More Case studies and model problems
Contact Information

Presenter
Arie Gurfinkel
RTSS
Telephone: +1 412-268-5800
Email: arie@cmu.edu

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

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

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