Hindawi Publishing Corporation
Journal of Electrical and Computer Engineering
Volume 2015, Article ID 939028, 10 pages
http://dx.doi.org/10.1155/2015/939028

Research Article

A Formal Verification Methodology for DDD Mode Pacemaker Control Programs

Sana Shuja, Sudarshan K. Srinivasan, Shaista Jabeen, and Dharmakeerthi Nawarathna

Department of Electrical and Computer Engineering, North Dakota State University, 1411 Centennial Boulevard, Fargo, ND 58102, USA

Correspondence should be addressed to Sudarshan K. Srinivasan; sudarshan.srinivasan@ndsu.edu

Received 1 June 2015; Revised 4 August 2015; Accepted 12 August 2015

Academic Editor: Massimo Poncino

Pacemakers are safety-critical devices whose faulty behaviors can cause harm or even death. Often these faulty behaviors are caused due to bugs in programs used for digital control of pacemakers. We present a formal verification methodology that can be used to check the correctness of object code programs that implement the safety-critical control functions of DDD mode pacemakers. Our methodology is based on the theory of Well-Founded Equivalence Bisimulation (WEB) refinement, where both formal specifications and implementation are treated as transition systems. We develop a simple and general formal specification for DDD mode pacemakers. We also develop correctness proof obligations that can be applied to validate object code programs used for pacemaker control. Using our methodology, we were able to verify a control program with millions of transitions against the simple specification with only 10 transitions. Our method also found several bugs during the verification process.

1. Introduction

The heart generates electrical signals to induce heartbeat. The heart's electrical system can become defective due to aging or other causes, leading to a slower heart rate (bradycardia). Such ailments can be treated using pacemakers, which are implantable medical devices that generate the electrical signals required to keep the heartbeat at a healthy rate. Faulty pacemakers can cause harm or even death to the patients using them. Hence pacemakers are safety-critical devices [1, 2].

A control program executed on a microcontroller embedded in a pacemaker is responsible for implementing the control functions of the device. With pacemakers being safety-critical, bugs in the control program cannot be tolerated. Medical devices such as pacemakers are very prone to software errors due to the complex control algorithms that they use [3]. From 2001 to 2015, the U.S. Food and Drug Administration (FDA) has issued 38 Class 1 recalls on medical devices due to software problems [4]. Currently, 169,184 units have been documented by the FDA to have been affected by these recalls. A Class I recall indicates that the continued use of the recalled medical device can result in harm or death to the patient.

We present a formal verification methodology [5] that can be used to check the correctness of control programs used in DDD mode pacemakers. The three-letter code of DDD represents that the pacemaker provides *"Dual"* chamber pacing, *"Dual"* chamber sensing, and an action of activation or inhibition of further pacing in *"Dual"* chambers on a sensed event. Pacemakers are most commonly used in the DDD mode. Our methodology is targeted at verifying control programs at the object code level. Control programs are coded using a high-level programming language. The resulting code (called source code) is compiled to generate object code, which is what is executed by the microcontroller embedded in the device. Validating source code is not sufficient for safety-critical devices, as the compilation process can introduce bugs in the object code.

The specific contributions of our work are as follows. First, we have developed a high-level formal specification that captures the safety-critical software requirements of a DDD mode pacemaker. We use the notion of a timed transition system (TTS) to model the specification, which captures both

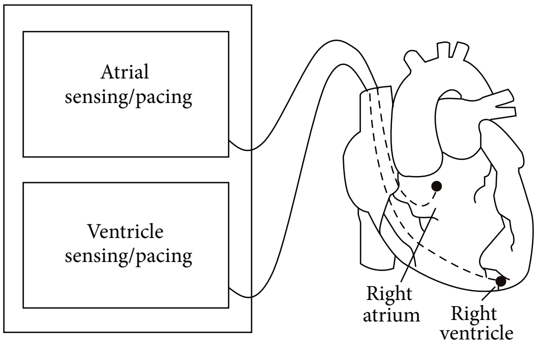

Figure 1: The interface between a heart and a DDD mode pacemaker.

functional and timing requirements. Second, based on the specification, we have developed a generic invariant predicate that captures the reachable states of a *DDD mode pacemaker object code control program (henceforth referred to as the implementation)*. The invariant essentially eliminates most of the unreachable states, which can cause spurious counterexamples during verification and significantly deteriorate the effectiveness of the verification process. Third, we have developed rank functions that are used to detect deadlock bugs in the implementation. Fourth, using the specification, invariant, and rank functions, we have developed a set of proof obligations that can be used to effectively check the correctness of the implementation. The proof obligations can be discharged using an SMT solver [6, 7] such as z3 [8]. Our methodology has been used to verify an implementation control program with over two million transitions. Note that, in contrast, our high-level specification has only 10 transitions. Our methodology also found several bugs in the implementation that we verified.

2. Background: DDD Mode Pacemakers

The heart is a four-chambered organ and has a pair of atria (left and right atrium) mounted on a pair of ventricles (left and right ventricle). The sinoatrial (SA) node, a set of specialized tissues located on the right atrium, is responsible for generating periodic electrical pulses. These electrical pulses contract the walls of the atria pushing the blood to the ventricles. The atrioventricular (AV) node which is a bundle of specialized tissues situated between the atria and ventricles does not allow the electrical signals to transmit to the ventricles until the ventricles are filled with blood. The bundle next to the AV node eventually transmits the electrical pulses to the ventricles with the aid of Purkinje fibers, causing the muscles of the ventricles to contract and pump the blood at a healthy pace to the entire body.

