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

Raimundo Barreto[1], Lucas Cordeiro[1], and Bernd Fischer[2]

[1]Universidade Federal do Amazonas
[2]University of Southampton

rbarreto@icomp.ufam.edu.br

SBESC
Symposium on Computing System Engineering

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

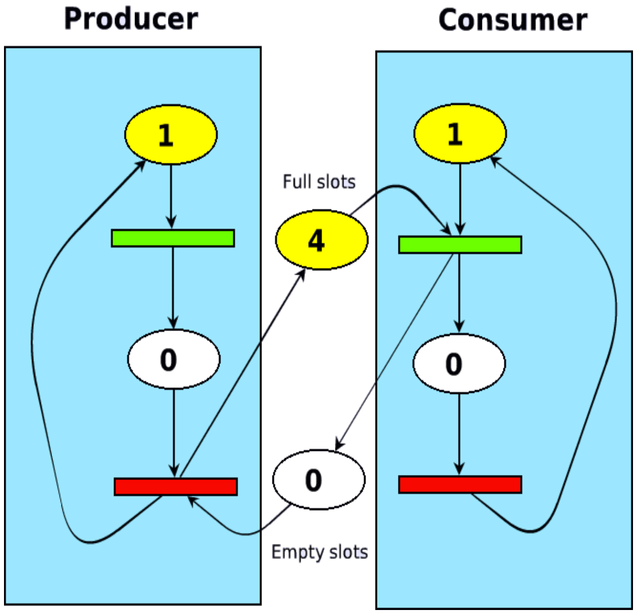
Wearable Computers: Improving information and communication.

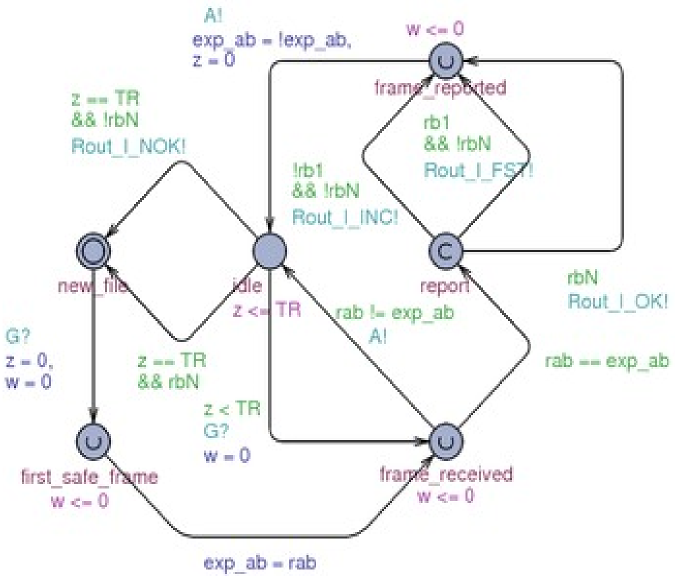
ES are everywhere

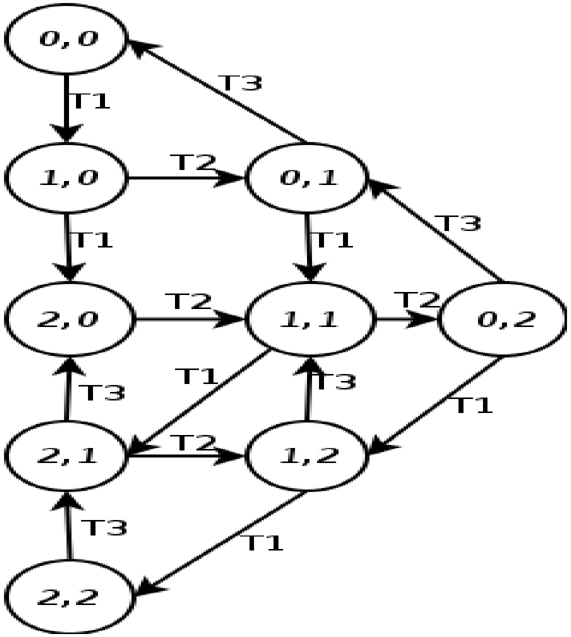
Unmanned Aeriel Vehicle: Defense, Environmental, ...

We are particularly interested in the formal verification of critical embedded real-time software.

