Verifying Embedded C Software with Timing Constraints using an Untimed Bounded Model Checker

Raimundo Barreto\textsuperscript{1}, Lucas Cordeiro\textsuperscript{1}, and Bernd Fischer\textsuperscript{2}

\textsuperscript{1}Universidade Federal do Amazonas
\textsuperscript{2}University of Southampton

rbarreto@icomp.ufam.edu.br
Embedded Systems are everywhere

Smartphone
Embedded Systems are everywhere

Digital Pets: AIBO (Artificial Intelligence roBOt)
Embedded Systems are everywhere

Home Appliances: Microwave Oven
Embedded Systems are everywhere

Wearable Computers: Improving information and communication.
ES are everywhere

Unmanned Aerial Vehicle: Defense, Environmental, ...
We are particularly interested in the **formal verification** of critical embedded real-time software.
Other Methods

TINA
Time Petri Net Analyzer

Kronos
We propose a different method!
The main aim of this work is to propose a method to check timing properties directly in the actual C code using a (conventional) software model checker.
Original Code (multi-threaded)

Translation steps:
1. Define UP TIMER timer
2. Define CS-OVERHEAD 1
3. void *philosopher(void *arg)
   
   ```c
   int THR_ID = *((int*)arg);
   int l, r;
   l=id; r=(id+1)%N;
   pthread_mutex_lock(&frk[r]);
   pthread_mutex_lock(&frk[l]);
   pthread_mutex_unlock(&frk[l]);
   pthread_mutex_unlock(&frk[r]);
   ++count;
   assert (timer<=DEADLINE);
   }
   ```

Translated Code

```c
#define CS_OVHD 1
void *philosopher(void *arg)
{
    int left, right;
    __ESBMC_atomic_begin();
    if (_actThr != THR_ID)
        timer += (timer_sign*CS_OVHD);
    _actThr = THR_ID;
    timer += (timer_sign*3);
    left = id; right = (id+1)%N;
    if (_actThr != THR_ID)
        timer += (timer_sign*CS_OVHD);
    _actThr = THR_ID;
    __ESBMC_atomic_end();
    __ESBMC_atomic_begin();
    pthread_mutex_lock(&frk[right]);
    pthread_mutex_lock(&frk[left]);
    pthread_mutex_unlock(&frk[left]);
    pthread_mutex_unlock(&frk[right]);
    ++count;
    __ESBMC_atomic_end();
    assert (timer<=DEADLINE);
}
```
Where to use?

- There are at least two scenarios:
  - (1) for legacy code that does not have a model, or where there are no automated tools to extract a faithful model from the code; and
  - (2) when there is no guarantee that the final code is in strict accordance with the model.
Motivation

Real Time Model Checking is Really Simple

Leslie Lamport

He just represents time as an *ordinary variable* and expresses timing requirements with special *timer variables*.

ESBMC
(Efficient SMT-Based Context-Bounded Model Checker)

- ESBMC is a **context-bounded** model checker for embedded C software based on Satisfiability Modulo Theories (SMT) solver.

- It allows:
  
  (i) to verify single- and **multi-threaded** software (with shared variables and locks);

  (ii) to reason about arithmetic under- and overflow, **pointer safety**, memory leaks, array bounds, atomicity and order violations, **deadlock** and data race;

  (iii) to verify programs that make use of **bit-level**, pointers, structs, unions and fixed-point arithmetic.

  (iv) to state additional properties using **assert-statements**.
ESBMC Overview

C code scan, parse, and type-check

multi-threaded goto programs

deadlock, atomicity, user-assertions, etc

reused/extended from the Cprover framework

IRep tree

properties

scheduler

guide the symbolic execution

symbolic execution engine

BMC

verification conditions

SMT solver

QF formula generation

check satisfiability using an SMT solver

stop the generate-and-test loop if there is an error
Coarse-grained timing resolution, since we specify timing attributes for C functions.
We verify timing constraints by using user-defined assertions on explicit-defined timer variables.
On-going work

