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

Raimundo Barreto*, Lucas Cordeiro† and Bernd Fischer‡
* *Institute of Computing, UFAM, Manaus AM, Brazil*
Email: rbarreto@icomp.ufam.edu.br
† *Dept. of Electronics and Telecommunications, UFAM, Manaus AM, Brazil*
Email: lucascordeiro@ufam.edu.br
‡ *Electronics and Computer Science, University of Southampton, UK*
Email: b.fischer@ecs.soton.ac.uk

***Abstract*—Embedded systems are everywhere, from home appliances to critical systems such as medical devices. They usually have associated timing constraints that need to be verified. Here, we use an untimed bounded model checker to verify timing properties of embedded C programs. We describe an approach to specify discrete-time timing constraints using code annotations. The annotated code is then automatically translated to code that manipulates auxiliary timer variables and is thus suitable as input to conventional, untimed software model checkers such as ESBMC. Moreover, we can check timing constraints in the same way and at the same time as untimed system requirements, and even allow for interaction between them. We applied the proposed method in a case study, and verified timing constraints of a pulse oximeter, a noninvasive medical device that measures the oxygen saturation of arterial blood.**

***Keywords*-Bounded model checker; timing constraints; code verification**

I. Introduction

Model checking is an automatic technique for verifying finite state (concurrent) systems [5]. The main problem in model checking is the well-known state space explosion; adding real-time aspects to model checking only makes this problem worse. Usually, real-time systems are modeled by timed automata, timed Petri nets, or labeled state graphs, and verified with specialized timed model checking tools, such as TINA [1], HyTech [9], Kronos [19], or UPPAAL [11]. For example, UPPAAL uses timed automata as input and a fragment of the TCTL temporal logic [17] to prove a safety property in an explicit-state model checking style. Here, we propose a different approach. In our method, the safety property is specified in an explicit-time style [10], using discrete-time timing annotations in single-threaded ANSI-C programs. We assume that timing annotations are given externally, either by a WCET analysis of the code, or by a domain expert. We then translate automatically such annotated C code into new code that manipulates auxiliary timer variables. This code is suitable as input for a conventional (i.e., untimed) software model checker; since we are working with a discrete-time model, timing assertions can simply be interpreted as integer constraints.

In this paper we are considering single-threaded software. In our implementation, we use ESBMC [6], a bounded symbolic model checker for ANSI-C which is based on *satisfiability modulo theories* (SMT) techniques, while specialized timed model checkers typically adopt an explicit-state style (e.g., UPPAAL). Symbolic model checkers can typically explore more states than explicit-state model checkers, despite some state-space reduction techniques [3]. Moreover, symbolic model checking can easily be combined with powerful symbolic reasoning methods such as decision procedures and SMT solving [12]. This reduces not only the state space but also allows us to handle timing constraints symbolically yet precisely. Note that the timing annotations need to be treated separately from the other assertions during loop unrolling (which is a crucial step in *bounded* model checking) in order to get correct results. We avoid this problem by annotating only function definitions.

Many safety-critical software systems are written in low-level languages such as ANSI-C. However, to the best of our knowledge, there is at present no tool that translates C code with timing constraints to either timed automata or timed Petri nets. The main aim of this paper is to propose a method to check timing properties directly in the actual C code using a (conventional) software model checker. Thus, we can check timing properties as well as safety and (untimed) liveness properties (see [6]). The proposed solution should not be considered as an alternative to other methods, but rather as complementary. There are at least two scenarios in which it can be used: (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. We illustrate our approach through an industrial case study involving a medical device called a pulse oximeter. Our experiments show that our technique can be used efficiently for verifying embedded real-time systems using an existing untimed model checker.

978-0-7695-4641-4/11 $26.00 © 2011 IEEE
DOI 10.1109/SBESC.2011.19

The main contribution of this work is to describe an approach to check timing properties in the same way as safety and liveness properties for untimed systems. Specifically: we use code annotation to express timing properties; we describe our automatic translation from the annotated code to a code suitable for model checking; and we report experiments on a medical device.