Other Methods







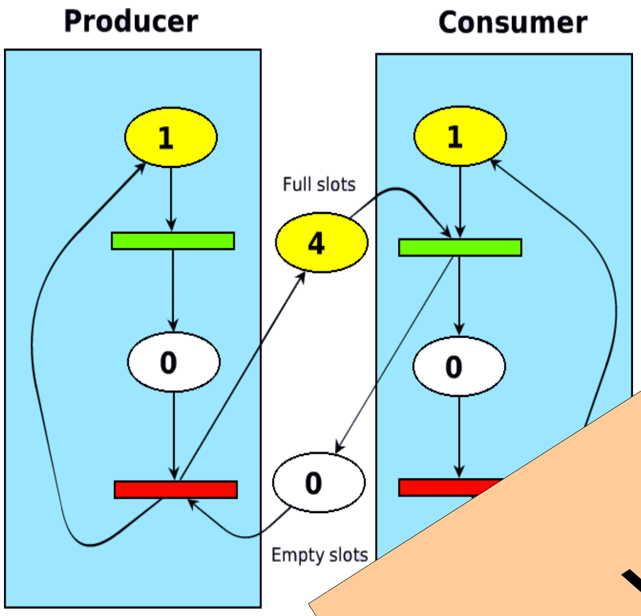
TINA

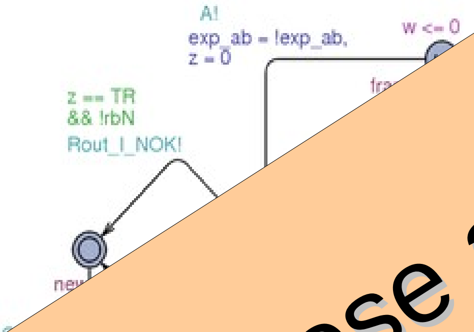
TIme petri Net Analyzer

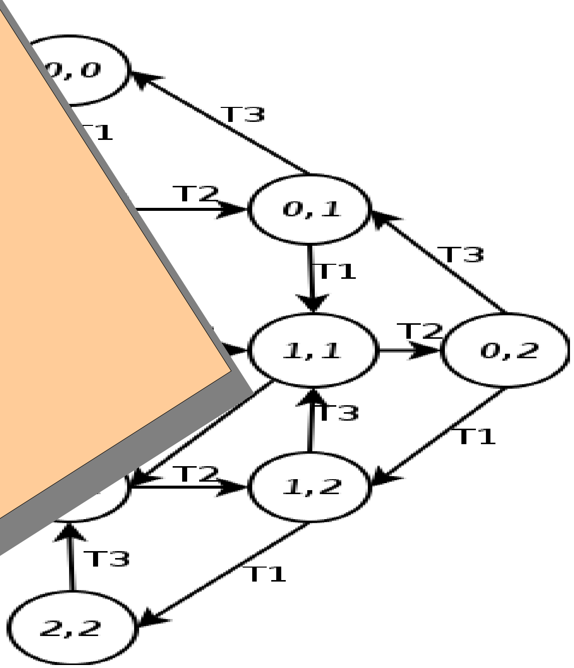
UPPAAL

Kronos

Other Methods







We propose a different method!