The mechanism of the pacemaker revolves around sensing and signaling of electrical pulses. A DDD mode pacemaker has leads connected to the right atrium and right ventricle [9]. The interface between a heart and a DDD mode pacemaker is shown in Figure 1. The leads sense the atrium for the atrial sense (AS: the electrical pulse that contracts the walls of the atria) and sense the ventricle for ventricle sense (VS: the electrical pulse that contracts the walls of the ventricle). If no AS or VS occurs within a healthy heart's time limits, the pacemaker generates electrical pulses to contract the atrium or the ventricle, respectively. The signals generated by the pacemaker to pace the atrium and the ventricle are called an atrial pace (AP) and a ventricle pace (VP), respectively.

The critical timing cycles of a DDD mode pacemaker as described by Barold et al. [10] are given below.

Lower Rate Interval (LRI). LRI is the longest interval between a ventricle event $\in$ {VS, VP} and the subsequent ventricle paced event (VP) without superseding sensed events.

Ventricular Refractory Period (VRP). VRP is initiated by a ventricle event $\in$ {VS, VP}. During VRP, LRI cannot be initiated or reset. During this period, a pacemaker does not respond to incoming signals.

Atrioventricular Interval (AVI). AVI is the time interval between an atrial event $\in$ {AS, AP} and the following ventricle event.

Atrial Refractory Period (ARP). ARP is the interval after a ventricular event $\in$ {VS, VP}. During this interval no atrial event can initiate a new AVI.

Upper Rate Interval (URI). URI limits the ventricle pacing rate by imposing a lower limit on consecutive ventricle events $\in$ {VS, VP}.

Atrial Escape Interval (AEI). AEI is the interval between a ventricle event $\in$ {VS, VP} and the subsequent atrial pacing event (AP) with no intervening sensed events

$$\mathrm{AEI} = \mathrm{LRI} - \mathrm{AVI}. \quad (1)$$

3. Related Work

Tuan et al. [11] have developed a formal model for a pacemaker as an RTS (real-time system) model. Correctness properties were checked using the PAT model checker. Gomes and Oliveira proposed a formal specification of a pacemaker using the Z notation [12]. They used the ProofPower-Z theorem prover to check if their specification model satisfied the pacemaker requirements. A Dual chamber implantable pacemaker was taken as a case study for modeling and verification of control algorithms for medical devices in UPPAAL [13, 14]. All of the above works are formal verification methods targeted at the verification of high-level pacemaker control models. In contrast, our formal verification methodology is targeted at the verification of *low-level interrupt driven object code* (which is what is executed by the microcontroller embedded in the pacemaker device).

In Section 4, we develop a formal specification model for DDD mode pacemaker control. Above, we have outlined several previous works that have proposed formal models for pacemaker control. Why do we develop another model? As stated earlier, our goal is to develop a verification methodology for object code. We use the theory of WEB refinement for this purpose. In Section 5, we have described why we use the WEB refinement theory. This theory of refinement requires that both the implementation and specification be modeled as transition systems. The previous formal models for pacemaker control cannot be employed in the context of WEB refinement. Also, we have developed a specification model that is as simple and clear and of high-level as possible, so that the resulting verification methodology is efficient.

Jiang et al. [15] proposed a set of general and patient condition-specific temporal requirements for the closed-loop heart and pacemaker system. They also have developed a closed-loop testing environment between a timed automata-based heart model and a pacemaker. Jiang et al. [16] have developed a cyber-physical system (CPS) model of the heart and used this model for testing of a pacemaker model and software. The above methods are based on testing, whereas we propose a formal verification methodology. The methods can be considered to complement each other. Testing is of course the industry standard and very useful in finding bugs. Formal verification is useful in locating corner-case hard-to-find bugs and can also be used to provide guarantees about software correctness. Another contrast with the above works is that they have developed and used a CPS model of the heart, which is used to test the software. We verify the low-level software against the high-level software requirements.

4. Formal Specification Model for DDD Mode Pacemakers

The requirements of a DDD mode pacemaker are given in [17]. These requirements are based on two timelines t_a and t_v. t_a is the time elapsed since the last atrial event (AS or AP). t_v is the time elapsed since the last ventricle event (VS or VP). A_{in} is the atrial input and V_{in} is the ventricle input received from the heart. If valid A_{in} is detected, then the pacemaker registers an atrial sense event (AS). If valid V_{in} is detected, then the pacemaker registers a ventricle sense event (VS). Figure 1 shows the interface between the heart and the pacemaker. The requirements from [17] are given below.

Atrial Sensing

AS.1: AS cannot occur within the interval $t_v \in (0, \text{ARP}]$.

AS.2: if atrial input (A_{in}) occurs within interval $t_v \in (0, \text{ARP})$, it should be disregarded (no AS is generated within $t_v \in (0, \text{ARP})$).

AS.3: if A_{in} occurs at $t_v \geq \text{ARP}$, AS is to be created at t_v.

Ventricle Sensing

VS.1: VS cannot be generated within the interval $t_v \in (0, \text{VRP})$.

VS.2: if ventricle input (V_{in}) occurs at $t_v \in (0, \text{VRP})$, it should be ignored (no VS is generated within $t_v \in (0, \text{VRP})$).

VS.3: if V_{in} occurs at $t_v \geq \text{VRP}$, VS is to be created at t_v.

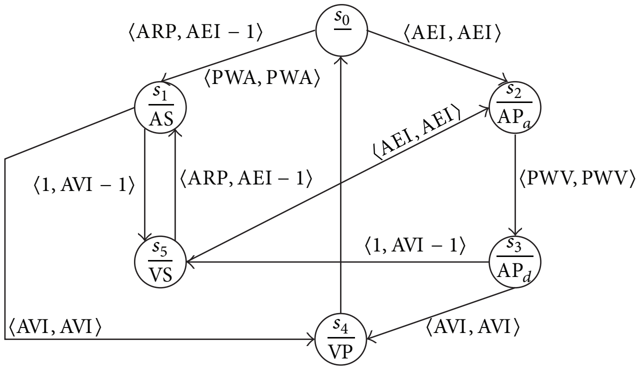

FIGURE 2: Figure depicts the TTS M_{PM} Specification.

