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

Raimundo Barreto[1], Lucas Cordeiro[2], and Bernd Fischer[3]

[1] rbarreto@dcc.ufam.edu.br
Dept. of Computer Science, Federal University of Amazonas, Manaus AM, Brazil
[2] lucascordeiro@ufam.edu.br
Dept. of Elect.&Telecom., Federal University of Amazonas, Manaus AM, Brazil
[3] b.fischer@ecs.soton.ac.uk
Electronics and Computer Science, University of Southampton, Southampton, 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 for the implementation. Here, we use an untimed bounded model checker to verify timing properties of embedded C programs. We propose 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 checker such as ESBMC. Thus, 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.

1 Introduction

Model checking is an automatic technique for verifying finite state concurrent systems [9]. 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 net, or a kind of labeled state graphs, and verified with specialized timed model checking tools, such as TINA [1], HyTech [8], Kronos [16], or UPPAAL [11]. For example, UPPAAL uses timed automata as input and a fragment of the TCTL temporal logic [15] 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 ANSI-C programs. We assume that timing annotations are given externally, either be a WCET analysis of the code, or by a domain expert. We then translate such annotated C code automatically to 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 our implementation, we use ESBMC [5], 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. Moreover, symbolic model checking can easily be combined with powerful symbolic reasoning methods such as decision procedures and SMT solving. 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 thus to propose a method to check timing properties directly in the actual C code using a (conventional) software model checker; however, we can check timing properties as well as safety and liveness properties (see [5]). 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 focus on time-critical embedded systems software which, due to predictability issues, require guarantees that in all execution paths the timing constraints are met. We illustrate our approach through an industrial case study involving a medical device called pulse oximeter. Our experiments show that our technique can be used efficiently for verifying embedded real-time systems using an existing untimed model checker.

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

The paper is organized as follows. The next section shows the background to understand the proposed method. Section 3 describes the proposed method. Section 4 analyzes the pulse oximeter case study. Section 5 reviews related work. Finally, Section 6 summarizes the paper and explains future work.

2 Backgroud

2.1 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 [6, 5], although we have focused in single-threaded software here. ESBMC supports full ANSI-C, and can verify programs that make use of bit-level, 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.

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 safety property ϕ, a context bound C and a bound k, ESBMC builds a reachability tree (RT) that represents the program unfolding for C, k and ϕ. We then derive a VC ψ_k^π for each given interleaving (or computation path) $\pi = \{\nu_1, \ldots, \nu_k\}$ such that ψ_k^π is satisfiable if and only if ϕ has a counterexample of depth k that is exhibited by π. ψ_k^π is given by the following logical formula:

$$\psi_k^\pi = I(s_0) \wedge \bigvee_{i=0}^{k} \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1}) \wedge \neg\phi(s_i) \tag{1}$$

Here, I characterizes the set of initial states of M and $\gamma(s_j, s_{j+1})$ is 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 executions of M of length i and ψ_k^π can be satisfied if and only if for some $i \leq k$ there exists a reachable state along π at time step i in which ϕ is violated. ψ_k^π is a quantifier-free formula in a decidable subset of first-order logic, which is checked for satisfiability by an SMT solver. If ψ_k^π is satisfiable, then ϕ is violated along π and 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^π is unsatisfiable, we can conclude that no error state is reachable in k steps or less along π. Finally, we can define $\psi_k = \bigwedge_\pi \psi_k^\pi$ and use this to check all paths.

2.2 Model Checking Real-Time Systems

Model checking is a verification technique that applies to systems that can be modeled by a mathematical formalism (finite automata and Petri nets are examples). In practice, the size of the systems is really the main obstacle to overcome.

Therefore, model checker users usually simplify the model under analysis. Model checking consists in three steps. (1) Mathematical representation (modeling). Usually such models represents states and transitions, and may be composed by and synchonized by several components. (2) Representation of a property by a logical formula. Once the model is built, we formally state the properties to be checked, usually in a temporal logic. (3) Model checking algorithm. Given a model $\mathbf{A}$ and a property ϕ, a model checking algorithm answers the question: "does the model $\mathbf{A}$ satisfy the property ϕ?"

There are several tools that model check real-time systems.

TSMV [12] is a symbolic model checker that verifies TCTL formulas on Timed Kripke Structures (TKS), i.e. finite state graphs where transitions carry a duration. The main feature of TKS's is that the durations of transitions are atomic, that is, when moving from state s to state s' in a step that lasts 10 time units, there is no intermediary configuration between s at time t and s' at time $t + 10$. The key motivation for this semantics is that it leads to simple and efficient model checking algorithms.