II. Model Checking with ESBMC

ESBMC is a context-bounded model checker for embedded ANSI-C software based on SMT solvers, which allows the verification of single- and multi-threaded software with shared variables and locks [7], [6], although we focus on single-threaded software here. ESBMC supports full ANSI-C, and can verify programs that make use of bit-level operations, arrays, pointers, structs, unions, memory allocation and fixed-point arithmetic. It can efficiently reason about arithmetic under- and overflows, pointer safety, memory leaks, array bounds violations, atomicity and order violations, local and global deadlocks, data races, and user-specified assertions. Here we use ESBMC simply as a "black-box".

In ESBMC, the program to be analyzed is modelled as a state transition system $M = (S, R, s_0)$, which is extracted from the control-flow graph (CFG). S represents the set of states, $R \subseteq S \times S$ represents the set of transitions (i.e., pairs of states specifying how the system can move from state to state) and $s_0 \subseteq S$ represents the set of initial states. A state $s \in S$ consists of the value of the program counter *pc* and the values of all program variables. An initial state s_0 assigns the initial program location of the CFG to *pc*. We identify each transition $\gamma = (s_i, s_{i+1}) \in R$ between two states s_i and s_{i+1} with a logical formula $\gamma(s_i, s_{i+1})$ that captures the constraints on the corresponding values of the program counter and the program variables.

Given the transition system *M*, a property ϕ, and a bound *k*, BMC unrolls the system *k* times and translates it into a verification condition (VC) ψ such that ψ is satisfiable if and only if ϕ has a counter-example of length k or less. The VC ψ is a quantifier-free formula in a decidable subset of first-order logic, which is then checked for satisfiability by an SMT solver. In this work, we are interested in checking safety properties of single-threaded programs. The associated model checking problem is formulated by constructing the following logical formula: $\psi_k = I(s_0) \wedge \bigvee_{i=0}^{k} \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1}) \wedge \neg\phi(s_i)$. Here, ϕ is a safety property, I the set of initial states of M and $\gamma(s_j, s_{j+1})$ the transition relation of M between time steps j and $j+1$. Hence, $I(s_0) \wedge \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1})$ represents the executions of M of length i and ψ_k can be satisfied if and only if for some $i \leq k$ there exists a reachable state at time step i in which ϕ is violated. If ψ_k is satisfiable, then the SMT solver provides a satisfying assignment, from which we can extract the values of the program variables to construct a counter-example. A counter-example for a property ϕ is a sequence of states $s_0, s_1, \ldots, s_k$ with $s_0 \in S_0$, $s_k \in S$, and $\gamma(s_i, s_{i+1})$ for $0 \leq i < k$. If ψ_k is unsatisfiable, we can conclude that no error state is reachable in k steps or less.

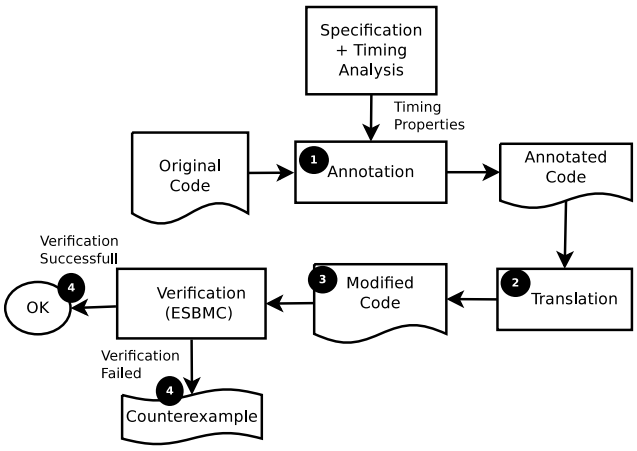

Figure 1. Overview of the Proposed Method

III. Proposed Method

This section describes the method proposed to verify timing properties on single-threaded C code. Figure 1 gives an overview of the approach. It is divided into four steps. The first step is to add timing constraints to the source code. Such annotations come from either a discrete timing model, a timing analyzer tool, or a domain expert. As usual, the annotations are just comments that are processed by a specific tool. The second step is the automatic translation from the annotated source code to new code that can be verified by the untimed model-checker. The third step is to check the translated code with the ESBMC model checker. Finally, the last step, evaluates ESBMC's results. As shown, we check timing properties using assertions.