Atrial Pacing

AP.1: AP cannot occur during the interval $t_v \in [0, \text{AEI})$, where $\text{AEI} = \text{LRI} - \text{AVI}$.

AP.2: if AS does not occur within interval $t_v \in [0, \text{AEI})$, an AP should occur at $t_v = \text{AEI}$.

AP.3: if AS occurs at $t_v \in [0, \text{AEI})$, AP should not be applied in the atrium within the interval $t_v \in [0, \text{AEI})$.

Ventricle Pacing

VP.1: VP cannot occur during the interval $t_a \in (0, \text{AVI})$.

VP.2: VP cannot be generated within $t_v \in (0, \text{URI})$.

VP.3: if VS does not occur in intervals $t_a \in (0, \text{AVI})$ and $t_v \geq \text{URI}$, VP should occur at $t_a = \text{AVI}$.

VP.4: if VS occurs at $t_a \in (0, \text{AVI})$, no VP should be generated within the interval $t_a \in (0, \text{AVI})$.

We present a formal specification model that captures the above requirements. We use timed transition system (TTS) to model the pacemaker specification. TTS is defined as follows.

Definition 1. A timed transition system (TTS) $\mathscr{M}$ is a 3-tuple $\langle S, R, L \rangle$, where S is the set of states, R is the transition relation, which is the set of all state transitions, and L is a labeling function that defines what is visible at each state. A state transition is of the form $\langle w, v, lb, ub \rangle$, where $w, v \in S$ and $lb, ub \in \mathfrak{R}$. lb and ub indicate the lower bound and the upper bound on the time delay of the transition, respectively.

The TTS specification is shown in Figure 2. The TTS specification $\mathscr{M}_{\text{PM}} = \langle S_{\text{PM}}, R_{\text{PM}}, L_{\text{PM}} \rangle$ has 6 states:

$$S_{\text{PM}} = \{s_0, s_1, s_2, s_3, s_4, s_5\}. \quad (2)$$

We use 5 atomic propositions for the model. Atomic propositions are predicates that are either true or false in each of the states. The atomic propositions are AS, AP_a, AP_d, VS, and VP. AS and VS indicate atrial sense and ventricle sense, respectively. VP indicates ventricle pacing. For atrial pacing (AP), we use two atomic propositions AP_a and AP_d. AP_a indicates when the pacemaker should assert an atrial pacing and AP_d indicates when the pacemaker should deassert an atrial pacing. The transition relation of the specification TTS is given below:

$$\begin{aligned} R_{\mathrm{PM}} = \{ & \langle s_0, s_2, \mathrm{AEI}, \mathrm{AEI} \rangle, \langle s_0, s_1, \mathrm{ARP}, \mathrm{AEI} - 1 \rangle, \\ & \langle s_1, s_4, \mathrm{AVI}, \mathrm{AVI} \rangle, \langle s_2, s_3, \mathrm{PWV}, \mathrm{PWV} \rangle, \\ & \langle s_1, s_5, 1, \mathrm{AVI} - 1 \rangle, \langle s_5, s_1, \mathrm{ARP}, \mathrm{AEI} - 1 \rangle, \\ & \langle s_3, s_5, 1, \mathrm{AVI} - 1 \rangle, \langle s_4, s_0, \mathrm{PWA}, \mathrm{PWA} \rangle, \\ & \langle s_3, s_4, \mathrm{AVI}, \mathrm{AVI} \rangle, \langle s_5, s_2, \mathrm{AEI}, \mathrm{AEI} \rangle \}. \end{aligned} \tag{3}$$

Pulse Width Atrial (PWA) and Pulse Width Ventricle (PWV) signify the time for which the ventricle pacing signal (VP) and the atrial pacing signal (AP), respectively, should remain asserted. PWA and PWV indicate the length of the pulses on the atrial timeline t_a and the ventricle timeline t_v, respectively, and are hence named as such. The labeling function is defined as follows:

$$\begin{aligned} L_{\mathrm{PM}}(s_0) &= \phi, \\ L_{\mathrm{PM}}(s_1) &= \{\mathrm{AS}\}, \\ L_{\mathrm{PM}}(s_2) &= \{\mathrm{AP}_a\}, \\ L_{\mathrm{PM}}(s_3) &= \{\mathrm{AP}_d\}, \\ L_{\mathrm{PM}}(s_4) &= \{\mathrm{VP}\}, \\ L_{\mathrm{PM}}(s_5) &= \{\mathrm{VS}\}. \end{aligned} \tag{4}$$

We now describe $\mathscr{M}_{\mathrm{PM}}$ and how it relates to the requirements.

State s_0. s_0 is the reset state. In this state, the pacemaker is expecting an atrial sense. If an atrial sense is detected, the pacemaker should transition to s_1, which is the state labeled with the AS predicate. However, an Ain that occurs in the interval $t_v \in (0, \mathrm{ARP})$ should be ignored (requirements AS.1 and AS.2). Also, if Ain occurs for $t_v \geq \mathrm{ARP}$, then AS should be generated (requirement AS.3). Requirements AS.1, AS.2, and AS.3 are enforced by imposing a lower bound of ARP on the transition $s_0 \rightarrow s_1$. If $t_v = \mathrm{AEI}$, then the pacemaker should generate an atrial pace AP (requirement AP.2). Therefore, the maximum time the pacemaker can wait for an AS is AEI – 1, which is the upper bound for $s_0 \rightarrow s_1$. Also, when $t_v = \mathrm{AEI}$ and an AS has not occurred yet, the pacemaker should generate an AP. Therefore, the specification transitions from s_0 to s_2 with a lower bound and upper bound of AEI. The lower bound of AEI for $s_0 \rightarrow s_2$ also satisfies AP.1. If the pacemaker transitions to s_1, it cannot generate an AP in the interval $t_v \in [0, \mathrm{AEI})$, because there is no path in the specification model from s_1 to s_2 in this interval. Therefore, the specification model captures requirement AP.3.