UPPAAL [11] allows one to analyze networks of timed automata with binary synchronization. It contains three main parts: (i) a graphical editor where timed process are described; (ii) a simulator where it is possible to choose a sequence of transitions, and to see the behavior of the system; and (iii) a verifier of reachability properties. The main drawbacks of UPPAAL are: (1) the binary synchronization is a bit restrictive and requires one to use ad hoc mechanisms to describe other kinds of synchronizations (e.g., broadcast); (2) the specification language considers only reachability properties and not a full temporal logic. This entails that it is necessary to include a observer automata to express complex properties.

KRONOS [16] is a model checker that can decide whether some property, expressed by a TCTL formula, holds for a timed automaton (also called timed graph), given in textual form. It allows one to verify liveness properties, and is not restricted to reachability properties. Even though KRONOS contains no graphical nor simulation modes, it is a true timed model checker. However, under its current form, it is mostly intended for advanced users with a good knowledge of formal methods.

HYTECH [8] receives a set of linear hybrid automata, and synchronizes them by some common transitions. From the automata in a textual form, HYTECH can compute subsets of the global state space. HYTECH also handles parametric analysis, that is, when a system (or a property) contains parameters, the analysis can provide the parameter values for which the property holds. The drawbacks of HYTECH is that it includes no simulation mode; model checking does not apply to a temporal logic: the user has to build himself the subset of states to be computed by combinations of basic constraints.

TINA [1] is a toolbox for the edition and analysis of Petri Nets and Time Petri Nets. The Tina toolbox includes several tools such as: (i) an editor for graphically or textually described Time Petri nets; (ii) construction of reachability graphs where it may build coverability graphs, persistent sets, state class

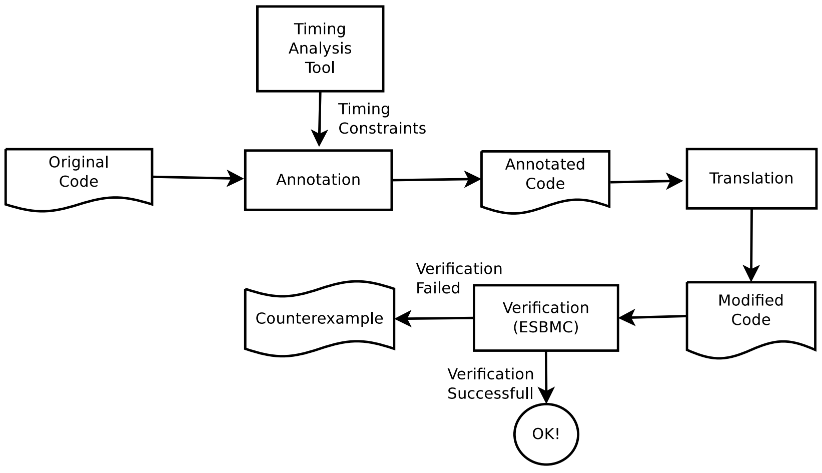

Fig. 1. Overview of the Proposed Method

graphs, and (iii) a state/event LTL model checker that checks reachability properties. However, it is difficult to check real-time quantitative properties.

3 Proposed Method

This section describes the method proposed to verify timing properties on single-threaded C code using a bounded model model-checker. Figure 1 gives an overview of the approach. It is divided into four phases. 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. Basically, this translation consists in (i) adding declarations of the timer variables; (ii) gathering the annotated timing constraints information and including assignment statments for the new added variables; (iii) adding (user-defined) assert statements at specific points of the program code. The third step is to check the translated code with the ESBMC model checker. Finally, the last phase, evaluates ESBMC's results. As shown, the way to check timing properties is by using assertion.

3.1 Timed Programming Model

The proposed method aims to pragmatically assist developers in the specification and analysis of timing constraints in C code. 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 timing assertions on timer variables to check timing properties; and

(iv) to introduce timer variable *reset* to restart the timer counting. 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, let consider that the semantics of a sequential program $\mathcal{P}$ is 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;
- $\mathcal{V} = \langle v_1, v_2, \cdots, v_z \rangle$ is a finite list 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 $\mathcal{V}$;
- $\rightarrow \subseteq \mathcal{S} \times \mathcal{F} \times \mathcal{S}$ is a finite set of labeled transitions, such that a state transition in $\langle s_i, f_\phi, s_j \rangle$, is represented by $s_i \xrightarrow{f_\phi} s_j$, $\forall s_i, s_j \in S, i \neq j, f_\phi \in \mathcal{F}$ and it is supposed that each function f_ϕ is executed to completion.