TIN

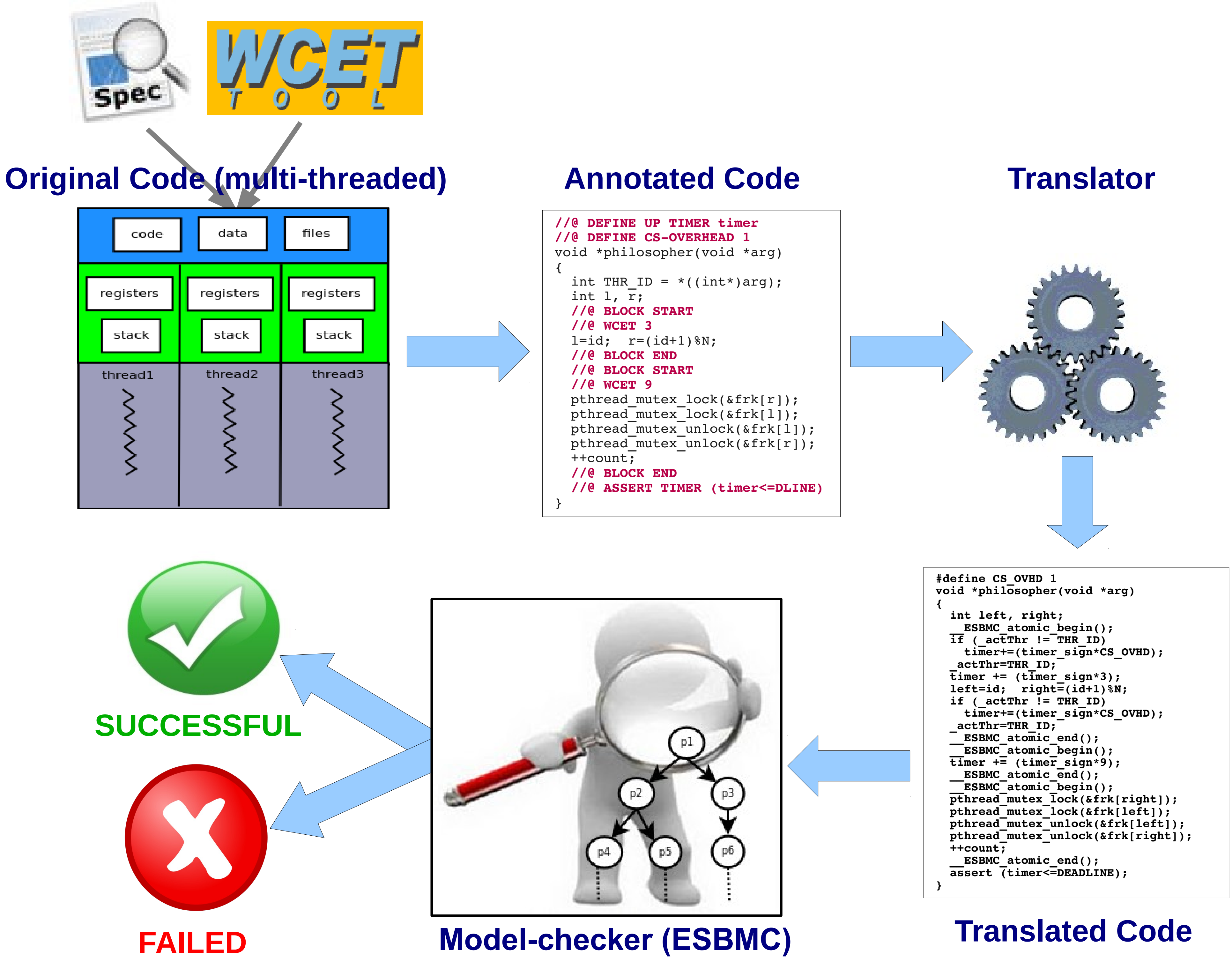
TIme petri N

Analyzer

UPPAAL

Kronos

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)

Annotated Code

```
//@ DEFINE UP TIMER timer
//@ DEFINE CS-OVERHEAD 1
void *philosopher(void *arg)
{
  int THR_ID = *((int*)arg);
  int l, r;
  //@ BLOCK START
  //@ WCET 3
  l=id;  r=(id+1)%N;
  //@ BLOCK END
  //@ BLOCK START
  //@ WCET 9
  pthread_mutex_lock(&frk[r]);
  pthread_mutex_lock(&frk[l]);
  pthread_mutex_unlock(&frk[l]);
  pthread_mutex_unlock(&frk[r]);
  ++count;
  //@ BLOCK END
  //@ ASSERT TIMER (timer<=DLINE)
}
```

Translator

```
#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();
  timer += (timer_sign*9);
  __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);
}
```

Translated Code

Model-checker (ESBMC)

SUCCESSFUL

FAILED

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

L. Lamport, “Real-time model checking is really simple,” in Correct Hardware Design and Verification Methods (CHARME’05). LNCS 3725, 2005, pp. 162–175.

Motivation

Rea

mple

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

L. Lamport, “Real-time model checking is really simple,” in Correct Hardware Design and Verification Methods (CHARME’05). LNCS 3725, 2005, pp. 162–175.

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);

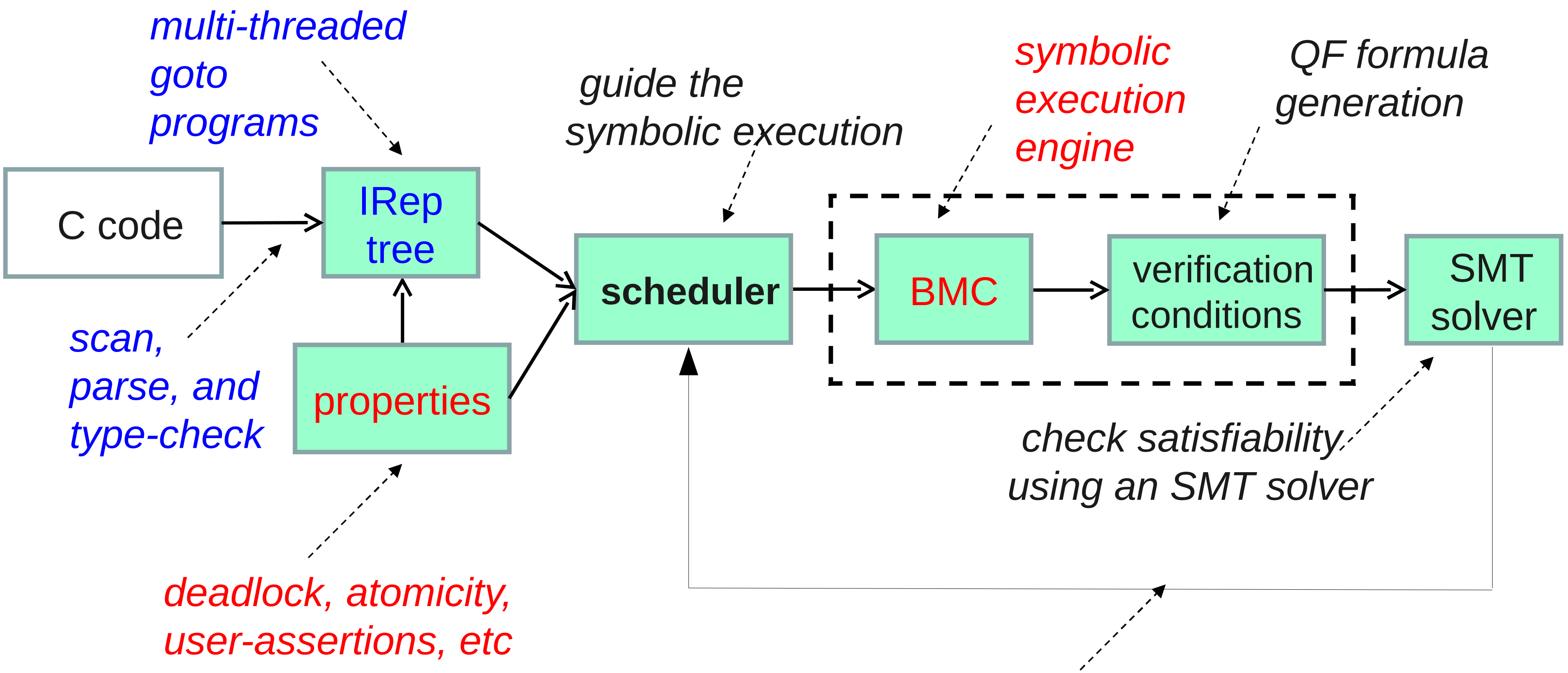
  (ii) to reason about arithmetic under- and overflow, **pointer safety**, memory leaks, array bounds, atomicity and order violations, **deadlock** and data race;