State s_1. After an atrial sense (AS) has occurred, the pacemaker waits for a ventricle sense in state s_1. If a VS occurs, the pacemaker transitions to state s_5, which is marked by predicate VS. Requirement VP.3 states that the maximum time a pacemaker can wait for a VS is $t_a \in (0, \mathrm{AVI})$, which enforces a lower bound of 1 and an upper bound of AVI – 1 on the transition $s_1 \rightarrow s_5$. Also, when $t_a = \mathrm{AVI}$ and a VS has not occurred yet (requirement VP.3), the pacemaker should generate a VP. Therefore, the specification transitions from s_1 to s_4 with an upper bound of AVI. Also from VP.1 we get a lower bound of AEI for the transition s_1 to s_4. If the pacemaker transitions to s_5, it cannot generate a VP in the interval $t_a \in [0, \mathrm{AVI})$ because there is no path in the specification model from s_5 to s_4 in this interval. Therefore, the specification model captures requirement VP.4.

State s_5. In state s_5, a VS has just occurred. The pacemaker is now waiting for an atrial event. Therefore, state s_5 is similar to state s_0 and has similar transitions. s_0 transitions to s_1 and s_2. Similarly, s_5 also transitions to s_1 and s_2 with the same lower and upper bounds for both transitions.

State s_3. In state s_3, an atrial event has just been completed. The pacemaker is now waiting for a ventricle event. Therefore, state s_3 is similar to state s_1 and has similar transitions. s_1 transitions to s_5 and s_4. Similarly, s_3 also transitions to s_5 and s_4 with the same lower and upper bounds for both transitions.

States s_2, s_4. From [18], the pulse width for emergency bradycardia pacing is approximately 1.00 ms ± 0.02 ms. Therefore, the pulse width of both AP and VP should be 1.00 ms ± 0.02 ms. AP is asserted in s_2 and deasserted in s_3. Therefore, $s_2 \rightarrow s_3$ has a lower bound and upper bound of PWV = 1.00 ms ± 0.02 ms; PWV stands for Pulse Width Ventricle, as the next event is a ventricle event. Similarly, VP is asserted in s_4 and deasserted in s_0. Therefore, $s_4 \rightarrow s_0$ has a lower bound and an upper bound of PWA = 1.00 ms±0.02 ms; PWA stands for Pulse Width Atrial, as the next event is an atrial event.

So far the specification TTS accounts for requirements AP.1–AP.3, AS.1–AS.3, VP.1, VP.3, and VP.4. Requirements VS.1, VS.2, and VS.3 can be enforced by imposing a lower bound of VRP on when VS is generated, but on the t_v timeline. VS is generated in state s_5. However, t_v is reset in states in which a ventricle event is completed, which are states s_0 and s_5. Hence the requirements VS.1, VS.2, and VS.3 can be enforced by imposing a lower bound on the combined delays of transitions $\langle s_0, s_1 \rangle$, $\langle s_1, s_5 \rangle$ and transitions $\langle s_0, s_2 \rangle$, $\langle s_2, s_3 \rangle$, and $\langle s_3, s_5 \rangle$. These constraints are not expressible in TTS. Therefore, we introduce a new notion called composite TTS (CTTS) defined below, to capture such requirements.

Definition 2. A composite constraint is a finite tuple $\langle s_i, s_{i+1}, \ldots, s_n, lb, ub \rangle$ such that, for $i \leq j < n$, $\langle s_j, s_{j+1} \rangle \in R$ and $lb, ub \in \mathfrak{R}$. lb and ub indicate the lower bound and the upper bound on the combined time delays of transitions $\langle s_i, s_{i+1} \rangle, \ldots, \langle s_{n-1}, s_n \rangle$, respectively.

Definition 3. A composite TTS (CTTS) is 4-tuple $\langle S, R, L, R_C \rangle$, where $\langle S, R, L \rangle$ is a TTS and R_C is a set of composite constraints.

The composite constraints corresponding to requirements VS.1, VS.2, and VS.3 are given below:

$$R_{C1} = \{ \langle s_0, s_1, s_5, \mathrm{VRP}, X \rangle, \langle s_0, s_2, s_3, s_5, \mathrm{VRP}, X \rangle \}. \tag{5}$$

In the above and in the discussions that follow, X indicates a don't care. X for a lower bound indicates that there is no requirement on the lower bound. Similarly, X on the upper bound indicates that there is no requirement on the upper bound.

Requirement VP.2 also results in composite constraints. VP.2 gives a lower bound on when VP can be generated, but on the t_v timeline. VP is generated in s_4. t_v is reset in s_0 and s_5. Hence, requirement VP.2 can be enforced by imposing a lower bound on the combined delays of transitions $\langle s_0, s_1\rangle$, $\langle s_1, s_4\rangle$ and transitions $\langle s_0, s_2\rangle$, $\langle s_2, s_3\rangle$, and $\langle s_3, s_4\rangle$. Therefore, to satisfy VP.2, the following composite constraints are required:

$$R_{C2} = \{\langle s_0, s_1, s_4, \text{URI}, X\rangle, \langle s_0, s_2, s_3, s_4, \text{URI}, X\rangle\}. \tag{6}$$

The composite constraint of the pacemaker specification R_C is given by

$$R_C = R_{C1} \cup R_{C2}. \tag{7}$$

4.1. Verification of CTTS Specification Using UPPAAL. We checked that the CTTS specification satisfies all the DDD mode pacemaker requirements from [17] (also given in Section 4) using UPPAAL [19, 20], which is a standard tool for checking properties of timed systems [21]. UPPAAL can be used to check if a real-time system modeled as a network of timed automata satisfies properties expressed in CTL (Computational Tree Logic) [22]. We encoded the CTTS specification as a timed automaton [23] and expressed all the requirements as CTL properties. We were able to verify that the CTTS specification satisfied all the CTL properties corresponding to the requirements.