Let $\pi[n \ldots m]$ for $0 \leq n < m \in \mathbb{N}$ be an *execution path* denoted by a finite sequence $s_n \xrightarrow{f_{\phi_1}} s_{n+1} \xrightarrow{f_{\phi_2}} \cdots \xrightarrow{f_{\phi_q}} s_m$ 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 $\pi[0..5] = s_0 \xrightarrow{f_1} s_1 \xrightarrow{f_3} s_2 \xrightarrow{f_5} s_3 \xrightarrow{f_2} s_4 \xrightarrow{f_4} s_5$.

In order to introduce timing constraints into the program, we change the original program $\mathcal{P}$ to another program $\mathcal{P}' = \langle \mathcal{S}, s_0, \mathcal{V}', \mathcal{F}', \rightarrow \rangle$ where:

- $\mathcal{V}^{'} = \texttt{cat}(\mathcal{V}, \mathcal{T})$, where `cat` means list concatenation in this case used to concatenate lists of variables;
- $\mathcal{F}^{'} = \mathcal{F} \cup \mathcal{A} \cup \mathcal{R}$;
- $\mathcal{T} = \langle t_1, t_2, \cdots, t_p \rangle$ is a finite list of timer variables;
- $\mathcal{A} = \{a_1(t_{k_1}), a_2(t_{k_2}), \cdots, a_n(t_{k_x}) \mid a_i$ is a special function that asserts on timer variables, $1 \leq i \leq n$, and $t_{k_d} \in \mathcal{T}, 1 \leq d \leq x, 1 \leq k_d \leq p, p = |\mathcal{T}|\}$;
- $\mathcal{R} = \{r_1(t_1), r_2(t_2), \cdots, r_p(t_p) \mid r_w$ is a special function that resets a timer variable, $1 \leq w \leq p$, $p = |\mathcal{T}|$, and $t_w \in \mathcal{T}\}$.

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 $D(\pi[n..m]) = \sum_{i=1}^{m-n} D(f_{\phi_i})$ 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 $\pi[0..11] = s_0 \xrightarrow{r_1(t_1)} s_1 \xrightarrow{r_2(t_2)} s_2 \xrightarrow{f_1} s_3 \xrightarrow{f_3} s_4 \xrightarrow{a_1(t_1)} s_5 \xrightarrow{r_1(t_1)} s_6 \xrightarrow{f_5} s_7 \xrightarrow{f_2} s_8 \xrightarrow{a_2(t_1)} s_9 \xrightarrow{f_4} s_{10} \xrightarrow{a_3(t_2)} s_{11}$, where $f_{\phi_1} = r_1(t_1)$; $f_{\phi_2} = r_2(t_21)$; $f_{\phi_3} = f_1$; $f_{\phi_4} = f_3$; $f_{\phi_5} = a_1(t_1)$; $f_{\phi_6} = r_1(t_1)$; $f_{\phi_7} = f_5$; $f_{\phi_8} = f_2$; $f_{\phi_9} = a_2(t_1)$; $f_{\phi_{10}} = f_4$; $f_{\phi_{11}} = a_3(t_1)$. We can conclude that $D(\pi[0..11]) = \sum_{i=1}^{11} D(f_{\phi_i}) = D(f_1) + D(f_3) + D(f_5) + D(f_2) + D(f_4)$. As we can see, in the execution path $\pi[0..11]$ we have three timing verifications: $a_1(t_1)$, $a_2(t_1)$, and $a_3(t_2)$; and three timer resets: $r_1(t_1), r_2(t_2)$, and $r_1(t_1)$.

3.2 Annotation of Timing Constraints

The inclusion of timing constraints in the source code is particularly interesting since it can automatically be checked as the program are being 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. Thus, we rely on a timing analyzer tool to predict worst-case timing bounds, for instance [3]. Using this annotation we can add the function D to the code.

Figure 2(a) shows an example of code annotation from the example shown on Section 3.1. 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 Figure 2, the TIMER1 is checking local timing constraints. Firstly, this timer verifies timing constraint related to functions $f_1()$ and $f_2()$. Later, this same timer is then used to verify timing constraint 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 sysem, i.e., the function calls from $f_1()$ up to $f_5()$.

