University of Pennsylvania
ScholarlyCommons

Departmental Papers (CIS) | Department of Computer & Information Science

8-23-2010

A Safety-Assured Development Approach for Real-Time Software

Eunkyoung Jee
University of Pennsylvania, eunkjee@seas.upenn.edu

Shaohui Wang
University of Pennsylvania, shaohui@seas.upenn.edu

Jeong Ki Kim
University of Pennsylvania, jeongki@seas.upenn.edu

Jaewoo Lee
University of Pennsylvania, jaewoo@seas.upenn.edu

Oleg Sokolsky
University of Pennsylvania, sokolsky@cis.upenn.edu

See next page for additional authors

"This work has been submitted to the IEEE for possible publication. Copyright may be transferred without notice, after which this version may no longer be accessible."
The 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), to appear, Aug. 23-25, 2010.

This paper is posted at ScholarlyCommons. http://repository.upenn.edu/cis_papers/430
For more information, please contact repository@pobox.upenn.edu.

A Safety-Assured Development Approach for Real-Time Software

Abstract
Guaranteeing timing properties is an important issue as we develop safety-critical real-time systems such as cardiac pacemakers. We present a safety assured development approach of real-time software using a pacemaker as our case study. Following the model-driven development techniques, measurement-based timing analysis is used to guarantee timing properties in implementation as well as in the formal model. Formal specification with timed automata is checked with respect to timing properties by model checking technique and is transformed into implementation systematically. When timing properties may be violated in the implementation due to timing delay, it is suggested to measure the time deviation and reflect it to the code explicitly by modifying guards. The model is altered according to the modifications in the code. These changes of the code and the model are considered safe if all the properties are still satisfied by the modified model in re-performed model hecking. We demonstrate how the suggested approach can be applied to single-threaded and multi-threaded versions of implementation. This approach can provide developers with a useful time-guaranteeing technique applicable to several code generation schemes without imposing many restrictions.

Keywords
real-time software, timed automata, formal verification, code generation, timing analysis

Disciplines
Computer Sciences

Comments
"This work has been submitted to the IEEE for possible publication. Copyright may be transferred without notice, after which this version may no longer be accessible."

The 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), to appear, Aug. 23-25, 2010.

Author(s)
Eunkyoung Jee, Shaohui Wang, Jeong Ki Kim, Jaewoo Lee, Oleg Sokolsky, and Insup Lee

This conference paper is available at ScholarlyCommons: http://repository.upenn.edu/cis_papers/430

A Safety-Assured Development Approach for Real-Time Software

Eunkyoung Jee, Shaohui Wang, Jeong Ki Kim, Jaewoo Lee, Oleg Sokolsky, Insup Lee
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, USA
Email: {eunkjee, shaohui, jeongki, jaewoo}@seas.upenn.edu, {sokolsky, lee}@cis.upenn.edu

***Abstract*—Guaranteeing timing properties is an important issue as we develop safety-critical real-time systems such as cardiac pacemakers. We present a safety assured development approach of real-time software using a pacemaker as our case study. Following the model-driven development techniques, measurement-based timing analysis is used to guarantee timing properties in implementation as well as in the formal model. Formal specification with timed automata is checked with respect to timing properties by model checking technique and is transformed into implementation systematically. When timing properties may be violated in the implementation due to timing delay, it is suggested to measure the time deviation and reflect it to the code explicitly by modifying guards. The model is altered according to the modifications in the code. These changes of the code and the model are considered safe if all the properties are still satisfied by the modified model in re-performed model checking. We demonstrate how the suggested approach can be applied to single-threaded and multi-threaded versions of implementation. This approach can provide developers with a useful time-guaranteeing technique applicable to several code generation schemes without imposing many restrictions.**

***Keywords*-real-time software; timed automata; formal verification; code generation; timing analysis;**

I. Introduction

When we develop a real-time system, guaranteeing timing properties on its implementation is an important but non-trivial issue. It becomes essential if the real-time system is a safety-critical one in which violation of timing properties can result in fatal loss of properties or life.

Focusing on how to implement time-guaranteed real-time software from the model systematically, we propose a safety-assured development approach of real-time software. We demonstrate the proposed approach using cardiac pacemaker software, representative of life-critical real-time systems in which many complex timing constraints are imposed. This work also was motivated from the Pacemaker Grand Challenge, the first certification challenge problem issued by the Software Certification Consortium (SCC) [1].

Several concepts and approaches can be effectively integrated to contribute to the development of safety-assured real-time software. We basically follow the model-driven development (MDD) concept which is considered a promising methodology and is applied in the development of embedded software. According to the MDD concept, we create a formal model of the real-time system, verify the model, and generate an implementation code from the model. In order to check and guarantee timing constraints on the implementation, we perform measurement-based timing analysis on the implementation and revise the implementation and the model according to the timing analysis result.

Formal methods have been used as a promising approach to assure software quality. We start the proposed safety-assured development approach by taking the software specification and creating a timed automata model capturing the requirements. The model is formally verified by the model checking technique using the real-time model checker UPPAAL [2] with respect to properties extracted from requirements specification.

After obtaining a timed automata model which has been proven correct, an implementation code is synthesized from the timed automata. Code generation from timed automata has been studied in [3]–[5]. In [3], the TIMES tool generates executable C code for the Lego Mindstorms™ [6] from timed automata extended with tasks. The code is generated assuming the synchrony hypothesis which says that the underlying machine is infinitely fast and the reaction of the system to an input event is instantaneous. The assumption of synchrony hypothesis is helpful in simplifying the behavioral specifications of reactive systems, but unrealistic. An implementation can only react within a non-zero reaction delay while timed automata is assumed to react instantaneously to events and time-outs according to the classical semantics of timed automata. The properties proved on the model are not guaranteed to be preserved by the code generation scheme of [3].