The UPPAAL model corresponding to the CTTS specification is described next. In UPPAAL, states are represented as locations, and locations are connected with edges. Edges represent transitions. An edge emanating from a state can be labeled with a guard or an update or both. The edge is enabled if the guard of that edge is evaluated to true. An update on an edge is an expression that is executed when the guard is evaluated to be true. The UPPAAL model of the CTTS specification is shown in Figure 3. Each state in the CTTS specification has a corresponding location in the UPPAL model. The UPPAL model has three additional locations: s_1c, s_3c, and s_5c. We will describe the need for these additional locations shortly. Timelines t_v and t_a, which are described in Section 4, are modeled as clocks clkt_v and clkt_a in UPPAAL. Clocks are encoded as state variables. The timing requirements (lower bounds and upper bounds on transitions) including the composite constraints are modeled as guards on the clock variables in UPPAAL. Due to lack of space, the UPPAAL model is marked with guards labeled with g_{xy}, where x is the source state and y is the destination state of the transition. The guards are given below. g_{1c5} and g_{3c5} incorporate composite constraints in R_{C1}. g_{1c4} and g_{3c4} incorporate composite constraints in R_{C2}:

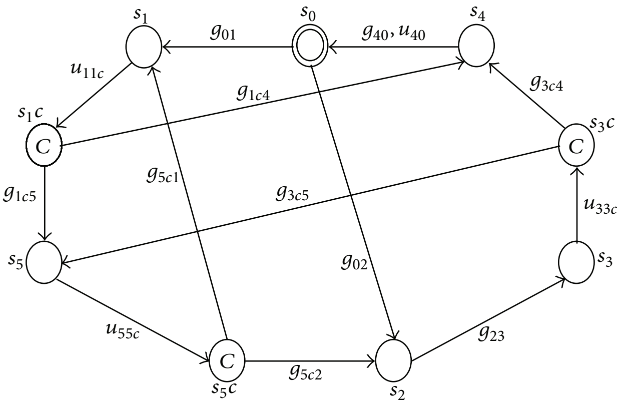

Figure 3: UPPAAL model of formal specification model.

$$\begin{aligned}
g_{01} &\longleftarrow ((\text{AEI} - 1) \geq \text{clk}t_v \geq \text{ARP}),\\
g_{02} &\longleftarrow (\text{clk}t_v = \text{AEI}),\\
g_{1c4} &\longleftarrow (\text{clk}t_a = \text{AVI}) \wedge (\text{clk}t_v \geq \text{URI}),\\
g_{1c5} &\longleftarrow (\text{clk}t_a \leq (\text{AVI} - 1)) \wedge (\text{clk}t_v \geq \text{VRP}),\\
g_{23} &\longleftarrow (\text{clk}t_v = \text{PWV}),\\
g_{3c4} &\longleftarrow (\text{clk}t_a = \text{AVI}) \wedge (\text{clk}t_v \geq \text{URI}),\\
g_{3c5} &\longleftarrow (\text{clk}t_a \leq (\text{AVI} - 1)) \wedge (\text{clk}t_v \geq \text{VRP}),\\
g_{40} &\longleftarrow (\text{clk}t_a = \text{PWA}),\\
g_{5c1} &\longleftarrow ((\text{AEI} - 1) \geq \text{clk}t_v \geq \text{ARP}),\\
g_{5c2} &\longleftarrow (\text{clk}t_v = \text{AEI}).
\end{aligned} \tag{8}$$

Timeline t_v is reset in states s_0 and s_5 and timeline t_a is reset in states s_1 and s_3. In the UPPAAL model, timelines are expressed using clock variables that are encoded as part of the state, whereas, in CTTS, timelines are delays on the transitions between states. In CTTS, timelines are therefore essentially reset (automatically) after every transition. Constraints involving more than one transition are encoded as composite constraints. In UPPAAL, clocks are not automatically reset. Therefore, we need additional states to reset clock variables. These additional states used to reset clock variables are called committed states in UPPAAL where time is frozen. Hence, we split each of the CTTS states s_1, s_3, and s_5 into two locations in UPPAAL. For example, state s_1 is modeled as locations s_1 and s_1c. Incoming transitions to state s_1 are mapped as incoming edges to location s_1 and outgoing transitions of state s_1 are mapped to outgoing edges of location s_1c. Clock is reset using an update $u_{11c} = (\text{clk}t_a = 0)$ on the edge from location s_1 to location s_1c. s_3 and s_5 are similarly modeled. The reason that we do not have a committed state for s_0 is that the guard g_{40} is dependent on clock clkt_a while the clock that is reset in this transition is clkt_v. The updates are given below:

$$\begin{aligned} u_{11c} &= \left(\mathrm{clk}t_a = 0\right), \\ u_{33c} &= \left(\mathrm{clk}t_a = 0\right), \\ u_{40} &= \left(\mathrm{clk}t_v = 0\right), \\ u_{55c} &= \left(\mathrm{clk}t_v = 0\right). \end{aligned} \tag{9}$$

We next describe the CTL properties that we verified. The properties are specified using state (location) operator A, which is a path quantifier that denotes all paths emanating from this state. We also use the temporal operator [ ] (globally), which indicates all states in the path. We have one property for each requirement. Below we give three examples.