In this paper we are just showing a coarse-grained timing constraint resolution in the level of functions. Therefore, we show only how to specifiy timing constraints in the source code on functions. However, as ongoing work, we are extending the proposed annotation method to consider fine-grained (in the level of instructions) timing constraints.

3.3 Translation and Verification

The translation consists in looking for comments that start by `//@` and treat them appropriately. The translation of the code shown on Figure 2(a) can be seen in Figure 2(b). This translation is carried out automatically by a specific tool[4]. 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,

[4] This tool is available at http://esbmc.org

```
//@ 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=0;

//@ RESET-TIMER TIMER2=0;

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

//@ RESET-TIMER TIMER1=0;

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=0;
TIMER1 = 0;
// RESET-TIMER TIMER2=0;
TIMER2 = 0;
f1(); f2();
// ASSERT-TIMER (TIMER1 <= alpha);
assert (TIMER1 <= alpha);
// RESET-TIMER TIMER1=0;
TIMER1 = 0;
f3(); f4();
// ASSERT-TIMER (TIMER1 <= beta);
assert (TIMER1 <= beta);
f5();
// ASSERT-TIMER (TIMER2 <= gamma);
assert (TIMER2 <= gamma);
...
```
(b)

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

memory leaks, etc), and then run the model check to find for timing violations in the *modified* code.

After translation, this new code is able to be run on ESBMC, which check properties using user-specified assert statement. In the proposed method the assert will be the way to check timing properties. In the code of Figure 2(b) we may see three timing verification.

3.4 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 correct solution in this case is to let P3 and P4 cross in the middle. Hence, 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 this problem[5] and submitted it the ESBMC model checker. We first verified that 60 is indeed the optimal solution, i.e., that the elapsed time cannot be less than 60. The timing assertion and the ESBMC's output can be seen in Figure 3. We can see that the verification failed, which means that ESBMC find *at least* one execution path where the asserted condition is false. The ESBMC spent 3m25s to give the result. The second attempt was to check if the time duration could be greater or equal to 60. The result was SUCCESSFULL, which means that the SMT solver found that in all execution paths the assert condition is true. With this two results, we may conclude that $time = 60$ is indeed the optimal solution. The ESBMC spent 16m28s to give the result. These experiments were conducted on an Intel Pentium Dual CPU with 4 GB of RAM running Linux OS, and ESBMC v.1.15.1 (64bits).

```
//@ ASSERT-TIMER (__timing__ < 60);
assert (__timing__ < 60);
...
size of program expression: 37084 assignments
Generated 1 VCC(s), 1 remaining after simplification
Encoding remaining VCC(s) using bit-vector arithmetic
Solving with SMT Solver Z3 v2.16
Runtime decision procedure: 177.843s
Building error trace
Counterexample:
...
Violated property:
  file __bridge__LT60.c line 151 function main
  assertion
  __timing__ < 60

VERIFICATION FAILED
```

Fig. 3. Verification failed as the result of the application of the ESBMC

4 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 in the verification of timing constraints. All experiments were conducted on an otherwise idle Intel Pentium Dual CPU with 4 GB of RAM running Linux OS. We chose ESBMC v.1.15.1-64bits as untimed bounded model checker.

[5] The code, counterexample and explanation on the results may be downloaded at http://esbmc.org

4.1 Problem Specification

The pulse oximeter is responsible for measuring the oxygen saturation (SpO_2) and heart rate (HR) in the blood system using a non-invasive method. This device was used as case study in [6] to raise the coverage of tests in embedded system combining hardware and software components. The implementation is relatively complex, since the final version has approximately 3500 lines of ANSI-C code and 80 functions. Considering that such paper does not checked timing constraints explicitly, and the implementation is publicly available[6], we used this problem as a case study for our proposed method.

The architecture consists in four components: sensor, data acquisition module (OEM-III)[7], microcontroller, and LCD display. The sensor captures data on oxygen saturation (SpO2) and heart rate (HR) of the patient. The OEM III module has an interface for communication with 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 serial port, treat them and displays on the LCD.

The packet description of the data format is shown in Table 1. A frame consists of 5 bytes; and a packet consists of 25 frames. Three packets (375 bytes) are transmitted each second. In this table, Byte1 is always “01” (usually used for synchroization); Byte2 is the status byte (if sensor is not connected, for instance); Byte3 shows the 8-Bit Plethysmographic Pulse Amplitude; Byte4 presents HR and SpO2 data; and Byte5 is the checksum.