In [4], authors presented a tool set which enables automatic generation of provably correct code from verified timed automata models. It is guaranteed that the verified timing properties in the model with the almost ASAP semantics [7] are preserved in the generated code without assuming the synchrony hypothesis, under the condition that $\delta < 3\Delta_L + 4\Delta_P$ where δ, Δ_L, Δ_P represents the reaction delay of the controller, time length of an execution loop, and time between two clock ticks, respectively. Their approach and tool set is quite promising, but obtaining benefits of preservation of timing properties requires many

assumptions be fulfilled ahead. For example, no shared continuous variables between automata are allowed in their approach. Shared variables should be modeled by discrete events because only events are visible by multiple automata.

While we try to answer how preservation of timing properties can be achieved in the implementation with minor changes of the code even when the original implementation is based on assumption of synchrony hypothesis, we propose a framework for time-guaranteed code generation applicable to any systematic code generation scheme in which concrete relation between the timed automata model and the generated code can be identified. Our focus is neither proposing a new code generation scheme from timed automata nor providing complete guarantees of all types of timing properties. We provide developers with a useful time-guaranteeing technique applicable to any systematic code generation approaches without imposing many restrictions.

We implement the pacemaker software by synthesizing code from the timed automata assuming the synchrony hypothesis. Once the code is obtained, we check to see if timing properties hold on the code by inserting instrumentation code. Timing properties requiring zero reaction delay are usually violated by the code although they are satisfied by the model. For the violated properties, we profile time-consuming operations like reading a clock and wating/posting a semaphore as well as execution of our implementation, and then determine a Δ which we will call *timing tolerance* in this paper. The Δ is used to revise the code by enlarging guards to ensure that the revised code satisfies the previously violated properties.

Then, developers check whether the revised code is safe by making corresponding changes back to the model and performing model checking again with respect to the properties. If all the properties are still satisfied on the modified model, the code including the timing tolerance is considered to guarantee timing properties safely. When the modified model fails to meet all the properties, analysis of the violated cases can provide developers with highly useful information with respect to property preservation. For example, guaranteeing one property may conflict with guaranteeing another property; here, the code may not guarantee timing properties even with enlarged guards.

We experiment with two different code generation schemes, one based on a single-threaded structure and the other based on a multi-threaded structure, to see how the proposed approach can work with different code generation structures.

The main contribution of this paper is the methodology for development of safety-assured real-time software. In order to preserve timing properties transferred from the verified model to the implementation, the model-driven development process is complemented by measurement-based timing analysis, revision of implementation and modeling with timing tolerances, and revisting model checking. We illustrate that this methodology can be effectively used for the development of time-critical software with a pacemaker case study.

The remainder of the paper is organized as follows: Section II explains the background of the case study. Section III presents the overview of the proposed process and demonstrates formal modeling and verification of the pacemaker software. Section IV explains code generation, timing analysis on the code, and the re-checking process. We discuss related issues in Section V and present a review of previous works related to topics addressed in this paper in Section VI. We conclude the paper in Section VII.

II. Case Study: Pacemaker Software

A. Heart

A human heart has four chambers: right and left atria, and right and left ventricles. De-oxygenated blood from the body is collected in the right atrium and then pumped into the lungs via the right ventricle. In the lungs, carbon dioxide in the blood is replaced with oxygen. This oxygenated blood then passes through the left atrium and enters the left ventricle, which pumps it out to the rest of the body.

From an electrical point of view, the heart is a pump made up of muscle tissue, controlled by an intrinsic electrical system. An electrical stimulus generated periodically (normally about 60-100 times per minute) by the sinus node, located in the right atrium, travels through the conduction pathways and causes the heart's chambers to contract and pump out blood. The atria are stimulated and contract shortly before the ventricles are stimulated and contract.

Under some conditions, this intrinsic cardiac system does not work properly and the heart rate becomes overly fast or slow, or irregular. In these situations, the body may not receive enough blood, which causes several symptoms such as low blood pressure, weakness, and fatigue. To avoid these symptoms, a pacemaker can be used to regulate the heartbeat [8].

B. Pacemaker

A cardiac pacemaker is an electronic device implanted into the body to regulate the heart beat by delivering electrical stimuli over leads with electrodes that are in contact with the heart. These stimuli are called *paces*. The pacemaker may also detect natural cardiac stimulations, called *senses*. We refer to cardiac paces and senses collectively as *events*.

A pacemaker must satisfy three fundamental medical requirements: the rate at which the cardiac chambers contract must not be too high; the rate at which the cardiac chambers contract must not be too low; the ventricles must contract at a particular interval after the atria contract. These general requirements are concretized by setting specific values or ranges to configurable parameters for the pacemaker.

In this paper, we are concerned with non-rate-adaptive operating modes of a pacemaker; that is, those modes which

do not change their pulse rate depending on the natural rate of the patient. There are ten such modes, each associated with a set of parameters that can be configured by a physician. Each of these modes is identified using a three-letter acronym. The first letter refers to the chambers of the heart that will be paced by the pacemaker. This letter may be V, A, D, or O representing Ventricle, Atrium, Dual (both chambers), or None, respectively. The second letter refers to the chambers of the heart that will be sensed by the pacemaker. As with the first letter, this may be V, A, D, or O. The third letter describes the pacemaker's response to sensing. This may be T, I, D, or O representing Triggers Pacing, Inhibits Pacing, Dual (T + I), or None, respectively. For example, a pacemaker in VVI mode paces and senses in the ventricle; pacing is inhibited when the pacemaker gets sensing. A pacemaker in DDD mode paces and senses in both the atrium and the ventricle; pacing is inhibited in the atrial channel by sensed ventricular or atrial activity and is inhibited in the ventricular channel by ventricular activity but triggered by sensing atrial activity [9].