VP.2 introduced a composite constraint encoded with the following CTL property:

$$A\,[\,]\,\left\{\left(\mathrm{clk}t_v < \mathrm{URI}\right) \longrightarrow \neg\left(s_4\right)\right\}. \tag{10}$$

The above property specifies that no VP can be generated within $t_v \in (0, \mathrm{URI})$. Note that we use state names (s_4) as opposed to atomic propositions in the properties, because each state (location) is associated with only one atomic proposition. s_1 and s_1c correspond to AS, s_2 corresponds to AP$_a$, and so on. AS.1 requires that no AS can be sensed within $t_v \in (0, \mathrm{ARP})$, expressed as the following property:

$$A\,[\,]\,\left\{\left(0 < \mathrm{clk}t_v < \mathrm{ARP}\right) \longrightarrow \neg\left(s_1 \vee s_{1c}\right)\right\}. \tag{11}$$

VS.1 states that no VS can be sensed within $t_v \in (0, \mathrm{VRP})$, expressed as the CTL formula in UPPAAL as

$$A\,[\,]\,\left\{\left(0 < \mathrm{clk}t_v < \mathrm{VRP}\right) \longrightarrow \neg\left(s_5 \vee s_{5c}\right)\right\}. \tag{12}$$

5. Formal Verification Methodology for Object Code Control Programs

In this section, we develop a methodology for formal verification of control programs for DDD mode pacemakers. Our methodology is targeted at the validation of the control programs at the object code level. For the verification methodology, we employ the theory of Well-Founded Equivalence Bisimulation (WEB) refinement [24], which is a notion of correctness that defines what it means for a low-level implementation (such as an object code program) to satisfy a high-level specification (such as the specification given in Section 4). In the context of WEB refinement, both the implementation and specification are modeled as transition systems (TSs). In Section 4, we have developed a TS specification for pacemaker control. The object code program can also be modeled as a TS. The instructions corresponding to the control program can be modeled as functions that capture the transitions of the program. The functions would take as input the current program state and values of program inputs and give the next state of the program as output.

Examined at a high-level, there are two differences between the TS corresponding to the object code control program and the specification TS. First, states of the specification TS can be encoded using 5 bits (AS, AP$_a$, VS, VP, and AP$_d$), whereas states of the implementation TS have other state components such as the registers in peripheral timers used to enforce the various timing cycles in the controller. The theory of WEB refinement [24] employs refinement maps, which are functions that map implementation states to specification states and are used to overcome differences in the implementation states and specification states. Refinement maps enable the comparison of implementation states and specification states, even if these states look very different. Second, the object code program TS has many more transitions than the specification. For the case study we use, the object code program has more than 2 million transitions, whereas the specification TS has only 10 transitions. Thus, typically, many transitions of the low-level implementation controller can match a single transition of the specification. This phenomenon is known as stuttering and is accounted for by WEB refinement. Below are the definitions for WEBs and WEB refinement. In [24–27] a more detailed description of WEB refinement is provided.

Definition 4 (see [25]). $B \subseteq S \times S$ is a WEB on TS $\mathscr{M} = \langle S, R, L\rangle$ iff

(1) B is an equivalence relation on S;

(2) $\langle \forall s, w \in S{::}sBw \rightarrow L(s) = L(w)\rangle$;

(3) There exist functions $erankl : S \times S \rightarrow \mathbb{N}$, $erankt : S \rightarrow W$, such that $\langle W, \lessdot\rangle$ is well-founded, and

$\langle \forall s, u, w \in S :: sBw \wedge sRu \rightarrow$

(a) $\langle \exists v :: wRv \wedge uBv\rangle \vee$

(b) $(uBw \wedge erankt(u) \lessdot erankt(s)) \vee$

(c) $\langle \exists v :: wRv \wedge sBv \wedge erankl(v, u) < erankl(w, u)\rangle\rangle$.

Definition 5 (see [25]). Let $\mathscr{M} = \langle S, R, L\rangle$, $\mathscr{M}' = \langle S', R', L'\rangle$, and $r : S \rightarrow S'$. We say that $\mathscr{M}$ is a WEB refinement of $\mathscr{M}'$ with respect to refinement map r, written $\mathscr{M} \approx_r \mathscr{M}'$, if there exists a relation, B, such that $\langle \forall s \in S :: sBr(s)\rangle$ and B is a WEB on the TS $\langle S \uplus S', R \uplus R', \mathscr{L}\rangle$, where $\mathscr{L}(s) = L'(s)$ for s an S' state and $\mathscr{L}(s) = L'(r(s))$ otherwise.

In the above definitions, $\mathscr{M}$ is the implementation TS and and $\mathscr{M}'$ is the specification TS. Informally, to prove a WEB refinement, we need to show that every transition of the implementation TS matches a transition of the specification TS (case (a)) or it is a stuttering transition (case (b)), meaning that both the implementation state and its successor match the same specification state. Case (c) corresponds to stutter on the specification side and this is not relevant for our verification methodology as our specification is very simple (with only 10 transitions) and will not stutter with respect to the low-level object code controller TS. Rank functions are employed to distinguish stutter from deadlock (infinite stutter). Eventually, the implementation should cease stuttering and make progress. If this does not happen, then it points towards a deadlock bug in the implementation. To define rank functions, we employ a well-founded structure $\langle W, <\rangle$, where W is a set and $<$ is a binary relation on W such that there are no infinitely decreasing sequences on W, with respect to $<$. We employ the well-founded structure consisting of the set of natural numbers and less than operator on the naturals ($\langle \mathbb{N}, <\rangle$). The value of the rank function should decrease when the implementation stutters.

The very nice property of WEB refinement is that it is enough to reason about single transitions of the implementation and specification to establish a correctness proof. This is easy to do on the specification side, as the specification has only 10 transitions, whereas the object code control program TS can have millions of transitions. Therefore, we employ a decision procedure (SMT solver) to check the WEB refinement proof obligations. There are several challenges to applying an SMT solver for this problem. The first challenge is that the WEB refinement definition cannot be encoded in a decidable fragment of first-order logic and hence cannot be directly checked using an SMT solver. We overcome this challenge by exploiting the fact that the specification CTTS is known. We use the specification to strengthen the WEB refinement definition to a decidable set of proof obligations, which are described subsequently in this section. The correctness of the proof obligations is given by Theorem 6.