A. Timed Programming Model

The proposed method aims to pragmatically assist developers in the specification and analysis of timing constraints in C code. However, we are just considering a coarse-grained timing constraint resolution on the level of functions. Thus, what we propose is (i) to associate with each function f_i a worst-case duration $d_i \geq 0$; (ii) to define explicit timer variables (or clocks) ($\mathcal{T}$), for expressing timing constraints; (iii) to introduce assertions on timer variables to check timing properties; and (iv) to introduce a *reset* operation to restart the timers. Therefore, when the program is executed, the timer variables are incremented by the respective duration d_i of the called function f_i, and assertions are used to ensure that computations are within timing constraints.

Formally, we consider the semantics of a sequential program $\mathcal{P}$ to be represented by the 5-tuple $\langle \mathcal{S}, s_0, \mathcal{V}, \mathcal{F}, \rightarrow \rangle$, where:

- S is a finite set of states of $\mathcal{P}$;
- s_0 is the initial state;
- $\mathcal{V} = \{v_1, v_2, \cdots, v_z\}$ is a finite set of data variables (local and global);
- $\mathcal{F} = \{f_1, f_2, \cdots, f_w\}$ is a finite set of functions that may change variables in $\mathcal{V}$; we suppose that each function f_k is executed to completion;
- $\rightarrow \subseteq \mathcal{S} \times \mathcal{F} \times \mathcal{S}$ is a finite set of labeled transitions, such that a state transition in $\langle s_i, f_k, s_j \rangle$, is represented by $s_i \xrightarrow{f_k} s_j$, $\forall s_i, s_j \in S, i \neq j, f_k \in \mathcal{F}$.

Let $\pi[n \ldots m]$ for $0 \leq n < m \in \mathbb{N}$ be an *execution path* denoted by a finite sequence $s_n \xrightarrow{f_{k_1}} s_{n+1} \xrightarrow{f_{k_2}} \cdots \xrightarrow{f_{k_q}} s_m$ with $m - n$ transitions and $m - n + 1$ states. As example, suppose we have $\mathcal{F} = \{f_1, f_2, f_3, f_4, f_5\}$; we may define the following execution path $\pi[0..5] = s_0 \xrightarrow{f_1} s_1 \xrightarrow{f_3} s_2 \xrightarrow{f_5} s_3 \xrightarrow{f_2} s_4 \xrightarrow{f_1} s_5$.

In order to introduce timing constraints into the program, we change the original program $\mathcal{P}$ into another program $\mathcal{P}' = \langle \mathcal{S}, s_0, \mathcal{V}', \mathcal{F}, \rightarrow, \mathcal{T}, \mathcal{A}, \mathcal{R} \rangle$ where:

- $\mathcal{T} = \{t_1, t_2, \cdots, t_p\}$ is a finite set of timer variables, with $\mathcal{V} \cap \mathcal{T} = \emptyset$;
- $\mathcal{A} = 2^{\mathcal{T}} \rightarrow \{\text{true}, \text{false}\}$, where $a_i \in \mathcal{A}$ is a special function that asserts on timer variables;
- $\mathcal{R} = \mathcal{T} \rightarrow \mathbb{N}$, where r_i is a special function that resets a timer variable to a specific value, usually zero.
- $\mathcal{V}' = \mathcal{V} \cup \mathcal{T}$;

We define $D : \mathcal{F} \mapsto \mathbb{N}$ as the worst-case duration of a function, such that

$$D(f_i),\ \forall f_i \in \mathcal{F} = \begin{cases} d_i \in \mathbb{N}, & \text{if}(f_i \in \mathcal{F}) \\ 0, & \text{if}(f_i \in \mathcal{A}) \\ 0, & \text{if}(f_i \in \mathcal{R}) \end{cases}$$