C. VVI Mode Pacemaker

Among ten non-rate-adaptive operating modes, we consider a pacemaker software in VVI mode which is simple but useful to help understand and analyze timing constraints. VVI mode pacemaker senses spontaneous ventricular signals from a human heart. It can deliver electrical stimuli or ventricular paces over leads implanted in a patient heart. A ventricular sensing signal results in inhibiting a scheduled ventricular pacing signal. VVI mode pacemaker has an internal clock or lower rate timing cycle that begins with a paced or sensed ventricular event. The initial portion of the cycle consists of the ventricular refractory period (VRP), usually 200-350 ms. VRP is a period following each ventricular pace during which ventricular sensing is disabled to prevent a sudden pacing caused by an unexpected sense. The pacemaker cannot sense any signals during VRP. More specifically, any signal during the VRP cannot initiate a new lower rate interval (LRI) which is the maximum amount of time between two consecutive events in one chamber. Beyond the VRP, a sensed ventricular event inhibits the pacemaker and resets the LRI so that the timing clock returns to the baseline. A new pacing cycle is initiated and if no event is sensed, the timing cycle ends with the release of a ventricular stimulus according to the LRI. When hysteresis pacing mode is on (in conjunction with VVI mode), the pacemaker will give the human heart a chance of resuming to continuous normal operation by setting LRI to a larger value, namely the hysteresis rate interval (HRI).

III. Formal Modeling and Verification

A. Overall Process

We propose a safety-assured development process for real-time software. The proposed process follows the model-driven development approach with the emphasis on how to ensure that the implementation satisfies timing properties which has been satisfied in model. Figure 1 shows the overall process.

During the requirements and design phases of the software life cycle, developers first start from formal modeling with timed automata of the real-time software. Second, model checking is performed on the timed automata model with respect to desired properties using a real-time model checker such as UPPAAL. We focus on safety properties, especially timing properties which require that a certain event should happen no later than a specific delay. Given a formal model proven correct with respect to the properties, an implementation code is synthesized in the third step.

In the fourth step, we check to see if the same properties checked on the model are still satisfied by the code running on a target platform. If some timing properties are not satisfied by the code, we measure how much actual time deviates from the expected. During the fourth step, we find a timing tolerance value, Δ, through the measurement-based timing analysis. Guards in the code become relaxed with this Δ to make timing properties satisfied by the code. Once it is confirmed that the code satisfies the desired timing properties with the Δ, changes of the code, i.e., relaxed guards with the Δ, are reflected to the model in the fifth step. If the modified model still satisfies all the properties, the overall process ends. Otherwise, the process is repeated by revising the problematic model and the code. We will describe each step with the pacemaker example in the following subsections.

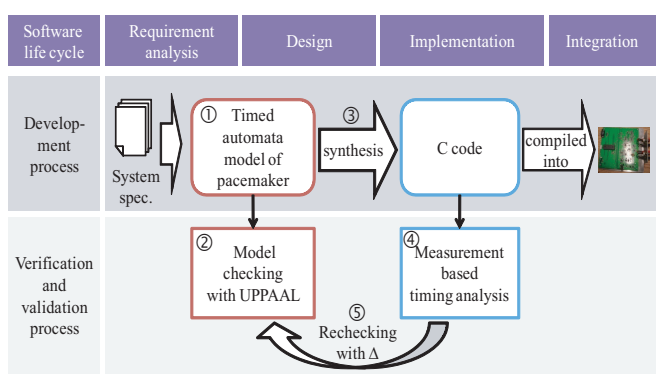
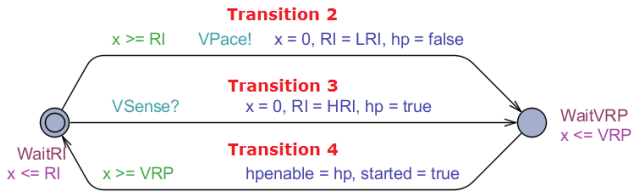


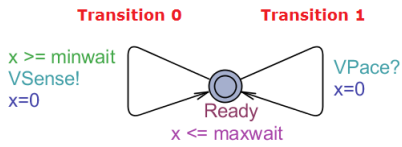
Figure 1. Overall process of a safety-assured development for a real-time pacemaker software

B. Formal Modeling

We used the Boston Scientific's system specification for a pacemaker [10]. Because timing constraints are so prevalent in the specification of the pacemaker, it is intuitive and straightforward to use timed automata [11] as our modeling language. Here we use the UPPAAL tool [2] to specify a timed automata model of the pacemaker in VVI mode.



(a) Ventricle controller model



(b) Heart model

Figure 2. Uppaal model for a pacemaker in VVI mode

We extracted properties to be satisfied by the VVI mode pacemaker from the system specification. LRI, HRI, and VRP are considered most important timing periods which should be guaranteed by the VVI mode pacemaker. Figure 2 shows two automata for Ventricle and Heart, representing a ventricle controller of the VVI mode pacemaker and a heart model as an environment, respectively.

1) Ventricle Model: Ventricle automaton shown in Figure 2(a) captures sensing signals from the ventricle of the heart and emitting ventricular pacing signals to the heart. Three important timing periods LRI, HRI, and VRP were captured in this automaton. The constants for timing intervals can be various from patients to patients. Normal values of these constants are 1000ms for LRI, 1200ms for HRI, and 320ms for VRP.

Event channels are used to communicate between different components of the system. The primary channels are VPace and VSense representing channels for paces and senses in the ventricle, respectively. There are two states WaitRI and WaitVRP in the Ventricle timed automaton in Figure 2(a).