  (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



stop the generate-and-test loop if there is an error

reused/extended from the Cprover framework

Timing Annotations & Translation

```
// DEFINE-TIMER TIMER1


// DEFINE-TIMER TIMER2



...

// WCET-FUNCTION [d1]
void f1(void) {
...

}

// WCET-FUNCTION [d2]
void f2(void) {
...

}
```

```
// DEFINE-TIMER TIMER1
unsigned int TIMER1;

// DEFINE-TIMER TIMER2
unsigned int TIMER2;


...

// WCET-FUNCTION [d1]
void f1(void) {
TIMER1 += d1; TIMER2 += d1;
...
}

// WCET-FUNCTION [d2]
void f2(void) {
TIMER1 += d2; TIMER2 += d2;
...
}
```

Coarse-grained timing resolution, since
we specify timing attributes for C functions.

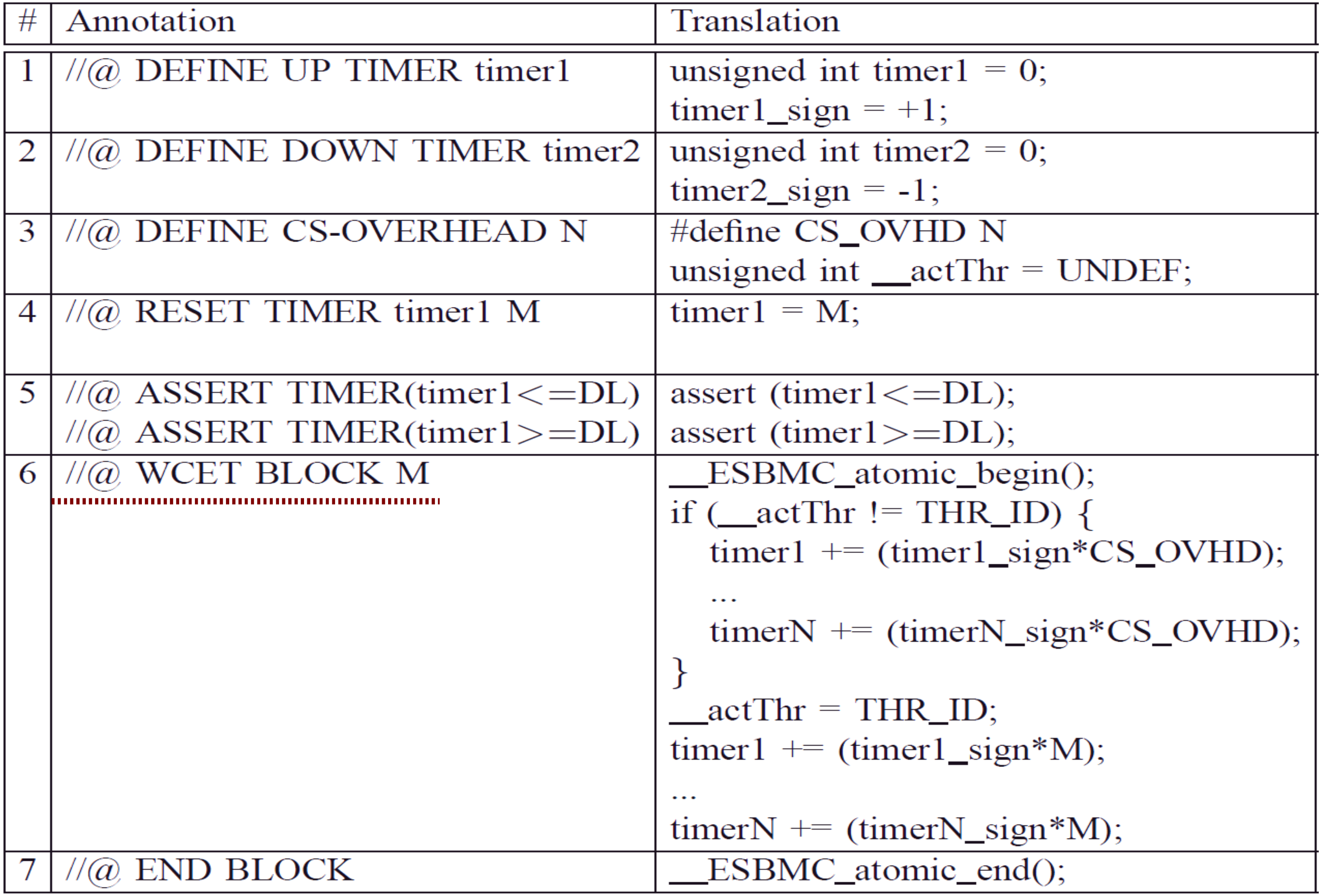
Timing Annotation & Translation

```
int main(int argc, char *argv[])
...
//@ RESET-TIMER TIMER1


//@ RESET-TIMER TIMER2


f1();
f2();

//@ ASSERT-TIMER(TIMER1<=alpha)
```

```
int main(int argc, char *argv[])
...
// RESET-TIMER TIMER1
TIMER1 = 0;

// RESET-TIMER TIMER2
TIMER2 = 0;

f1();
F2();

// ASSERT-TIMER(TIMER1<=alpha)
assert (TIMER1 <= alpha);
```