The second challenge is that of reachability. The WEB refinement proof obligations need only to be checked for the reachable states of the implementation. If we consider all the states (including states which are not reachable from the initial states), this would lead to spurious counterexamples, making verification very hard and probably intractable. Hence, as part of our verification methodology, we have also derived an invariant property that should be satisfied by the implementation. Invariant properties are those that are satisfied only by reachable states of the implementation and hence provide a useful mechanism to identify the reachable states of the implementation for the SMT solver. The invariant property is given below:

$$\begin{aligned}
&\{\{r(w) = s_0 \wedge (w \cdot td_v \leq \text{AEI})\}\\
&\quad \vee \{r(w) = s_0 \wedge (w \cdot td_a = w \cdot td_v)\}\\
&\quad \vee \{r(w) = s_1 \wedge (w \cdot td_a \leq \text{AVI})\}\\
&\quad \vee \{r(w) = s_2 \wedge (w \cdot td_v \leq \text{PWV})\}\\
&\quad \vee \{r(w) = s_2 \wedge (w \cdot td_v \geq \text{AEI})\}\\
&\quad \vee \{r(w) = s_3 \wedge (w \cdot td_a \leq \text{AVI})\}\\
&\quad \vee \{r(w) = s_4 \wedge (w \cdot td_a \leq \text{PWA})\}\\
&\quad \vee \{r(w) = s_5 \wedge (w \cdot td_v \leq \text{AEI})\}\}.
\end{aligned} \tag{13}$$

In the above w is an implementation state. The invariant uses the refinement map function r, which we define as a function that projects the values AS, AP_a, VS, VP, and AP_d from the implementation state to give a specification state. The invariant stipulates that the reachable states of the implementation will map to one of the specification states under the refinement map. Also, the object code control program will require two counters we call td_a and td_v that keep track of time that has passed since the last atrial and ventricle event, respectively. $w \cdot td_a$ and $w \cdot td_v$ indicate the counters td_a and td_v in the implementation state w. We can deduce from the pacemaker specification and the clinical values of derived and fundamental timing cycles of pacemaker that all the transitions of the controller are always dependent on the value of only one of the two counters. An active counter at any state is the counter, based on whose value the transitions will be made. The invariant also gives the permissible range for the active counter at each state. The permissible ranges are given using constants AEI, AVI, PWV, and PWA. In the TTS specification $\mathscr{M}_{\mathscr{P}\mathscr{M}}$, these constants correspond to time. However, when these constants are used in the invariant and proof obligations that follow, they are integer constants that still define the same time constants, but in terms of number of clock cycles of the microcontroller. Hence, their value will depend on the clock rate of the microcontroller that is used.

Next we derive the proof obligations. The pacemaker specification (Figure 2) is nondeterministic. For the pacemaker specification, we need 16 proof obligations, where 10 proof obligations represent the nonstuttering cases (which correspond to the transitions of the specifications) and the other 6 proof obligations represent the stuttering cases, one for each state of the specification. In the proof obligations, w is an implementation state and v is its successor (implementation is also nondeterministic). A_{in} and V_{in} correspond to the inputs to the pacemaker from the atrium and ventricle, respectively. A_{in} and V_{in} are typically implemented as external interrupts in the controller. PF01–PF06 give the proof obligations corresponding to the stuttering cases. When stutter occurs, we have to show that a witness rank function decreases. We have six stutter cases for six states of the specification.

PF01:

$$\begin{aligned}
&[(r(w) = s_0) \wedge (\text{ARP} \leq w \cdot td_v \leq \text{AEI} - 1)\\
&\quad \wedge (A_{\text{in}} = 0)] \longrightarrow (r(v) = s_0).
\end{aligned} \tag{14}$$

PF02:

$$\begin{aligned}
&[(r(w) = s_1) \wedge (w \cdot td_a \leq \text{AVI} - 1) \wedge (V_{\text{in}} = 0)\\
&\quad \wedge (w \cdot td_v \geq \text{VRP})] \longrightarrow (r(v) = s_1).
\end{aligned} \tag{15}$$

PF03:

$$[(r(w) = s_2) \wedge \neg (w \cdot td_v = \text{PWV})] \longrightarrow (r(v) = s_2). \tag{16}$$

PF04:

$$\begin{aligned}
&[(r(w) = s_3) \wedge (w \cdot td_a \leq \text{AVI} - 1) \wedge (V_{\text{in}} = 0)\\
&\quad \wedge (w \cdot td_v \geq \text{VRP})] \longrightarrow (r(v) = s_3).
\end{aligned} \tag{17}$$

PF05:

$$[(r(w) = s_4) \wedge \neg (w \cdot td_a = \text{PWA})] \longrightarrow (r(v) = s_4). \tag{18}$$

PF06:

$$\begin{aligned}&\left[\left(r\left(w\right)=s_5\right)\wedge\left(\mathrm{ARP}\leq w\cdot td_v\leq\mathrm{AEI}-1\right)\right.\\&\quad\left.\wedge\left(A_{\mathrm{in}}=0\right)\right]\longrightarrow\left(r\left(v\right)=s_5\right).\end{aligned}\tag{19}$$

We define the rank of an implementation state w as the difference between the maximum value (max) the active counter can take at that state and the current value of the counter. When counter = max, the implementation should make progress with respect to the specification. Otherwise the implementation stutters. Rank_a is the rank for the states where the active counter is $w\cdot td_a$ and Rank_v is the rank for the states where the active counter is $w\cdot td_v$. Note that, based on the invariant property, Rank_a and Rank_v can be combined into a single rank function for all the implementation states:

$$\begin{aligned}\mathrm{Rank}_a&:\mathrm{rank}\left(w\right)=\max-w\cdot td_a,\\\mathrm{Rank}_v&:\mathrm{rank}\left(w\right)=\max-w\cdot td_v.\end{aligned}\tag{20}$$

PF07–PF16 give the proof obligations corresponding to the nonstuttering cases.