Therefore, we may express the duration $D(\pi[n..m]) = \sum_{i=n}^{m} D(f_{\pi[i]})$ of such a finite sequence $\pi[n..m]$ representing the time elapsed from s_n to s_m. As example, suppose we have $\mathcal{F} = \{f_1, f_2, f_3, f_4, f_5\}$; $\mathcal{T} = \{t_1, t_2\}$; $\mathcal{A} = \{a_1(t_1), a_2(t_1), a_3(t_2)\}$; $\mathcal{R} = \{r_1(t_1), r_2(t_2)\}$; and the execution path $\pi[0..11] = s_0 \xrightarrow{r_1(t_1)} s_1 \xrightarrow{r_2(t_2)} s_2 \xrightarrow{f_1} s_3 \xrightarrow{f_3} s_4 \xrightarrow{a_1(t_1)} s_5 \xrightarrow{r_1(t_1)} s_6 \xrightarrow{f_5} s_7 \xrightarrow{f_2} s_8 \xrightarrow{a_2(t_1)} s_9 \xrightarrow{f_4} s_{10} \xrightarrow{a_3(t_2)} s_{11}$. We can conclude that $D(\pi[0..11]) = \sum_{i=1}^{11} D(f_{\pi[i]}) = D(f_1) + D(f_3) + D(f_5) + D(f_2) + D(f_4)$. As we can see, in the execution path $\pi[0..11]$ we have three timing assertions: $a_1(t_1)$, $a_2(t_1)$, and $a_3(t_2)$; and three timer resets: $r_1(t_1), r_2(t_2)$, and $r_1(t_1)$.

B. Annotation of Timing Constraints

The inclusion of timing constraints in the source code is particularly interesting since they can automatically be checked as the program is developed. To annotate the timing constraints in the code we use a special kind of C comment in such a way that this annotation does not change the code itself. In this way, the same annotated code can be compiled by any C compiler without breaking the compilation. The proposal is to have four kinds of annotations:

- `//@ DEFINE-TIMER <timer-name>`. Defines a new timer variable *timer-name* which is automatically declared as an unsigned int variable. Using this annotation we can add the set $\mathcal{T}$ to the code.
- `//@ RESET-TIMER <timer-name>`. Resets the timer variable to zero. Using this annotation we can add the set $\mathcal{R}$ to the code.
- `//@ ASSERT-TIMER (<logic-expr>)`. Checks a user defined assert. This annotation specifically is useful to check timing properties, where the assertion language consists in arithmetic operations with timer variables. Using this annotation we can add the set $\mathcal{A}$ to the code.
- `//@ WCET-FUNCTION [<int-expr>]`. Defines the WCET of the next defined function. We rely on a timing analyzer tool to predict worst-case timing bounds (see [18]). Using this annotation we can add the function D to the code. We show only how to specifiy timing constraints in the source code on functions, because we are just considering a coarse-grained timing constraint resolution.

Fig. 2(a) shows an example of code annotation corresponding to the example shown in Section III-A. Even though all timer variables are incremented together, the fact that we have defined more than one timer implies that we may verify several timing constraints. In the example of Fig. 2, TIMER1 is checking local timing constraints. Firstly, this timer verifies timing constraint related to functions f_1 and f_2, and later this same timer verifies over the functions f_3 and f_4. On the other hand, the timer variable TIMER2 is used to verify the complete behavior of the system, i.e., the function calls from f_1 up to f_5.

C. Translation and Verification

The translation looks for comments that start by `//@` and treats them appropriately. The translation of the code shown in Figure 2(a) can be seen in Figure 2(b). This translation is carried out automatically by a specific tool. It is important to emphasize that the user has first to run the model checker to find conventional errors (e.g., buffer overflow, arithmetic overflow, memory leaks, etc), and then run the model checker to search for timing violations in the translated code. After translation, this new code can be analyzed by ESBMC using user-specified assert statements. In the proposed method the assertions will be the way to check timing properties. In the code of Figure 2(b) we can see three timing verifications. It is worth pointing out that the verification results depend on the accuracy of the WCET estimates; in particular, if the WCET estimates are not tight enough, the verification may fail but the program may execute within the specified time limits.

```
//@ DEFINE-TIMER TIMER1

//@ DEFINE-TIMER TIMER2

...
//@ WCET-FUNCTION [d1]
void f1(void)...
//@ WCET-FUNCTION [d2]
void f2(void)..
//@ WCET-FUNCTION [d3]
void f3(void)...
//@ WCET-FUNCTION [d4]
void f4(void)...
//@ WCET-FUNCTION [d5]
void f5(void)...
...
int main(int argc, char *argv[])
...
//@ RESET-TIMER TIMER1

//@ RESET-TIMER TIMER2

f1(); f2();
//@ ASSERT-TIMER (TIMER1 <= alpha)

//@ RESET-TIMER TIMER1

f3(); f4();
//@ ASSERT-TIMER (TIMER1 <= beta)

f5();
//@ ASSERT-TIMER (TIMER2 <= gamma)

...
```