We verify timing constraints by using user-defined assertions on explicit-defined timer variables.

On-going work

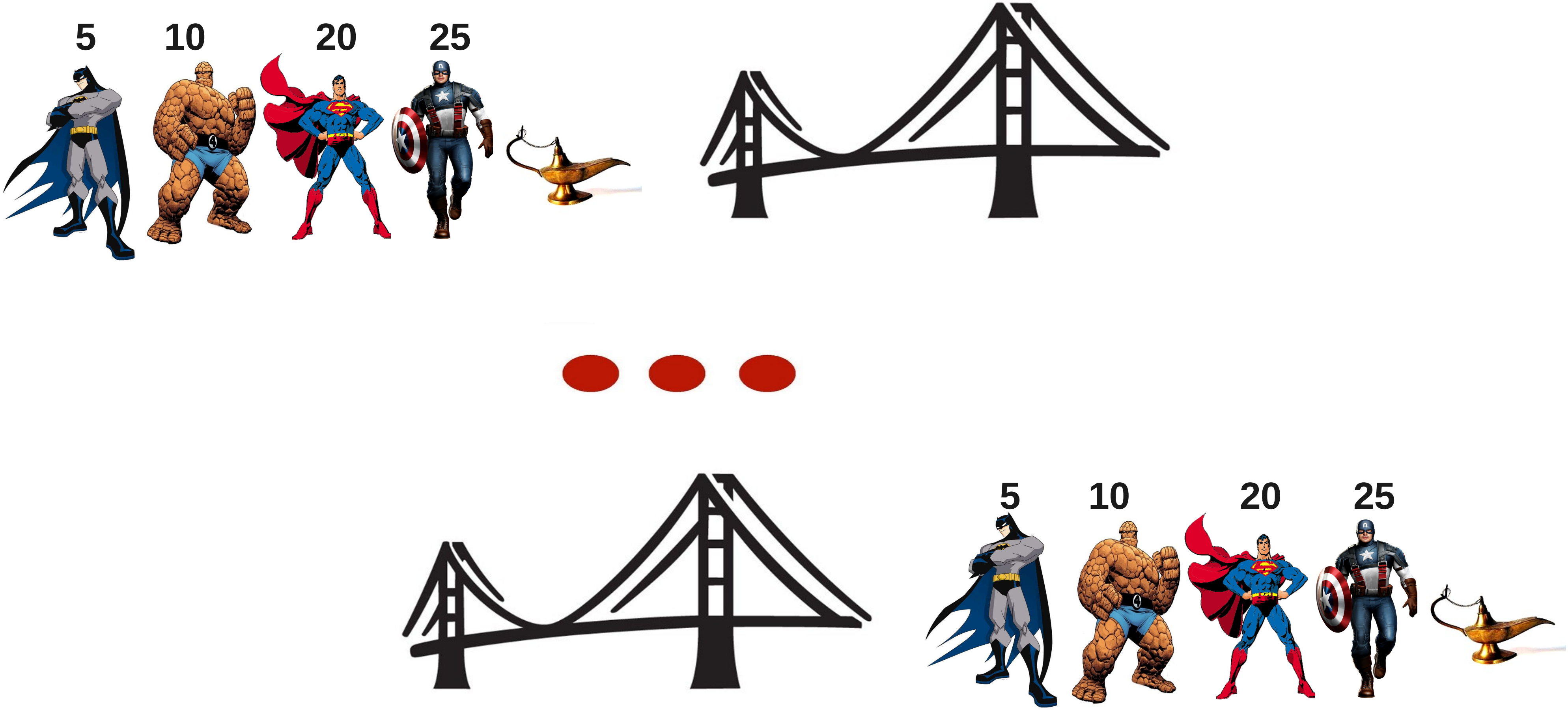


<table>
<tr><th>#</th><th>Annotation</th><th>Translation</th></tr>
<tr><td>1</td><td>//@ DEFINE UP TIMER timer1</td><td><pre>unsigned int timer1 = 0;
timer1_sign = +1;</pre></td></tr>
<tr><td>2</td><td>//@ DEFINE DOWN TIMER timer2</td><td><pre>unsigned int timer2 = 0;
timer2_sign = -1;</pre></td></tr>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

