Formal Verification of Embedded Software in Medical Devices Considering Stringent Hardware Constraints

L. Cordeiro, B. Fischer, H. Chen, J. P. Marques-Silva

Lucas Cordeiro
lcc08r@ecs.soton.ac.uk
Agenda

- Introduction
- Formal Verification Methodology
- Case Study and Experimental Results
- Conclusions and Future Work
Introduction

• Design HW/SW that implements functionalities and satisfies constraints.

- Allocation, Partition, and Refinement
  - System-Design and Verification Tasks
    - Software
    - Hardware
    - Interface

- Specification
- Evolving system’s specification

- Identify market needs
- Time-to-market
- Product

• The complexity of ESW increased in embedded products
Platform-Based Design

- Design methodologies looks for solutions to reduce time-to-market, manufacturing and design costs.

Reuse and programmability

Platform
- Reference Applications
- Platform API
- Operating System
- Device Drivers
- Hardware

- The size of ESW is increasing to millions of LOC.
- Software builds are produced on a weekly or daily basis.
Verification Methodologies and Challenges

- State-of-the-art ESW verification methodologies aim to:
  
i. Generate test vectors (with constraints)
  ii. Use assertion-based verification
  iii. Use the high-level processor model during simulation

- Verification of embedded systems raises some additional challenges:
  
i. Meet the timing constraints
  ii. Handle software concurrency
  iii. Platform-dependent software
  iv. Legacy designs (written in low-level languages)
Objective of this work

- Improve coverage and reduce verification time by combining **static** and **dynamic verification**.
Bounded Model Checking

• The basic idea of BMC is to check the negation of a given property $\phi$ at a given depth.

• Given a transition system $M$, a property $\phi$ and a bound $k$:

• BMC unrolls the design $k$ times and translates it into a verification condition $\psi$ such that $\psi$ is satisfiable iff $\phi$ has a counter-example of depth less than or equal to $k$. 
Predicate Abstraction

- It abstracts data by only keeping track of certain predicates to represent the data.

- Conservative approach reduces the state space, but generates spurious counter-examples.
Agenda

- Introduction
- Formal Verification Methodology
- Case Study and Experimental Results
- Conclusions and Future Work
Verification Methodology

Consider not only higher levels of abstraction, but also the HW/SW interface.

Reflect about the design, coverage and reduce the cyclomatic complexity.

Evolving and prioritizing queue of requirements.

CTL, RTCTL, LTL and PSL.

BMC, predicate abstraction, BDDs.

Model checker

Decision Procedure

Encode model and property

Microprocessor Model

Embedded Software

Unit and Functional Tests

Product Backlog

Property Description

Verification Methodology
Proposed Approach

• In complex embedded systems, there will be modules that depend on the hardware and others that do not.

- 1st phase: Platform independent code
- 2nd phase: Platform dependent code
- 3rd phase: Domain-level

• To reason about temporal properties to assure the correctness and timeliness of the design.
Platform-Independent Software Verification

• Implement small changes in the ESW to be able to:

  i. Use model checkers;
  ii. Perform automated unit tests;
  iii. Run the ESW on the target platform.

• Include the platform-dependent software in lower level driver files:

Platform-independent software

Platform-dependent software
Platform-Independent Software Verification

- We separate into two software classes: pure and driven by the environment.

```plaintext
a = nondet_int(); /* assign arbitrary values*/
assume(a > 10 && a < 200); /* constrain the values*/
```
Platform-Dependent Software Verification