(a)

```
// 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;...}
// WCET-FUNCTION [d3]
void f3(void) {TIMER1 += d3; TIMER2 += d3;...}
// WCET-FUNCTION [d4]
void f4(void) {TIMER1 += d4; TIMER2 += d4;...}
// WCET-FUNCTION [d5]
void f5(void) {TIMER1 += d5; TIMER2 += d5;...}
...
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);
// RESET-TIMER TIMER1
TIMER1 = 0;
f3(); f4();
// ASSERT-TIMER (TIMER1 <= beta)
assert (TIMER1 <= beta);
f5();
// ASSERT-TIMER (TIMER2 <= gamma)
assert (TIMER2 <= gamma);
...
```

(b)

Figure 2. (a) Example of Annotated C Code; and (b) Translation Result

D. Verifying the Bridge Crossing Problem

The bridge-crossing problem is a mathematical puzzle with real-time aspects [14]. Four persons, P_1 to P_4, 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. Therefore any solution requires that, after two persons cross the bridge, one of them returns, bringing back the light for any remaining person(s). The four persons have different maximal speeds: P_i crosses in t_i time units (t.u.), and we assume that $t_1 \leq t_2 \leq t_3 \leq t_4$. When a pair crosses the bridge, they move at the speed of the slowest person in the pair. Consider that $t_1 = 5$; $t_2 = 10$; $t_3 = 20$; and $t_4 = 25$, the question is: how much time is required before the whole group is on the other side? Rote [14] pointed out that the most obvious solution is to let the fastest person (P1) accompany each other person over the bridge and return alone with the lamp. In this case, the total duration of this solution is $t_2 + t_1 + t_3 + t_1 + t_4 = 2t_1 + t_2 + t_3 + t_4 = 65$ t.u. However, the obvious solution is not optimal. The right way to solve this problem is to let P3 and P4 cross in the middle, i.e., as we have five crossing, P3 and P4 have to cross in the third crossing. In this case, the new total duration is $t_2 + t_1 + t_4 + t_2 + t_2 = t_1 + 3t_2 + t_4 = 60$ t.u.

We implemented the bridge crossing problem in C and submitted it to the ESBMC model checker. Each configuration of persons crossing the bridge was represented by one C function with their respective WCET, which corresponds to the speed of the slowest person in the pair. Since the system can *livelock* (i.e. the same persons can continuously cross back and forth), we may have an infinite timing in the worst-case scenario. Thus, the main aim of this experiment is to verify the best-case timing. We first verified that the elapsed time cannot be less than 60. For this, we included `assume (timer<60);` followed by `assert (FALSE);` in the code. ESBMC failed to reach the statement `assert(FALSE)` in the code, because there is no execution path where the assumed condition can be true. This shows that the best-case solution cannot be better than 60 t.u. The ESBMC execution time was 12m43s to give this result. We also verified that the elapsed time is greater than or equal to 60 t.u. for all possible solutions by adding `assert(timer >= 60)`. ESBMC succeeded on this, which means that the model checker found that in all execution paths the asserted condition is true. With these two results, we may conclude that 60 t.u. is thus the optimal (best-case) solution. ESBMC spent 16m28s to give this result. All experiments were conducted on an otherwise idle Intel Xeon 5160, 3GHz server with 24 GB of RAM running Linux OS. We chose ESBMC v.1.16 (64bits) as our untimed bounded model checker.

IV. Pulse Oximeter Case Study

This section describes the main characteristics of the pulse oximeter and shows results on the application of the model

checker ESBMC to the verification of timing constraints. The pulse oximeter is responsible for measuring the oxygen saturation (SpO2) and heart rate (HR) in the blood system using a non-invasive method. This device was used as case study in [7] to raise the coverage of tests in embedded system combining hardware and software components. The implementation is relatively complex, comprising approximately 3500 lines of ANSI-C code and 80 functions.