Fine-grained timing resolution on the block level.

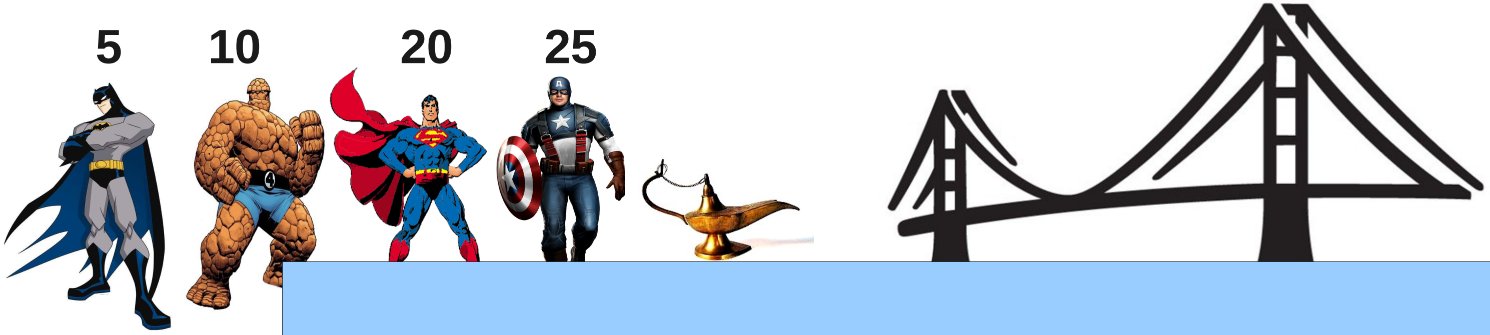
Example: Bridge Crossing Problem

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?



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.

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?

Example: Bridge Crossing Problem

1) The elapsed time cannot be less than 60.

- Modelled as:

```
assume(timer<60);

assert(FALSE);
```

  - ESBMC result was sucessful, 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:

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

  - ESBMC **succeeded** **=>** asserted condition is always true.

Conclusion: The best-case is 60 t.u.