- WaitRI: The pacemaker starts from this state and waits for a ventricular sensing or pacing event. If sensing does not occur before RI period ends, the ventricle controller sends a pacing signal to the heart and the timer x is reset. RI value turns into LRI and hp is set to false. hp is a boolean value for setting hysteresis pacing. When the hysteresis pacing is applied (i.e., hp is true), the pacemaker provides a longer period (i.e., HRI) following a sensing event before pacing. hp is set to true after every ventricular sensing or false after every ventricular pacing. It determines whether LRI and HRI will be assigned to RI. For Transition 2, when a ventricular sense occurs, the timer x is reset, HRI is assigned to RI and hp is set to true. Once the ventricle is paced or sensed, current state is changed to WaitVRP
- WaitVRP: In this state the pacemaker does nothing but waiting for a VRP period to end. It returns to the WaitRI state after a VRP period by setting hpenable to hp and started to true. hpenable and started are auxiliary variables to be used in property description for model checking.

2) Heart Model: The second automaton, Heart, shown in Figure 2(b), simulates a human heart. It is an environment for us to simulate and verify the pacemaker software in modeling phase. There is only one state, Ready, where the heart waits a ventricular pacing event from the pacemaker or randomly sends a ventricular sensing event to the pacemaker during time interval between lower bound (minwait) and higher bound (maxwait) for two consecutive heart beats. Once it sees an event in the channel VPace or VSense in the Ready state, it goes to the Ready state again by resetting the clock.

C. Formal Verification

We mapped the timing requirements to corresponding verification queries in UPPAAL. For easy reference to the properties, we labeled a unique name to each property. These are four properties which we used in verification:

- PropDeadlock: Deadlock freeness
  `A[] (not deadlock)`
  This property is checked to make sure that there are no deadlocks in any execution sequences.
- PropLRI: Lower Rate Limit under disabled Hysteresis Pacing
  `A[] (!Ventricle.hpenable imply Ventricle.x <= Ventricle.LRI)`
  When the hysteresis pacing is disabled (i.e., `Ventricle.hpenable` is false), `RI` should be the value of `LRI`. The pacemaker should trigger a ventricular pace before `RI` period expires, if the heart does not beat by itself.
- PropHRI: Hysteresis Rate Limit under enabled Hysteresis Pacing
  `A[] (Ventricle.hpenable imply Ventricle.x <= Ventricle.HRI)`
  When the hysteresis pacing is enabled, the pacemaker uses a pre-defined value `HRI` instead of `LRI` after every sensing. In the hysteresis-enabled status, the pacemaker should trigger a ventricular pace before `HRI` period expires, if the heart does not beat by itself.
- PropVRP: Ventricular Refractory Period
  `A[] ((Ventricle.WaitRI && Ventricle.started) imply Ventricle.x >= Ventricle.VRP)`
  The pacemaker should turn off ventricular sensing for a `VRP` period after any ventricular pacing or sensing event. In our model, the timer `Ventricle.x` is reset whenever the ventricular pacing or sensing

event happens. Therefore, we can check whether the ventricular controller follows the VRP period requirement or not by checking whether it stays in the WaitRI state for at least VRP period when the pacemaker detects a ventricular sensing or pacing event (i.e., Ventricle.started is true).

When we performed model checking on the model shown in Figure 2 with the above four properties, we confirmed that the model satisfied all these properties.

IV. Code Synthesis and Timing Analysis

We implemented the pacemaker software on a hardware reference platform of the Pacemaker Formal Method Challenge [1], which is based on a Microchip 8-bit PIC18F4520 MicroController Unit (PIC18 MCU) [12] running at 40 MHz clock speed. We generated two flavors of implementation code from the timed automata model. One is a single-threaded code where the timed automata models are implemented inside a large loop. The single-threaded code checks the current enabled transitions and takes one of them in each iteration. The other is a multi-threaded code where each transition in timed automata is implemented by an individual thread.

The multi-threaded approach used functionalities supported by the operating system called PICos18 [13] for the PIC18 MCU while the single-threaded approach did not use any operating system supports. The MPLAB C Compiler for PIC18 MCUs was used to compile these implementation codes.

We utilized MPLAB SIM, a software simulator for PIC18 MCU in MPLAB Integrated Development Environment (IDE) [12] to execute the code and measure timing aspects of it. With MPLAB, we can set the CPU frequency to various values. The faster the CPU is, the more instantly the program responds to an event. We select our simulator parameters to model the behavior of a minimal possible CPU. In practice, more powerful CPUs may be used and correctness and safety requirement hold.

A. Single-threaded Approach

1) Code Generation: This single-threaded code structure was inspired by techniques used in TIMES tool [14] which supports code synthesis from timed automata extended with tasks for Lego Mindstorms™ platform. While we use their basic code structure, we target a different platform, PIC18 MCU board and consider only timed automata without tasks.

The check_trans function includes the main algorithm. It checks all the transitions for every automaton, takes one transition if the guard holds, and loops until there are no more transitions to take. Listing 1 is a pseudo-code for the check_trans.

```
1  function check_trans
2  for each trn do
3      if (trn is active) and (eval_guard(trn))
4          if (there is synchronization)
5              if (compl_trn exists) and (eval_guard(
                   compl_trn))
6                  read test clock; /* verification
                       purpose */
7                  update variables of both trn and
                       compl_trn;
8                  take both trn and comp_trn to new
                       states;
9                  perform verification and print
                       results; /* verification
                       purpose */
10                 set trn to -1; /* to check all
                       outgoing transitions again */
11             end if
12         else /* no synchronization */
13             update variables of trn;
14             take the transition trn to new state;
15             set trn to -1;
16         end if
17     end if
18 end for
```

Listing 1. The modified function check_trans

The function check_trans is executed repeatedly as long as the system progresses. For each transition trn, the check_trans first checks whether the trn is active, i.e., whether it is an outgoing transition of the current location. If the trn is active and its guard evaluates to true, the check_trans examines whether there exists synchronization for this transition. If there is synchronization, it examines whether the complementary transition compl_trn is also active and the guard of the compl_trn evaluates to true. If these conditions are true, variables of both the trn and the compl_trn are updated and finally both transitions are taken. If no synchronization is involved, only trn is taken.