<table>
<thead>
<tr>
<th>#</th>
<th>Annotation</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>//@ DEFINE UP TIMER timer1</td>
<td>unsigned int timer1 = 0; timer1_sign = +1;</td>
</tr>
<tr>
<td>2</td>
<td>//@ DEFINE DOWN TIMER timer2</td>
<td>unsigned int timer2 = 0; timer2_sign = -1;</td>
</tr>
<tr>
<td>3</td>
<td>//@ DEFINE CS-OVERHEAD N</td>
<td>#define CS_OVHD N</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned int __actThr = UNDEF;</td>
</tr>
<tr>
<td>4</td>
<td>//@ RESET TIMER timer1 M</td>
<td>timer1 = M;</td>
</tr>
<tr>
<td>5</td>
<td>//@ ASSERT TIMER(timer1 &lt;== DL)</td>
<td>assert (timer1 &lt;== DL);</td>
</tr>
<tr>
<td></td>
<td>//@ ASSERT TIMER(timer1 &gt;== DL)</td>
<td>assert (timer1 &gt;== DL);</td>
</tr>
<tr>
<td>6</td>
<td>//@ WCET BLOCK M</td>
<td>__ESBMC_atomic_begin();</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if (__actThr != THR_ID) {</td>
</tr>
<tr>
<td></td>
<td></td>
<td>timer1 += (timer1_sign*CS_OVHD);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>timerN += (timerN_sign*CS_OVHD);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>__actThr = THR_ID;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>timer1 += (timer1_sign*M);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>timerN += (timerN_sign*M);</td>
</tr>
<tr>
<td>7</td>
<td>//@ END BLOCK</td>
<td>__ESBMC_atomic_end();</td>
</tr>
</tbody>
</table>

Fine-grained timing resolution on the block level.
Example: Bridge Crossing Problem

It is a mathematical puzzle with real-time aspects. The main aim is to verify the best-case timing properties.
Four persons, P1 to P4, have to cross a narrow bridge. It is dark, so they can cross only if they carry a light. Only one light is available and at most two persons can cross at the same time. When a pair crosses the bridge, they move at the speed of the slowest person in the pair.

What is the timing best-case for the whole group to be on the other side?
Four persons, P1 to P4, have to cross a narrow bridge. It is dark, so they can cross only if they carry a light. Only one light is available and at most two persons can cross at the same time. When a pair crosses the bridge, they move at the speed of the slowest person in the pair.

What is the timing best-case for the whole group to be on the other side?

Two observations:

1) we may have an infinite timing in the worst-case scenario, since the system can livelock (i.e. the same persons can continuously cross back and forth); and
2) the main aim of this experiment is to verify the best-case timing scenario.
Example: Bridge Crossing Problem

1) The elapsed time cannot be less than 60.
   • Modelled as:
     
     ```
     assume(timer<60);
     assert(FALSE);
     ```

     • ESBMC result was successful, since it failed to reach `assert(FALSE)` => no execution path where the condition `timer<60` is true.

     • Proof by contradiction!
Example: Bridge Crossing Problem

2) The elapsed time is greater than or equal to 60 t.u.
   • Modelled as:

     ```plaintext
     assert(timer >= 60)
     ```

   • ESBMC succeeded => asserted condition is always true.

   Conclusion: The best-case is 60 t.u.
The pulse oximeter is responsible for measuring the oxygen saturation (SpO2) and heart rate (HR) in the blood system using a non-invasive method.
# Experimental Evaluation

## Pulse Oximeter

### Packet Description

<table>
<thead>
<tr>
<th>#</th>
<th>Byte1</th>
<th>Byte2</th>
<th>Byte3</th>
<th>Byte4</th>
<th>Byte5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01</td>
<td>STATUS</td>
<td>PLETH</td>
<td>HR MSB</td>
<td>CHK</td>
</tr>
<tr>
<td>2</td>
<td>01</td>
<td>STATUS</td>
<td>PLETH</td>
<td>HR LSB</td>
<td>CHK</td>
</tr>
<tr>
<td>3</td>
<td>01</td>
<td>STATUS</td>
<td>PLETH</td>
<td>SpO2</td>
<td>CHK</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>25</td>
<td>01</td>
<td>STATUS</td>
<td>PLETH</td>
<td>reserved</td>
<td>CHK</td>
</tr>
</tbody>
</table>