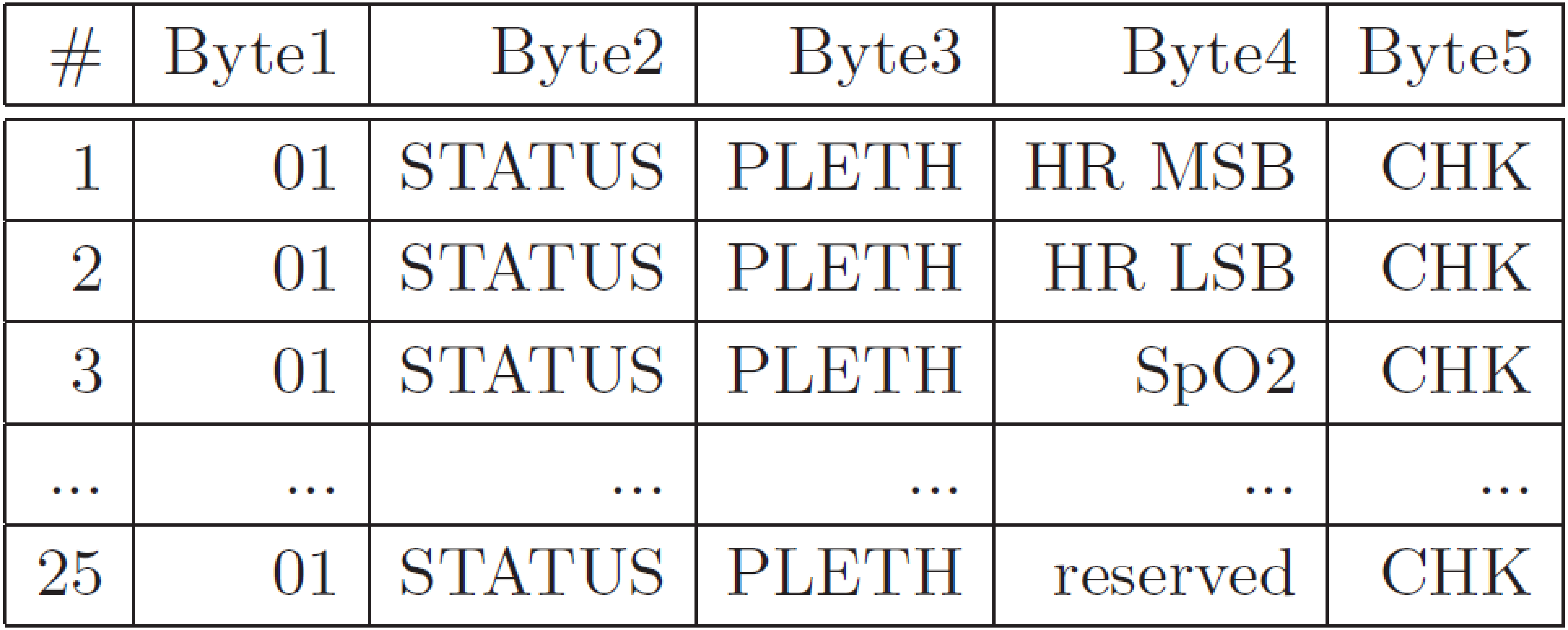
Experimental Evaluation

Pulse Oximeter

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

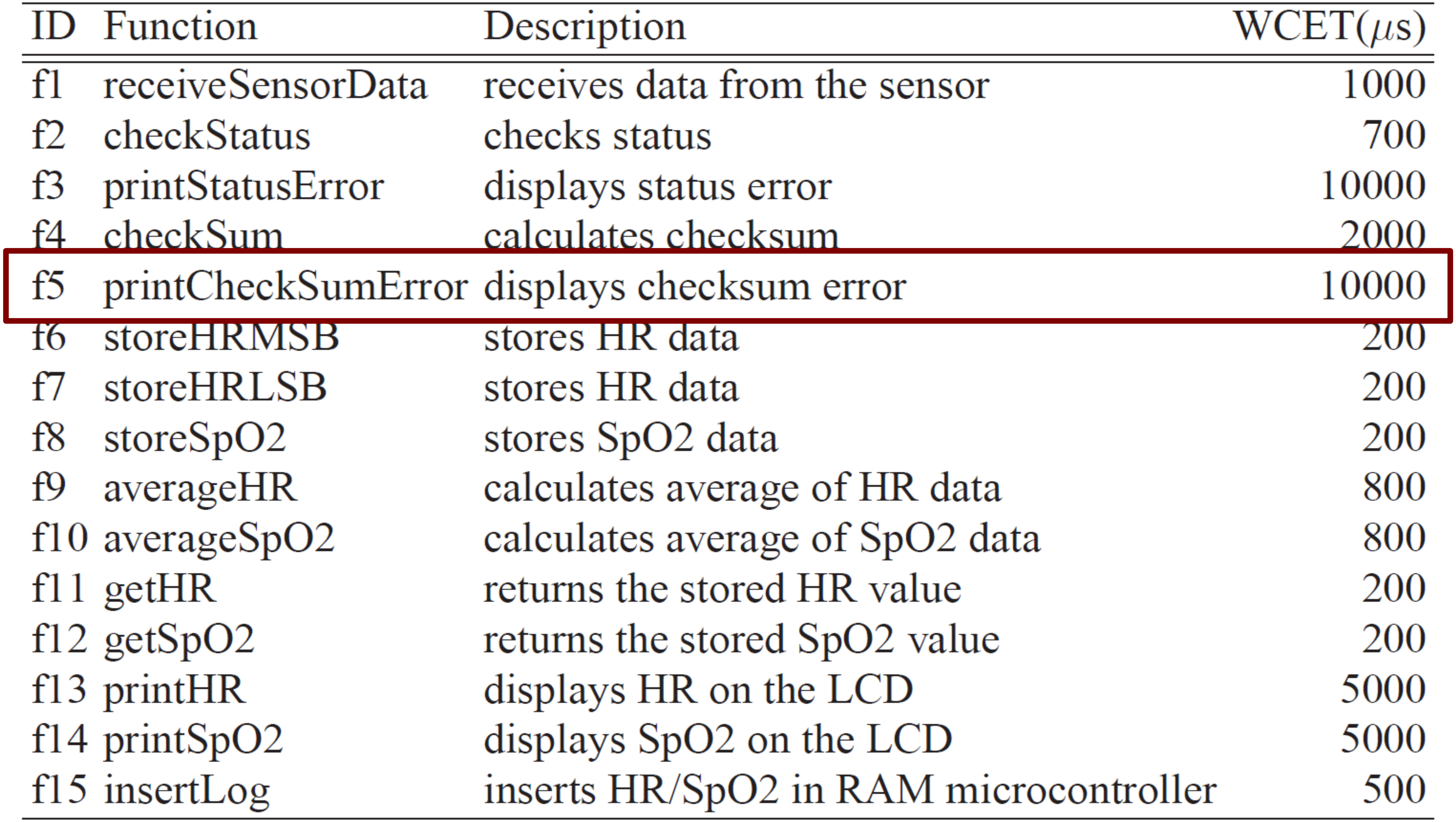
Packet Description



#	Byte1	Byte2	Byte3	Byte4	Byte5
1	01	STATUS	PLETH	HR MSB	CHK
2	01	STATUS	PLETH	HR LSB	CHK
3	01	STATUS	PLETH	SpO2	CHK
...	...	...	...	...	...
25	01	STATUS	PLETH	reserved	CHK