The system model may have several automata and states, and hence there may be a lot of transitions. However, each automaton has only one current state and outgoing transitions from the current state are usually not many. Most transitions do not go into over Line 5 in Listing 1. Thus, it does not take a long time to go over all transitions of the system.

2) Checking Timing Constraints:

Testing scenarios: In order to check whether the running pacemaker code satisfies the timing properties which have been verified in the model, we created a set of test scenarios by considering various possible sequences of interactions between heart and the pacemaker. We used interrupts to simulate signals from heart in MPLAB SIM. For the VVI mode pacemaker, at least five different scenarios are possible.

- Pacing–Pacing: when a patient heart does not send ventricular signals, a VVI mode pacemaker generates ventricular pacing signals every 1000ms continuously if LRI is set to 1000ms.
- Pacing–Sensing (during VRP): a pacemaker ignores any incorrect sensing signals during VRP after pacing;

sensing signals occurring during VRP do not affect the behavior of the pacemaker.

- Pacing–Sensing (after VRP): if a patient heart becomes able to make spontaneous ventricle signals after being paced, a VVI mode pacemaker can sense these ventricular signals after VRP.
- Sensing–Pacing: whenever a single, non-refractory sensed ventricular event occurs, it shall activate hysteresis pacing; thus, a VVI mode pacemaker waits for HRI, longer than LRI, before making next pacing signal.
- Sensing–Sensing: if a patient heart makes consecutive sensing signals, a VVI mode pacemaker inhibits scheduled ventricle pacing signals; it waits for HRI before making next pacing signal.

Checking method and result: We checked three timing properties PropLRI, PropHRI, and PropVRP by inserting instrumentation code. Because all three timing properties are simple safety properties which should be satisfied in all states, we put the property checking code at the end of the **for** loop as shown in Line 9 in Listing 1. We used a sequence of *if-then-else* statements for the property checking. For example, the following is a part of `perform verification` code to check PropLRI:

```
if( RI is LRI )
  if( timer <= LRI )
    print (TL: time_stamp)
  else
    print (FL: time_stamp)
  end if
end if
```

TL means that PropLRI is satisfied (true) and FL means that PropLRI is not satisfied (false). We read a value of the test clock right before the program updates variables.

#	Heart Event (ms)	VRP	LRI	HRI
1	P:1000.960	–	No	–
2	P:1000.448	–	No	–
3	S:910.464	Yes	–	–
4	S:1059.328	Yes	–	–
5	P:1200.832	–	–	No
6	P:1000.448	–	No	–

Table I
CHECKING RESULT OF SINGLE-THREADED CODE

Table I shows a summary of the program output. The first column is the sequential number of program output. The second column shows times, measured in millisecond, when sensing (S) or pacing (P) signals are detected. The rest columns show which properties among PropVRP, PropLRI, and PropHRI are relevant to the test scenario and how the checking results are. A dash means that the corresponding property is not relevant to the scenario. It is shown that the properties PropLRI and PropHRI which require that the pacemaker deliver pacing events no later than LRI (1000ms in our setting) and HRI (1200ms in our setting), respectively, did not hold. For example, the first row shows that the PropLRI was not satisfied because the pacing signal was delivered at 1000.960ms which is later than 1000ms. On the other hand, the property PropVRP which requires that sensing be detected only after VRP (320ms in our setting) always held.

Timing analysis: From the analysis of the violated properties, we found that code execution time is not ignorable in the aspect of property preservation for the pacemaker software. We also found that all the deviations from the expected time were less than 1ms in our experiment, as shown in TableI.

In order to get better understanding about the impact of code execution time to timing property preservation, we measured how much time is spent on execution of each piece of code in `check_trans`. By using the stopwatch functionality provided by the MPLAB IDE [12], we measured time periods between two breakpoints of the code and the whole execution time of the program. We tested the Pacing–Pacing scenario to measure execution time of each piece of the code. The Pacing–Pacing scenario consists of three phases of execution as follows:

- Before Pacing: this is a period between consecutive pacing signals; the program checks Transition 0 to Transition 3 in Figure 2(a) and 2(b) since enabled states are Ready and WaitRI, and repeats this checking of transitions for 1000ms.
- Pacing: at this moment, the pacemaker software delivers a pacing signal and takes the transition to WaitVRP state; it also prints a result such as `P (FL: 1000.448ms)` on screen.
- During VRP: the pacemaker does nothing but checks Transition 0, 1, and 4 in Figure 2(a) and 2(b), and repeats this for VRP (320ms).

When we measured time of each phase of the above scenario using the stopwatch feature, we obtained the following result of time measurement.

Before pacing: 1.070ms
Pacing: 25.054ms
  Before printing "P": 1.871ms
  Pacing (printing "P"): 2.175ms
  Verification including printing: 20.372ms
  After pacing: 628us
During VRP: 628us

Re-checking result: We modified our code according to the above analysis. We chose 2ms, among values greater than 1ms, as the value of timing tolerance Δ and did the experiment again. Specifically, our modified code now actually implements a model which differs from the model in Figure 2(a) slightly in which Transition 2 now has a guard of $x >= RI - \Delta$ rather than the original $x >= RI$. It means that the implementation will be able to check the

guard of the respective transition 2ms earlier; thus, making a pacing event no later than RI becomes possible, resulting in satisfying the corresponding timing property. Note that we did not change the guard of Transition 4 because the PropVRP was never violated. Only transitions relevant to the violated PropLRI and PropHRI were changed.