We should receive 3 packets in each second
# Experimental Evaluation

## Pulse Oximeter

<table>
<thead>
<tr>
<th>ID</th>
<th>Function</th>
<th>Description</th>
<th>WCET (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1</td>
<td>receiveSensorData</td>
<td>receives data from the sensor</td>
<td>1000</td>
</tr>
<tr>
<td>f2</td>
<td>checkStatus</td>
<td>checks status</td>
<td>700</td>
</tr>
<tr>
<td>f3</td>
<td>printStatusError</td>
<td>displays status error</td>
<td>10000</td>
</tr>
<tr>
<td>f4</td>
<td>checkSum</td>
<td>calculates checksum</td>
<td>2000</td>
</tr>
<tr>
<td>f5</td>
<td>printChecksumError</td>
<td>displays checksum error</td>
<td>10000</td>
</tr>
<tr>
<td>f6</td>
<td>storeHRMSB</td>
<td>stores HR data</td>
<td>200</td>
</tr>
<tr>
<td>f7</td>
<td>storeHRLSB</td>
<td>stores HR data</td>
<td>200</td>
</tr>
<tr>
<td>f8</td>
<td>storeSpO2</td>
<td>stores SpO2 data</td>
<td>200</td>
</tr>
<tr>
<td>f9</td>
<td>averageHR</td>
<td>calculates average of HR data</td>
<td>800</td>
</tr>
<tr>
<td>f10</td>
<td>averageSpO2</td>
<td>calculates average of SpO2 data</td>
<td>800</td>
</tr>
<tr>
<td>f11</td>
<td>getHR</td>
<td>returns the stored HR value</td>
<td>200</td>
</tr>
<tr>
<td>f12</td>
<td>getSpO2</td>
<td>returns the stored SpO2 value</td>
<td>200</td>
</tr>
<tr>
<td>f13</td>
<td>printHR</td>
<td>displays HR on the LCD</td>
<td>5000</td>
</tr>
<tr>
<td>f14</td>
<td>printSpO2</td>
<td>displays SpO2 on the LCD</td>
<td>5000</td>
</tr>
<tr>
<td>f15</td>
<td>insertLog</td>
<td>inserts HR/SpO2 in RAM microcontroller</td>
<td>500</td>
</tr>
</tbody>
</table>
Experimental Evaluation
Pulse Oximeter

The implementation is relatively complex.
It has approximately 3500 lines of ANSI-C code and 80 functions.

```c
for (k=0; k<3; k++) {
    for (j=0; j<25; j++) {
        for (i=0; i<5; i++) {
            Byte[i] = receiveSensorData();
            if ((i==1) && (checkStatus(Byte[i])))
                printStatusError(LINE1);
            if ((i==4) && (checkSum(Byte)))
                printCheckSumError(LINE2);
            if (i==3) {
                if (j==0) storeHRMSB (Byte[i], k);
                if (j==1) storeHRLSB (Byte[i], k);
                if (j==2) storeSpO2 (Byte[i], k);
            }
        }
    }
}
```

# Experimental Evaluation
Pulse Oximeter

We experimented several scenarios

<table>
<thead>
<tr>
<th>ID</th>
<th>% Checksum Error</th>
<th>Time(s)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0%</td>
<td>28.9</td>
<td>successful</td>
</tr>
<tr>
<td>2</td>
<td>10%</td>
<td>20.3</td>
<td>successful</td>
</tr>
<tr>
<td>3</td>
<td>20%</td>
<td>20.2</td>
<td>successful</td>
</tr>
<tr>
<td>4</td>
<td>30%</td>
<td>19.9</td>
<td>successful</td>
</tr>
<tr>
<td>5</td>
<td>40%</td>
<td>19.9</td>
<td>failed</td>
</tr>
<tr>
<td>6</td>
<td>50%</td>
<td>21.1</td>
<td>failed</td>
</tr>
<tr>
<td>7</td>
<td>100%</td>
<td>30.2</td>
<td>failed</td>
</tr>
</tbody>
</table>
Conclusions

- This work described how to use an untimed software model checker to verify timing constraints in C code.
- No other method model checks timing constraints directly in the actual C code without explicitly generating a high-level model.
- We specified the timing behavior using an explicit-time code annotation technique.
- We provide a method able to use languages and tools not specially designed for real-time model checking.
- We show experimental evaluation on a medical device case study.
- We show that using our proposed method it is possible to investigate several scenarios.
Future Work

- To consider multi-threaded code;
- To extend the code annotation method to consider fine-grained timing constraints;
- To express context-dependent execution time bounds, e.g. loops, arrays, etc.