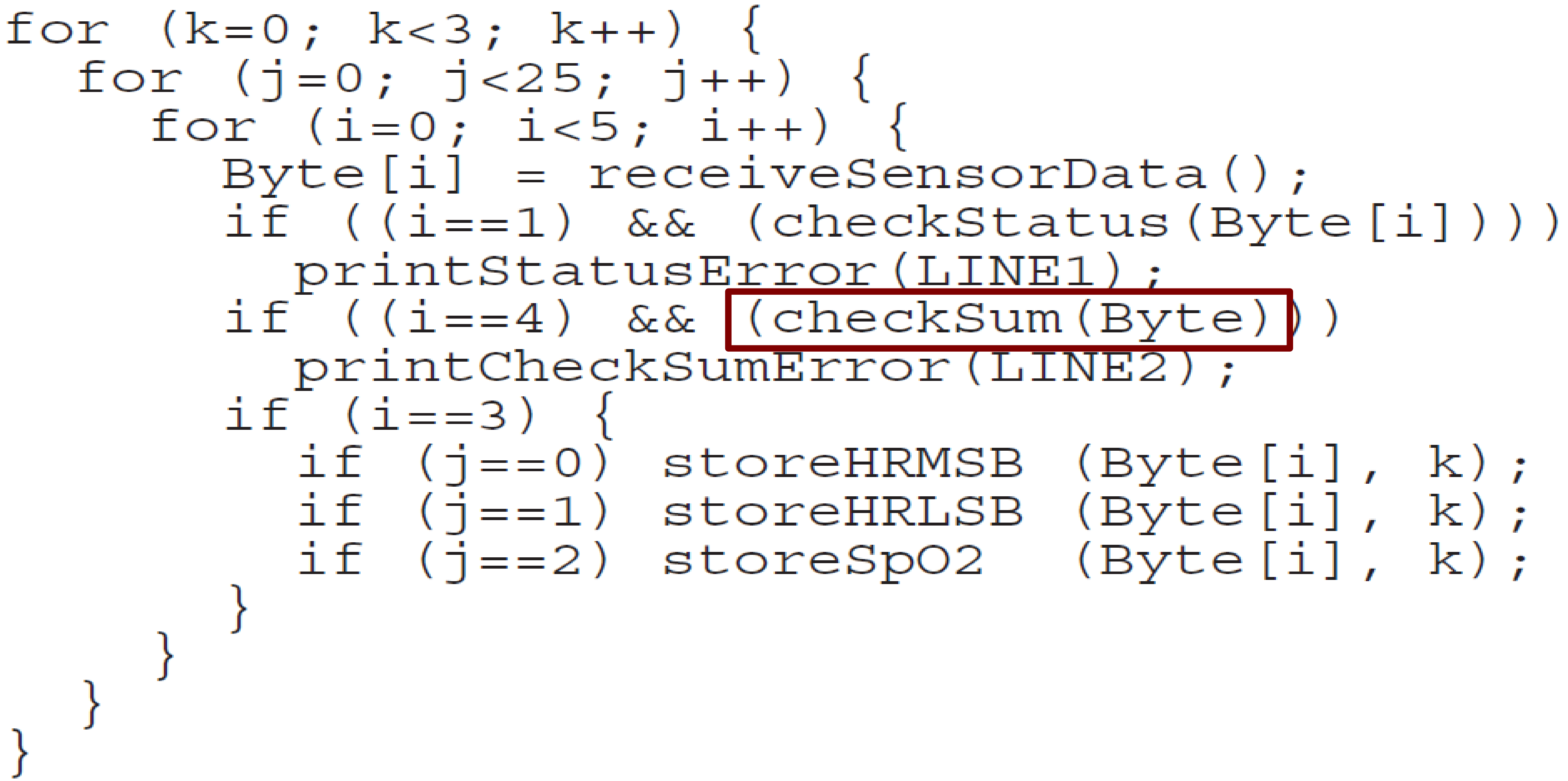
We should receive 3 packets in each second

Experimental Evaluation
Pulse Oximeter



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

Experimental Evaluation
Pulse Oximeter



```
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);
      }
    }
  }
}
```

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

Experimental Evaluation
Pulse Oximeter

We experimented several scenarios

ID	% Checksum Error	Time(s)	Result
1	0%	28.9	**successful**
2	10%	20.3	**successful**
3	20%	20.2	**successful**
4	30%	19.9	**successful**
5	40%	19.9	**failed**
6	50%	21.1	**failed**
7	100%	30.2	**failed**

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.

UNIVERSIDADE FEDERAL DO AMAZONAS

UNIVERSITY OF Southampton

http://esbmc.org

rbarreto@icomp.ufam.edu.br