Table II is the result of enforcing 2ms timing tolerance in the program. We can now see that all of three timing properties are satisfied with the modified guards with the timing tolerance. Last but not the least, to validate the changes in the code, we also propagate changes backward to the models. With the timing tolerance value Δ, we verified all the properties on the modified model again. The result showed that all the properties were still satisfied. This result confirmed that the change of the code does not harm preservation of all the desired timing properties.

#	Heart Event (ms)	VRP	LRI	HRI
1	P:998.880	–	Yes	–
2	P:998.720	–	Yes	–
3	S:793.760	Yes	–	–
4	S:1000.704	Yes	–	–
5	P:1198.204	–	–	Yes
6	P:998.496	–	Yes	–

Table II
RECHECKING RESULT OF SINGLE-THREADED CODE

B. Multi-threaded Approach

There are at least two schemes to employ multiple threads to synthesize code for timed automata models. One is to implement the behavior of each timed automaton in one thread, and communications between channels of these timed automata are implemented using thread communications provided by the underlying operating system running on the board. In another approach, inspired by the generated code from ELASTIC2BRICK tool [4], the transition functions run in individual threads and are triggered by events sent to the respective threads.

1) Code Generation: The ELASTIC2BRICK tool can produce C code for LEGO MINDSTORMS™ running BRICKOS from timed automata models without invariants. Since it does not support shared variables and we are targeting PIC18 MCU platform [12], we did not use ELASTIC2BRICK directly but implemented our UPPAAL models in a manual but systematic scheme similar to the structure of the code automatically generated by the ELASTIC2BRICK tool. With thread support in PIC18 MCU's operating system PICos18 [13], our approach implements shared variables as process variables, which can be accessed by all threads in the hosting process.

In our code generation scheme, each transition of an automaton is transformed into a single thread. The structure for such a thread is as follows. The thread will wait for the semaphore for the location from which the corresponding transition starts. It then will check the guard by reading the clock value, waiting for input event signal semaphores, if any, and evaluating Boolean guards. If the guard is false, the thread will repeat the above logic. If the guard is true, it will execute the associated actions, including updating shared variables, resetting clocks, posting semaphores for output signals, and switching locations (changing a process variable `cur_loc` indicating the current location of the automaton, as well as posting respective semaphores). This scheme can be described with pseudo-code in Listing 2.[1] With this scheme, we can systematically generate code (though currently manually) for the multi-threaded approach.

```
task Template

while true
  WaitEvent(CURR_LOC_EVENT);
  ClearEvent(CURR_LOC_EVENT);
  thread specific calculation
  sleep when necessary
  if ((sleeping is used) and
      (cur_loc is changed when waking up))
        cancel transition
  else
    update variables
    clear timer when necessary
    change to destination location
    SetEvent(DEST_task_ID, EVENT_ID);
    perform verification
  end if
end while
```

Listing 2. Task Template

The variable `cur_loc` is used to check if the automaton is still at the same location when it wakes up from sleeping (corresponding to waiting for a certain period in the model). For example, in the ventricle controller model in Figure 2(a), Transition 2 and Transition 3 may both happen when the automaton is in the state WaitRI. The thread for Transition 2, however, will sleep for the duration of RI because of the guard condition $x >= RI$. In the meanwhile, Transition 3 may happen without such constraint and the corresponding thread can change the value of `cur_loc` to WaitVRP. When the thread for Transition 2 wakes up, checking for `cur_loc` fails and Transition 2 is not taken in effect.

2) Checking Timing Constraints:

Testing scenarios: As in the single threaded approach, we used the test scenarios of Pacing–Pacing, Pacing–Sensing, and Sending–Pacing and checked whether the properties PropLRI, PropHRI, and PropVRP hold in the code.

Checking method and result: Note that in the template in Listing 2, we also instrumented the `perform verification` which does the same checking as in the single-threaded approach. One difference, however, is that,

[1] On the PICos18 OS we are using, threads and semaphores are actually simplified to tasks and events, respectively.

to avoid printing overhead in the multi-threaded approach,[2] we pre-allocated memory locations to arrays for recording timing information. Printing of the functions is finished later when monitoring is stopped. In doing so, we can bring the time spent on monitoring to the minimum amount.

To measure the execution time of one iteration of a thread (corresponding to the execution time for one transition), we record the time immediately after the thread has been waken up, and when an execution of one iteration finishes, another time value is recorded.

We tested the program with the test scenarios several times, but repetitive execution did not make much differences because the experiment was conducted in the simulator environment and random numbers used to simulate heart beats were predefined ones. Experiment synthesized from one nominal sample run of this approach is shown in Table III. We observe from the result that the PropVRP property holds for all the values, while the PropHRI and PropLRI properties do not hold.

#	Heart Event (ms)	VRP	LRI	HRI
1	`P:1001`	–	No	–
2	`S:395`	Yes	–	–
3	`P:1202`	–	–	No
4	`S:687`	Yes	–	–
5	`S:1089`	Yes	–	–
6	`P:1202`	–	–	No
7	`S:419`	Yes	–	–
8	`S:1184`	Yes	–	–
9	`P:1202`	–	–	No
10	`S:642`	Yes	–	–
11	`P:1202`	–	–	No
12	`P:1001`	–	No	–

Table III
CHECKING RESULT OF MULTI-THREADED CODE

Timing analysis: According to the above experiment result, we can see that there is always a delay than expected pacing time. The maximum delay is bounded in the above result by 2 milliseconds.