Architecture. The architecture consists of four components: sensor, data acquisition module, microcontroller, and LCD display. The sensor captures oxygen saturation and heart rate data of the patient. The data acquisition module has an interface for communication with the sensor, an ASIC (Application-Specific Integrated Circuit) component, and a serial communication interface (RS-232). The ASIC component provides the values of SpO2 and HR data in the serial port. The microcontroller receives this data, via the serial port, processes it, and displays it on the LCD. A serial data frame consists of five bytes; and a packet consists of 25 frames. Three packets (375 bytes) are transmitted each second. Byte2 of each frame is the status byte (whether the sensor is not connected, for instance). Byte4 represents HR and SpO2 data; and Byte5 is the checksum.

Timing Constraints. In the context of timing constraints, the following functional requirements are considered: (i) The system has to read all HR and SpO2 data in at most one second. (ii) The software must check whether the frames sent by the sensor are correct (i.e., no checksum and status errors), and show in the LCD if any problem is found; (iii) The user should be able to see, every second, the data of heart rate and oxygen saturation in the patient's blood; and we have to store patient's information and to show in the LCD display. (iv) The system must allow users to store data on HR and SpO2 in the external memory of the microcontroller. In order to reach timing constraints we have to take into account several important questions, such as, the maximum frequency of the serial communication (in this case 9600bps), the amount of bytes sent by the sensor device, the time it takes to store data in the external memory, and the time to calculate and to show checksum/status errors.

Code Annotation. The timing constraints for this project are shown in Table I. They come from either the specification or a domain expert. As presented before (see Section III-B), these timing constraints are annotated into the code.

Verification Results. The pulse oximeter code is part of a real implementation. The code adopted, and the verification results are publicly available at http://esbmc.org. In order to verify the timing constraints using ESBMC, we had to isolate hardware-dependent code. With this aim we used `#if`, `#else`, and `#endif` preprocessor directives. This experiment verifies if in all execution paths the timing constraints are met when implementing the four functional requirements (FR_1, FR_2, FR_3, and FR_4). This program behavior is explained as follows: The specification considers

Table I
TIMING INFORMATION

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	500

Table II
EXPERIMENTAL RESULTS

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

that we should read three packets of data per second. Each packet has twenty five frames. Each frame has five bytes. In this way we have to:

1) read data bytes calling function *receiveSensorData*;
2) for each byte read: (a) to check status of the second byte of each frame by calling function *checkStatus*; if there is an error, the function *printStatusError* should be called; (b) to check the fifth byte of each frame by calling function *checkSum*; if there is an error, it should be called the function *printCheckSumError*; (c) to read the fourth byte of first frame and to call function *storeHRMSB*; (d) to read the fourth byte of second frame and to call function *storeHRLSB*; and (e) to read the fourth byte of third frame and to call function *storeSpO2*.
3) call the functions *average_HR*, *average_SpO2*, *getHR*, *getSpO2*, *printHR* with HR value as argument, *printSpO2* with SpO2 value as argument, *insertLog* with HR value as argument, and *insertLog* with SpO2 value as argument.

Figure 3 shows part of the pulse oximeter code submitted to ESBMC. Table II shows the experimental results. We analyzed seven scenarios, taking into account the percentage of checksum errors. The percentages considered were 0%, 10%, 20%, 30%, 40%, 50%, and 100%. Excepting the best scenario (0%) and worst scenario (100%), all timing performance was 20s in average. As presented in Table II, the experiments show that the system will reach timing

```
...
// DEFINE-TIMER TIMER;
unsigned int TIMER;
...
//@ WCET-FUNCTION [5000]
void printHR (unsigned int line, unsigned int valueHR)
{
TIMER += 5000;
    char sHR[16];
    sprintf(sHR, "HR:%d\n", valueHR);
    printLCD(sHR, line, 1);
}
...
int main(void) {
...
// RESET-TIMER TIMER;
TIMER=0;
...
  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);
        }
      }
    }
  }

  averageHR();
  averageSpO2();
  HR = getHR();
  SpO2 = getSpO2();
  printHR(LINE1, HR);
  printSpO2(LINE2, SpO2);
  insertLog(HR);
  insertLog(SpO2);

  // ASSERT-TIMER (TIMER < 1000000)   // one second;
  assert (TIMER < 1000000);
  ...
}
```

Figure 3. Code for running in the ESBMC model-checker

constraints up to 30% of checksum errors.