• Specify properties based on C’s `assert` macro using the microprocessor model.

```c
struct module_tc {
    unsigned int tl0;
};
extern struct module_tc oc8051_tc;
oc8051_tc.tl0=TLOW;
for(cycle=0; cycle<n; cycle++)
    next_timeframe();
assert(oc8051_tc.tl0==Y);
...
Domain-Level Verification

We use RTCTL to specify properties that involve time bounds.

\[ T = \{ t \in \mathbb{R} \mid m \leq t \leq n \} \]

compute_expr :: MIN [ rtctl_expr , rtctl_expr ] (shortest path)
| MAX [ rtctl_expr , rtctl_expr ] (longest path)

rtctl_expr :: EBF m..n p| ABF m..n p| EBG m..n p| ABG m..n p
| E [p U m..n q] | A [p U m..n q]

Log system

<table>
<thead>
<tr>
<th>Timer</th>
<th>Component</th>
<th>Function:Filename (line)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12320</td>
<td>c_LCD</td>
<td>LCD_Driver_InitModule: lcd_class_driver.c(85)</td>
</tr>
<tr>
<td>12789</td>
<td>c_LCD</td>
<td>LCD_WriteData: lcd_class_driver.c(90)</td>
</tr>
<tr>
<td>13452</td>
<td>c_LCD</td>
<td>LCD_InterfaceDescriptor: lcd_class_interface.c(102)</td>
</tr>
<tr>
<td>14216</td>
<td>c_LCD</td>
<td>LCD_InterfaceContext_Create: lcd_class_interface.c(18)</td>
</tr>
<tr>
<td>14834</td>
<td>c_LCD</td>
<td>LCD_initialize: lcd_class_interface.c(80)</td>
</tr>
</tbody>
</table>
Infrastructure

SCM Subversion

Check Modifications

CruiseControl

Send feedback

Scheduled Builds

Invoke Build System

CruiseControl Build Console

Local Builds

Local Developer Environment

Build Environment

If build process returns OK, generate and flash the file

Local Developer Environment

Verification

Embedded Unit

Embedded Unit Libraries

Reports with all metrics generated after the build process

Library

CruiseControl

invoke build system

Send feedback

Local Developer Environment

make

Embedded Unit

embUnit Libraries

Scheduled Builds

Check Modifications

CruiseControl

Send feedback

Local Builds

Local Developer Environment

Build Environment

If build process returns OK, generate and flash the file

Local Developer Environment

Verification

Embedded Unit

Embedded Unit Libraries

Reports with all metrics generated after the build process

Library

CruiseControl

invoke build system

Send feedback

Local Builds

Local Developer Environment

Build Environment

If build process returns OK, generate and flash the file

Local Developer Environment

Verification

Embedded Unit

Embedded Unit Libraries

Reports with all metrics generated after the build process

Library

CruiseControl

invoke build system

Send feedback
Agenda

• Introduction
• Formal Verification Methodology
• Case Study and Experimental Results
• Conclusions and Future Work
Medical Device Case Study

• The pulse oximeter measures the oxygen saturation and cardiac frequency.

  i. Show SpO2 and HR on each second.

  ii. Change the alarm configuration.

  iii. User interface (keyboard and a graphical display).

  iv. The design is highly optimized for life-cycle cost and effectiveness.

• Typical of many embedded real-time systems.
Formal Verification using Model Checking

• How many bugs can you find in this ANSI-C code fragment? (the compiler compiles it without errors)

```c
#define BUFFER_MAX 6400

typedef int Data8;
typedef unsigned int uData8;

static char buffer[BUFFER_MAX];
static Data8 next=0;
static uData8 buffer_size=BUFFER_MAX;