One observation is that, making an event happen at the exact expected time is not possible, since there is always a processing delay between the time an event is supposed to happen and the time that it is actually carried out. This deviant is unavoidable for any implementations on any real platforms we use. However, another observation from our experiment result is that, the deviant values can be bounded.

If we incorporate this bound to the code and make the events happen earlier, it is possible to reduce the violations of the desired timing properties. For example, with the knowledge that all time delays are within 2ms, we can modify our code such that the events happen 2ms earlier than expected. This is the idea for the "re-checking" step.

[2]Due to limitations of the PICos18 OS, the minimum time resolution obtainable from the program is 1ms.

Re-checking result: We modified our multi-threaded code to incorporate the result of our experiment, using 2ms as the timing tolerance value Δ, and experimented again. The result is shown in Table IV.

#	Heart Event (ms)	VRP	LRI	HRI
1	`S:872`	Yes	–	–
2	`P:1200`	–	–	Yes
3	`S:398`	Yes	–	–
4	`S:1153`	Yes	–	–
5	`S:351`	Yes	–	–
6	`P:1200`	–	–	Yes
7	`P:1000`	–	Yes	–
8	`S:444`	Yes	–	–
9	`P:1200`	–	–	Yes
11	`P:1000`	–	Yes	–
12	`P:999`	–	Yes	–

Table IV
RECHECKING RESULT OF MULTI-THREADED CODE

It can be observed from the result that the PropLRI and PropHRI properties now hold. As is with the single-threaded approach, we modified the model according to the changes to the code and verified the properties again. All the properties were still satisfied by the modified model.

In the worst case, it may be possible that such a timing tolerance value of Δ cannot be found. For example, the smallest timing tolerance value of Δ may be larger than the allowed tolerance value from specifications. If this happens, the implementation of the code should be examined to see if the underlying coding scheme is sufficient enough for achieving timing constraints.

V. DISCUSSION

A. Type of Timing Properties

From the experiment of property checking in the code, we found that verification results are highly dependent on types of timing properties. We dealt with following types of timing properties: 1) whenever a certain event happens, a timer value should be greater than or equal to a certain value, and 2) whenever a certain event happens, a timer value should be less than or equal to a certain value. Properties requiring that a specific timer be always greater than or equal to a certain time limit when an event happens (e.g., VRP period check) were always satisfied on the code. We did not put any changes to the code in these satisfied cases. Properties requiring that a specific timer be always less than or equal to a certain time limit when an event happens (e.g., LRI and HRI period checks) were not satisfied in many cases. To make sure a desired event occur no later than the specified time instant, we made guard-checking done earlier by considering execution time of corresponding code segments. We considered two types of timing properties in our approach. More complex types of timing properties need to be considered in future work.

B. Instrumentation Overhead

Checking properties on the code often requires additional instrumentation code such as code for evaluating checking condition and printing. Time overhead from instrumentation code may sometimes cause the code to fail in satisfying timing properties. In our pacemaker example, time for the instrumentation code did not affect the code in its satisfaction of three major timing properties although time delay which we measured includes overhead time for the instrumentation code. However, this example does not cover general cases. Although instrumentation overhead is considered ignorable as far as it does not harm timing property preservation in our approach, reducing instrumentation overhead is strongly preferable. Extensive work has been done in code instrumentation and improving the performance and accuracy of time profilers based on code instrumentation. Existing techniques can be applied to our approach to make the Δ safely tighter.

C. Code Generation Scheme

We showed two different code generation schemes from timed automata and demonstrated how the proposed approach can be applied to them. Different code structure can make different impacts on timing properties. Although significant differences between two code schemes in terms of timing tolerance, performance, property preservation, etc. were not found in our case study, we witnessed that the multi-threaded code scheme has more time uncertainty than the single thread. We have a plan to analyze and evaluate different code generation schemes from timed automata.

D. Scalability

The proposed approach includes both manual and automatic operations. Scalability issue may be raised when the proposed approach is applied to development of complex real-time software. The proposed development approach would be much more useful if it is supported by automated tools. There are many spaces where automated tools can be used to improve the applicability of this approach to large real-time software.

Creating a formal model from system specification is generally done manually. Model checking on the created model is performed automatically. For the code synthesis, use of automated tools for code synthesis is highly recommended. However, it is not always easy to find tools fit for the developer's purpose because code synthesis for models is inherently model-dependent and platform-dependent. If it is the case, automatic code generation using an existing tool complemented by manual modification or development of a new tool can be considered. We utilized TIMES and ELASTIC2BRICK tools to generate the basis codes from the timed automata model and modified the generated codes manually to make them executable in a different hardware platform in our case study.

Alterations to the model based on code changes have been made manually in the proposed approach. However, it is highly possible to automate this alteration process by implementing backward mapping from the generated code to the formal model because mapping changes in the code to changes in the model may not be difficult as far as the code generation process is systematic.

E. Generalization

The proposed development approach can be applied to development of general safety-critical real-time software with different design decisions, although we showed a few specific decisions in our case study - timed automata as a modeling language, UPPAAL as a verification tool, stop watch technique as a timing analysis method, and two specific methods of code synthesis. As long as the modeling language captures timing behaviors of the system and the formal verification tool can check safety timing properties, other modeling languages and verification tools may also be used. As long as the code synthesis is systematic and sound, other synthesis techniques can also be used. Moreover, as long as the timing analysis method can give information to find Δ, other timing analysis methods can be used in the proposed development approach.