V. Related Work

Lamport [10] argues that most real-time specifications can be verified using existing languages and methods. He proposed to represent time as an ordinary variable, in this case variable `now`, which is incremented by an action (`Tick`), and express timing requirements with a special timer variable in such a way that such specifications can be verified with an conventional model checker. He calls this method as model checking explicit-time specifications. We follow the same general approach. However, Lamport proposes to specify the system and timing constraints using TLA+ (Temporal Logic of Actions), which is a high-level mathematical language. The problem is that the learning curve of TLA+ may be steep.

Ostroff and Ng [13] present a framework that allows specification and verification of discrete real-time properties of reactive systems. They consider a Timed Transition Model (TTM) as underlying computational model, and Real-Time Temporal Logic (RTTL) as the requirements specification language. They map the model and specification into a finite state fair transition systems, which may be input to a (untimed) tool, in this case STeP model-checker [2], for checking timing properties. One problem of this method is that the size of the formulas grow according to the bounds that must be checked. Since the cost of checking a linear time formula is exponential in the size of the formula, these procedures are only useful for small bounds.

Chun and Hung [4] propose a new class of Duration Calculus (DC) called $DC^*_{\leq 1}$, whose formulas correspond to expressions over the set of state occurrence for one time unit (or less). They adopt conventional variables to implement relative time. They model the real-time system using DC and convert its components into $DC^*_{\leq 1}$ specifications. Each $DC^*_{\leq 1}$ specifications is translated to an automata model. In this way, the whole system is modeled by the synchronisation of several automatas. Later, the resulting automata is translated to the Promela language. However, it is not clear what timing constraint was verified in the case study (Biphase Mark Protocol).

Ganty and Majumdar [8] show that checking safety properties for real-time event-driven programs is undecidable. The undecidability proof for the safety checking problem uses an encoding of the execution of a 2-counter machine as a real-time event-driven program. They suggest to use higher-level languages, such as Giotto, which statically restricts the ability to post tasks arbitrarily. In this case, these higher-level languages ensure that for any program, at any point of the execution, there is at most a bounded, statically determined, number of pending calls. In this case, just by finiteness of the state space, all verification problems are decidable.

Sifakis et al. [16] proposed a modeling methodology for real-time systems programmed in the Esterel synchronous language extended with timing constraints specified as annotations. The program is compiled with the Esterel compiler SAXO-RT that generates sequential C code, where such code is instrumented by best and worst execution times provided by annotations. The execution of the instrumented C code generates the timed automaton model of the system, which is verified by KRONOS [19] timing verifier. There are some similarities with our work. They propose annotations to specifiy timing constraints in the code, and they also considered just single-threaded code. Nevertheless, the difference is that they adopt Esterel as input language, while we adopt the C language. They had to generate a timed automata model to verify the Esterel code using the Kronos model verifier, while our work remains on the implementation level.

The input of the related work analyzed are: Temporal Logic of Actions (TLA) specifications, Timed Transition Model + Real-Time Temporal Logic (TTM/RTTL), Duration Calculus (DC), Giotto programs, and Esterel programs. To the best of our present knowledge, there is no work to verify timing constraints directly in the actual C code without explicitly translating to a high level model.

VI. Conclusions and Future Work

Model checking is often used for finding errors rather than for proving that they do not exist [15]. However, model

checkers are capable of finding errors that are not likely to be found by simulation or test. The reason is that unlike simulators/testers, which examine a relatively small set of test cases, (bounded) model checkers consider all (up to the specified bounds) possible behaviors of the system. This paper described how to use an untimed software model checker to verify timing constraints in C code. In our proposed method we use the C language because it is one of the most common implementation language of embedded systems. As far as we are aware, there is no other approach that deals with model-checking 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 for verifying timing properties using ordinary model checkers. The main advantage of an explicit-time approach is the ability to use languages and tools not specially designed for real-time model checking. As pointed out by Lamport [10] "*the results reported that verifying explicit-time specifications with an ordinary model checker is not very much worse than using a real-time model checker*".

Experimental results have shown that the proposed method is promising, mainly because it is now possible to verify timing constraints in the C code. Hence, we are just following a movement towards the application of formal verification techniques to the implementation level. In this case, we avoid constructing models explicitly and go directly to code verification. As presented before, this method is particularly interesting when taking into account legacy code. However, we argue that our proposed method is not an alternative to methods currently available in the literature, but complementary. We also show that using our proposed method it is possible to investigate several scenarios.