Table 1. 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

In the context of timing constraints, the following functional requirements are considered:

FR1. The system has to read all HR and SpO2 data in at most 1 second. In this case, we have to take into account the maximum frequency of the serial communication (9600bps), and the amount of bytes sent by the sensor device.

[6] Availabe at: http://esbmc.org

[7] For more information refer to www.nonin.com/OEMSolutions/OEM-III-Module

FR2. The software must check whether the frames sent by the sensor is correct, and show in the LCD if found any problem. This implies in verification of the checksum, and status bytes. Besides that, we have to show any problem in the LCD display.

FR3. The user should be able to see, every second, the data of heart rate and oxygen saturation in the patient's blood. Therefore, we have to store patient's information and to show in the LCD display.

FR4. The system must allow users to store data on HR and SpO2 in the external memory of the microcontroller. We have to consider the the amount of data and the time to store in the external memory.

4.2 Code Annotation

The timing constraints for this project is shown in Table 2. These constraints come from either the specification or a domain specialist. As presented before (see Section 3.2), these timing constraints are annotated into the code. It is worth noting that if one function calls another function, the timing constraint may be specified on the caller function, or on the called function, or both.

Table 2. 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 microcontroller	500

4.3 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 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 f_1 (`receiveSensorData`);
2. for each byte read:
   (a) to check status of the second byte of each frame by calling function f_2 (`checkStatus`); if there is an error, it should be called the function f_3 (`printStatusError`);
   (b) to check the fifth byte of each frame by calling function f_4 (`checkSum`); if there is an error, it should be called the function f_5 (`printCheckSumError`);
   (c) to read the fourth byte of first frame and to call function f_6 (`storeHRMSB`);
   (d) to read the fourth byte of second frame and to call function f_7 (`storeHRLSB`);
   (e) to read the fourth byte of third frame and to call function f_8 (`storeSpO2`);
3. call the functions f_9 (`average_HR`), f_{10} (`average_SpO2`), f_{11} (`getHR`), f_{12} (`getSpO2`), f_{13} (`printHR`) with HR value as argument, f_{14} (printSpO2) with SpO2 value as argument, f_{15} (insertLog) with HR value as argument, and f_{15} (insertLog) with SpO2 value as argument.

Table 3. Experimental Results

ID	% Checksum Error	Time(s)	Result
1	0%	28.9	successful
2	16.6%	20.3	successful
3	20%	20.2	successful
4	25%	19.9	successful
5	33.3%	19.9	successful
6	50%	21.1	failed
7	100%	30.2	failed

Figure 4 depicts part of the pulse oximeter code submitted to the ESBMC. Table 3 shows the experimentl results. We experimented seven scenarios taking into account the percentage of checksum errors. The percentage considered was 0%, 16.6%, 20%, 25%, 33.3%, 50%, and 100%. Excepting the best scenario (0%)

```
...
// 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);
  ...
}
```

Fig. 4. Code for running in the ESBMC model-checker

and worst scenario (100%), all timing performance was 20s in average. As presented in Table 3, if considered the worst-case scenario, in this case 100% of data error, timing constraints are not met. However, if it is considered that it is not practical to consider an extreme situation like this one, we may conclude that up to 33.3% of checksum errors, the system will continue to reach the timing constraints.

5 Related Work

Lamport [10] advocates that most real-time specifications can be verified using existing languages and methods. He proposed to represent time as an ordinary 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 call this method as model checking explicit-time specifications. He 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 the TLA+ may be high.

Ostroff and Hg [13] presented a framework that allows specification, development and verification of discrete real-time properties of reactive systems. This framework considers a Timed Transition Model (TTM) as underlying computational model, and Real-Time Temporal Logic (RTTL) as the requirements specification language. The authors provide a conversion procedure for mapping 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 state exploration for checking real-time systems properties. One problem of this method is that the converted clocked formulas count the number of tick events that occur. Thus, 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), and using 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 SPIN untimed model checking language (in this case Promela). They applied their method in the Biphase Mark Protocol. However, they do not show how translate from automata model to Promela language. Additionaly, it is not clear what timing constraint was verified in this case study.

Ganty and Majumdar [7] 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. The result is undecidable because such programs are not finite-state. In this case, the task buffer as well as the call stack can grow unboundedly large in the course of the execution. They suggest to use higher-level languages, such as Giotto, which statically restricts the ability to post tasks arbitrarily, 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.