VI. RELATED WORK

A. Code Generation from Timed Automata

Code generation from state machines in general is a well studied topic [15]. Specifically, a couple of tools support generating code from timed automata [11]. The TIMES tool [14] is designed for modeling and implementation of embedded systems. Main targets are time/event triggered systems which can be described as sets of tasks. It supports code generation from timed automata extended with tasks for BRICKOS [6] platform. Assuming synchrony hypothesis, the code synthesis of the TIMES is guaranteed to preserve safety, schedulability and boundedness properties given those properties have been checked on a system design model prior to its implementation.

The ELASTIC2BRICK [16] tool is based on the AASAP semantics [4]. It takes a simplified version of timed automata without invariants as its specification language and generates code for BRICKOS platform. The ELASTIC2BRICK approach guarantees that safety properties proven correct with Δ in the model are preserved in the generated code, when certain assumptions for the execution time and the clock precision are satisfied. Neither shared variables among different automata nor broadcasting communications are supported by the ELASTIC2BRICK tool.

B. Timing Analysis on C code

Worst Case Execution Time (WCET) analysis is one of the commonly used timing analysis techniques. WCET analysis techniques are categorized into dynamic and static analysis.

Dynamic timing analysis is based on measuring exhaustive running on program code including profiling, emulating, and logic analysis. pWCET [17], a dynamic WCET analysis tool, computes probabilistic execution time bounds. Static timing analysis is based on analyzing mathematical model on program code without running program code. Since static timing analysis rely on mathematical model, it is more accurate than testing-based dynamic timing analysis. Some of commercial WCET tools using static timing analysis techniques are RapiTime [18], Bound-T [19], and aiT [20]. [18] and [20] support WCET analysis for restricted subsets of ANSI C.

Currently, we are using a rather simple technique to obtain execution time of the code in the proposed approach. We have a plan to use existing WCET analysis techniques and tools to find more rigorous timing tolerances.

VII. Conclusion

We presented a safety-assured development method of a real-time software. While following model-driven development concept basically, we modeled the real-time system with timed automata and performed model checking on it. After generating implementation code systematically from timed automata model, we checked preservation of properties transferred from model on the implementation code. When a timing property was violated on the code, we tried to find a timing tolerance Δ, which can be used to make the property satisfied by enlarging the corresponding guard in the code. This Δ was obtained based on measuring time deviation from the expected one. In order to confirm that this Δ can be safely used, model checking with respect to all the properties was performed again on the modified model having corresponding changes. We used a pacemaker software as our case study to demonstrate how this approach can be applied to development of real-time software. Theoretical analysis of the proposed approach and comparison of different code generation schemes are in progress. We also have a plan to construct assurance cases for our pacemaker software to figure out how the proposed methodology can affect construction of assurance cases.

Acknowledgment

This research was supported in part by NSF CNS-0931239, NSF CNS-0930647, NSF CNS-0834524, and NSF CNS-0720703.

References

[1] Software Quality Research Laboratory, "Pacemaker formal methods challenge," http://sqrl.mcmaster.ca/pacemaker.htm.

[2] G. Behrmann, A. David, and K. G. Larsen, *A Tutorial on Uppaal*, Department of Computer Science, Aalborg University, Denmark, Nov 2004.

[3] T. Amnell, E. Fersman, P. Pettersson, W. Yi, and H. Sun, "Code synthesis for timed automata," *Nordic J. of Computing*, vol. 9, no. 4, pp. 269–300, 2002.

[4] M. de Wulf, L. Doyen, and J.-F. Raskin, "Systematic implementation of real-time models," in *FM*, ser. Lecture Notes in Computer Science, J. Fitzgerald, I. J. Hayes, and A. Tarlecki, Eds., vol. 3582. Springer, 2005, pp. 139–156.

[5] K. Altisen and S. Tripakis, "Implementation of timed automata: an issue of semantics or modeling," in *In Proc. 3rd Int. Conf. Formal Modelling and Analysis of Timed Systems (FORMATS05), Lecture Notes in Computer Science*. Springer, 2005, pp. 273–288.

[6] S. Nielsson, "Introduction to the legOS kernel," Sep. 2000.

[7] M. D. Wulf, L. Doyen, and J. francois Raskin, "Almost ASAP semantics: From timed models to timed implementations." Springer, 2003, pp. 296–310.

[8] Oregon Health and Science University, "Overview of pacemakers," http://www.ohsu.edu/health/index.cfm.

[9] S. S. Barold, R. X. Stroobandt, and A. F. Sinnaeve, *Cardiac Pacemakers Step by Step: An Illustrated Guide*. Blackwell Futura, 2005.

[10] Boston Scientific, "Pacemaker system specification," Jan 2007, http://sqrl.mcmaster.ca/_SQRLDocuments/PACEMAKER.pdf.

[11] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, pp. 183–235, 1994.

[12] Microchip, "PIC18 family microcontroller," http://www.microchip.com/.

[13] Pragmatec Inc, "PICos18 : RTOS for PIC18," http://www.picos18.com/.

[14] Uppsala University Design and Analysis of Real-Time Systems team, "TIMES–a tool for modeling and implementation of embedded systems," 2007, http://www.timestool.com.

[15] M. Samek, *Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems*, 2nd ed. Elsevier, Oct. 2008.

[16] M. D. Wulf, "From timed models to timed implementations," Ph.D. dissertation, Université Libre De Bruxelles, 2007.

[17] NEXT TTA, "pwect: Probabilistic worst case execution time analysis," http://www.pwcet.com/.

[18] Rapita Systems Ltd., "Rapitime on target timing analysis," http://www.rapitasystems.com/rapitime.

[19] Tidorum Ltd., "Bound-T time and stack analyser," http://www.pwcet.com/.

[20] AbsInt., "aiT worst-case execution time analyzers," http://www.pwcet.com/.