Although we believe that most embedded systems are still single-threaded programs, as future work we plan to consider multi-threaded code, which is also supported by ESBMC, and to extend the code annotation method to consider fine-grained timing constraints by adding timing duration in individual instructions or blocks, and to add bounds for loops. In this way, more sophisticated annotation mechanisms will allow us to express context-dependent execution time bounds.

ACKNOWLEDGMENT

The authors acknowledge the support granted by FAPESP process 08/57870-9, CAPES process BEX-3586/10-3, and by CNPq processes 575696/2008-7, and 573963/2008-8. This research was conducted while the first author was visiting the University of Southampton.

REFERENCES

[1] B. Berthomieu and F. Vernadat, "Time petri nets analysis with tina," in *Int. Conf. on the Quantitative Evaluation of Systems*. IEEE Computer Society, 2006, pp. 123–124.

[2] N. S. Bjørner, A. Browne, M. A. Colón, B. Finkbeiner, Z. Manna, H. B. Sipma, and T. E. Uribe, "Verifying temporal properties of reactive systems: A STeP tutorial," *Formal Methods in System Design*, vol. 16, pp. 227–270, June 2000.

[3] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking: 10^{20} states and beyond," *Information and Computation*, vol. 98, pp. 142–170, June 1992.

[4] K. Y. Chun and D. V. Hung, "Verifying real-time systems using untimed model checking tools," The United Nations University. Tech. Report UNU-IIST 302, June 2004.

[5] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. The MIT Press, January 2000.

[6] L. Cordeiro and B. Fischer, "Verifying Multi-threaded Software using SMT-based Context-Bounded Model Checking," in *International Conference on Software Engineering (ICSE'2011)*. ACM/IEEE, May 21-28 2011, pp. 331–340.

[7] L. Cordeiro, B. Fischer, H. Chen, and J. Marques-Silva, "Semiformal verification of embedded software in medical devices considering stringent hardware constraints," in *Int. Conf. on Emb. Soft. and Systems (ICESS'09)*. pp. 396–403.

[8] P. Ganty and R. Majumdar, "Analyzing real-time event-driven programs," in *Int. Conf. Formal Modeling and Analysis Timed Systems (FORMATS'09)*, 2009, pp. 164–178.

[9] T. Henzinger, P.-H. Ho, and H. Wong-Toi, "Hytech: A model checker for hybrid systems," *Software Tools for Technology Transfer*, vol. 1, pp. 460–463, 1997.

[10] L. Lamport, "Real-time model checking is really simple," in *Correct Hardware Design and Verification Methods (CHARME'05)*, LNCS 3725, October, 3-6 2005, pp. 162–175.

[11] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a Nutshell," *Int. Journal on Software Tools for Technology Transfer*, vol. 1, no. 1–2, pp. 134–152, Oct. 1997.

[12] L. D. Moura and N. Bjrner, "Z3: An efficient smt solver," *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 4963, pp. 337–340, 2008.

[13] J. Ostroff and H. Ng, "Verifying real-time systems with standard theories," in *In AMAST Workshop on Real-Time Systems*, 2000.

[14] G. Rote, "Crossing the bridge at night," *EATCS Bulletin*, vol. 78, pp. 241–246, October 2002.

[15] B. Schlich and S. Kowalewski, "Model checking C source code for embedded systems," *Software Tools for Technology Transfer*, vol. 11, no. 3, pp. 187–202, June 2009.

[16] J. Sifakis, S. Tripakis, and S. Yovine, "Building models of real-time systems from application software," *In Proceedings of the IEEE*, vol. 91, pp. 100–111, January 2003.

[17] F. Wang, G.-D. Huang, and F. Yu, "TCTL inevitability analysis of dense-time systems: From theory to engineering," *IEEE Trans. Softw. Eng.*, vol. 32, pp. 510–526, July 2006.

[18] R. Wilhelm et. al., "The worst-case execution-time problem: overview of methods and survey of tools," *ACM Trans. Emb. Comp. Systems*, vol. 7, pp. 36:1–36:53, May 2008.

[19] S. Yovine, "Kronos: A Verification Tool for Real-Time Systems," *International Journal on Software Tools for Technology Transfer*, vol. 1, pp. 123–133, 1997.