void insertLogElement(Data8 b) {
    if (next < buffer_size) {
        buffer[next] = b;
        next = (next+1)%buffer_size;
    }
}
```

(pre-production code)

First bug: the array buffer is a char data type, but b is a signed integer data type (i.e., typecast overflow might occur)

Second bug: there is a division by zero in (next+1)%buffer_size
Model Checking with NuSMV2

NuSMV2 accepts models in NuSMV language and system properties in CTL, Real-Time CTL, LTL and PSL.

Property (a): ensure that the buffer does not overflow.

MODULE log
VAR
  buffer_size : 0..255;
  nextptr : 0..255;
DEFINE
  nextptr_condition := nextptr < buffersize;
ASSIGN
  init(nextptr) := 0;
  next(nextptr) := case
    nextptr = nextptr_condition & buffer_size > 0
      :((nextptr+1) mod buffer_size);
    1 : nextptr;
  esac;
PSLSPEC AG (nextptr <= buffer_size)

NuSMV2 found a division by zero and a typecast overflow.

Ensure that on all paths, at all states on each path the formula holds.
Specifying Complex Properties in CBMC and SATABS

• We specified property (b) in LTL and translated it into Buechi Automata.

Property (b): check the data flow to compute the HR value provided by the pulse oximeter sensor hardware.

• Property (b) can be expressed as:

\[ AG(p \rightarrow Fr) \]

• Let \( p \) denote the state that the buffer contains HR. Let \( r \) denote the state that defines the respective HR value.

• Any state containing the HR raw data is eventually followed by a state representing the respective HR value.
Specifying Complex Properties in CBMC and SATABS

Example:

\[ AG(p \rightarrow Fr) \]

Property in LTL

Buechi Automata

C code
Experimental Results

- The pulse oximeter ESW contains approximately **3500 lines** of ANSI-C code.

![Table showing module details and verification metrics with notes for bugs.]

- First phase: one bug related to array bounds.
- First phase: one bug related to pointer safety.
- Third phase: one bug related to timing constraints.
- First phase: one bug related to division by zero and another bug related to typecast overflow.

<table>
<thead>
<tr>
<th>Module</th>
<th>Lines</th>
<th>RegEx</th>
<th>Properties</th>
<th>SMV2</th>
<th>V.T. (s)</th>
<th>Test Cases</th>
<th>V.T. (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MenuApp</td>
<td>53</td>
<td>53</td>
<td>5.4</td>
<td>62</td>
<td>130</td>
<td>3.7</td>
<td>6</td>
</tr>
<tr>
<td>Sensor</td>
<td>224</td>
<td>224</td>
<td>4.1</td>
<td>42</td>
<td>403</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>LCD</td>
<td>9</td>
<td>9</td>
<td>3.7</td>
<td>5</td>
<td>8</td>
<td>7</td>
<td>12</td>
</tr>
<tr>
<td>Serial</td>
<td>5</td>
<td>5</td>
<td>4.6</td>
<td>16</td>
<td>18</td>
<td>4.9</td>
<td>12</td>
</tr>
<tr>
<td>Timer</td>
<td>1</td>
<td>1</td>
<td>4.8</td>
<td>16</td>
<td>18</td>
<td>4.9</td>
<td>12</td>
</tr>
<tr>
<td>Keyboard</td>
<td>7</td>
<td>7</td>
<td>16</td>
<td>10</td>
<td>18</td>
<td>4.9</td>
<td>12</td>
</tr>
<tr>
<td>Log</td>
<td>14</td>
<td>14</td>
<td>16</td>
<td>10</td>
<td>18</td>
<td>4.9</td>
<td>12</td>
</tr>
<tr>
<td>Total</td>
<td>310</td>
<td></td>
<td>162</td>
<td>106</td>
<td>32.9</td>
<td>139</td>
<td>625</td>
</tr>
</tbody>
</table>

- The most relevant related work verified dynamically ESW from automotive domain with approximately **3000 lines of C code** in **34388 seconds (~9 h)** using SystemC models [Lettnin’08].
Conclusions and Future Work

• We have combined static and dynamic verification for “pure” and hardware-related embedded software.

• Test driven development helps reduce the cyclomatic complexity and alleviates the state explosion problem.

• The proposed methodology allowed us to find undiscovered bugs.

• We intend to verify formally ANSI-C and SystemVerilog using SAT Modulo Theories solvers.

• We aim at defining a subset of Real-Time CTL and PSL to verify more complex properties in embedded software.