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), and Giotto programs. To the best of our present knowledge, there is no work that verify timing constraints using C code as input language.

6 Conclusions and Future Work

Model checking is often used for finding errors rather than for proving that they do not exist. However, model checkers are capable of finding errors that are not likely to be found by simulation or test. The reason for this is that unlike simulators/testers, which examine a relatively small set of test cases, model checkers consider all possible behaviors of the system.

This paper described how to use an untimed software model checker to verify timing constraints in C program language. In our proposed method we use the C language because it is one of the most commom implementation language of embedded systems. As far as we are aware, there are only few approaches that deal with model-checking timing constraints using C code as input language. 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 promissing, mainly because now it is possible to verify timing constraints in the C code. Therefore, we are just following a movement toward 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.

This paper just considered single-threaded code. It is an ongoing work to consider multi-threaded code, which is also supported by ESBMC. Therefore, we need to consider interleavings and priorities.

This work proposed just a coarse-grained timing constraint resolution, in this case, we considered just timing constraints in functions. Thus, one future work is to extend both the code annotation method and the timing verification to consider fine-grained timing constraints, maybe in critical sections, for instance to add timing duration between two instructions representing a sequential block, and timing bounds for loops.

Acknowledgements. 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.

References

1. Bernard Berthomieu and Francois Vernadat. Time petri nets analysis with tina. In *Proceedings of the 3rd international conference on the Quantitative Evaluation of Systems*, pages 123–124, Washington, DC, USA, 2006. IEEE Computer Society.
2. Nikolaj S. Bjrner, Anca Browne, Michael A. Colon, Bernd Finkbeiner, Zohar Manna, Henny B. Sipma, and Tomas E. Uribe. Verifying temporal properties of reactive systems: A step tutorial. In *FORMAL METHODS IN SYSTEM DESIGN*, page 2000. Springer, 2000.
3. Stefan Bygde, Andreas Ermedahl, and Björn Lisper. An efficient algorithm for parametric wcet calculation. In *Proceedings of the 2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '09, pages 13–21, Washington, DC, USA, 2009. IEEE Computer Society.
4. Kim Yong Chun and Dang Van Hung. Verifying real-time systems using untimed model checking tools. Technical Report UNU-IIST Report 302, The United Nations University. International Institute for Software Technology, June 2004.
5. Lucas Cordeiro and Bernd Fischer. Verifying Multi-threaded Software using SMT-based Context-Bounded Model Checking. In *International Conference on Software Engineering (ICSE'2011)*, pages 331–340, Waikiki, Hawaii, May 21-28 2011. ACM/IEEE.
6. Lucas Cordeiro, Bernd Fischer, Huan Chen, and Joao Marques-Silva. Semiformal verification of embedded software in medical devices considering stringent hardware constraints. In *International Conference on Embedded Software and Systems (ICESS'09)*, pages 396–403, Washington, DC, USA, 2009. IEEE Computer Society.
7. Pierre Ganty and Rupak Majumdar. Analyzing real-time event-driven programs. In *Proceedings of the 7th International Conference on Formal Modeling and Analysis of Timed Systems*, FORMATS '09, pages 164–178, Berlin, Heidelberg, 2009. Springer-Verlag.
8. Thomas Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. Hytech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:460–463, 1997.
9. Edmund Clarke Jr, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, January 2000.
10. Leslie Lamport. Real-time model checking is really simple. In *Correct Hardware Design and Verification Methods (CHARME'05)*, volume 3725 of *Lecture Notes in Computer Science*, pages 162–175, Saarbrücken, Germany, October, 3-6 2005.
11. Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.
12. N. Markey and Ph. Schnoebelen. Symbolic Model Checking of Simply-Timed Systems. Technical Report LSV-03-14, Lab. Spécification et Vérification. Ecole Normale Supérieure de Cachan, October 2003.
13. Jonathan Ostroff and Hak Ng. Verifying real-time systems with standard theories. In *In AMAST Workshop on Real-Time Systems*, 2000.
14. Günter Rote. Crossing the bridge at night. In *EATCS Bulletin*, volume 78, pages 241–246, October 2002.
15. Farn Wang, Geng-Dian Huang, and Fang Yu. TCTL inevitability analysis of dense-time systems: From theory to engineering. *IEEE Trans. Softw. Eng.*, 32:510–526, July 2006.
16. Sergio Yovine. Kronos: A Verification Tool for Real-Time Systems. *International Journal on Software Tools for Technology Transfer*, 1:123–133, 1997.