PF07:

$$\begin{aligned}&\left[\left(r\left(w\right)=s_0\right)\wedge\left(\mathrm{ARP}\leq w\cdot td_v\leq\mathrm{AEI}-1\right)\right.\\&\quad\left.\wedge\left(A_{\mathrm{in}}=1\right)\right]\longrightarrow\left(r\left(v\right)=s_1\right).\end{aligned}\tag{21}$$

PF08:

$$\left[\left(r\left(w\right)=s_0\right)\wedge\left(w\cdot td_v=\mathrm{AEI}\right)\right]\longrightarrow\left(r\left(v\right)=s_2\right).\tag{22}$$

PF09:

$$\begin{aligned}&\left[\left(r\left(w\right)=s_1\right)\wedge\left(w\cdot td_a\leq\mathrm{AVI}-1\right)\wedge\left(V_{\mathrm{in}}=1\right)\right.\\&\quad\left.\wedge\left(w\cdot td_v\geq\mathrm{VRP}\right)\right]\longrightarrow\left(r\left(v\right)=s_5\right).\end{aligned}\tag{23}$$

PF10:

$$\begin{aligned}&\left[\left(r\left(w\right)=s_1\right)\wedge\left(w\cdot td_a=\mathrm{AVI}\right)\wedge\left(td_v\geq\mathrm{URI}\right)\right]\\&\quad\longrightarrow\left(r\left(v\right)=s_4\right).\end{aligned}\tag{24}$$

PF11:

$$\left[\left(r\left(w\right)=s_2\right)\wedge\left(w\cdot td_v=\mathrm{PWV}\right)\right]\longrightarrow\left(r\left(v\right)=s_3\right).\tag{25}$$

PF12:

$$\begin{aligned}&\left[\left(r\left(w\right)=s_3\right)\wedge\left(w\cdot td_a\leq\mathrm{AVI}-1\right)\wedge\left(V_{\mathrm{in}}=1\right)\right.\\&\quad\left.\wedge\left(w\cdot td_v\geq\mathrm{VRP}\right)\right]\longrightarrow\left(r\left(v\right)=s_5\right).\end{aligned}\tag{26}$$

PF13:

$$\begin{aligned}&\left[\left(r\left(w\right)=s_3\right)\wedge\left(w\cdot td_a=\mathrm{AVI}\right)\wedge\left(td_v\geq\mathrm{URI}\right)\right]\\&\quad\longrightarrow\left(r\left(v\right)=s_4\right).\end{aligned}\tag{27}$$

PF14:

$$\left[\left(r\left(w\right)=s_4\right)\wedge\left(w\cdot td_a=\mathrm{PWA}\right)\right]\longrightarrow\left(r\left(v\right)=s_0\right).\tag{28}$$

PF15:

$$\begin{aligned}&\left[\left(r\left(w\right)=s_5\right)\wedge\left(\mathrm{ARP}\leq w\cdot td_v\leq\mathrm{AEI}-1\right)\right.\\&\quad\left.\wedge\left(A_{\mathrm{in}}=1\right)\right]\longrightarrow\left(r\left(v\right)=s_1\right).\end{aligned}\tag{29}$$

PF16:

$$\left[\left(r\left(w\right)=s_5\right)\wedge\left(w\cdot td_v=\mathrm{AEI}\right)\right]\longrightarrow\left(r\left(v\right)=s_2\right).\tag{30}$$

Note that the invariant guarantees that PF01–PF16 cover all reachable states of implementation. The correctness of the proof obligations is given by the following theorem.

Theorem 6. *Let $\mathscr{M}'=\mathscr{M}_{\mathrm{PM}}$. Let $\mathscr{M}$ be an implementation of $\mathscr{M}'$. $\mathscr{M}$ is WEB refinement of $\mathscr{M}'$ if every transition of $\mathscr{M}$ satisfies one of the following, PF01–PF16, and if every nonstuttering proof obligation (PF07–PF16) is satisfied by at least one transition of $\mathscr{M}$.*

Proof. For an implementation (object code) to be a WEB refinement of a specification (CTTS model), as per the definition of WEB refinement, every transition of the implementation has to match a transition of the specification and vice versa, up to stuttering. To prove the above theorem, we use a proof by exhaustion (or proof by cases). First, we show that the cases are exhaustive; that is, all the transitions of the implementation and specification are accounted for. PF07–PF16 account for each of the specification transitions. The implementation states are characterized by the invariant. States that are not sensitive to A_{in} or V_{in} have one outgoing transition. States that are sensitive to A_{in} or V_{in} have two outgoing transitions depending on the value of the input. Thus, the invariants along with the values of A_{in} and V_{in} characterize all the transitions of the implementation. Each proof obligation (PF01–PF16) corresponds to a subset of the implementation transitions. The union of the set of implementation transitions covered by each of the proof obligations PF01–PF16 is equal to the set of all transitions of the implementation. Second, we give a proof of each of the cases. Each of the proof obligations PF07–PF16 satisfies case (a) of Definition 4 (nonstuttering transitions). Each of the proof obligations PF01–PF06 satisfies case (b) of Definition 4 (stuttering transitions). Therefore, if the proof obligations are satisfied by an implementation TS, it follows that the implementation TS is a WEB refinement of the pacemaker specification $\mathscr{M}_{\mathrm{PM}}$. □

6. Experimental Results

We applied our verification methodology to the object code of a DDD mode pacemaker control program implemented on an ARM Cortex M3 based NXP LPC 1768 microcontroller. The program uses two peripheral interrupt-driven timers of the LPC1768 to implement the two timelines t_a and t_v. Timer0 is used as t_v with four match registers T0MR0–T0MR3 having values of ARP, AEI, VRP, and URI, respectively. Similarly Timer1 is used as t_a with one match register T1MR0 having value of AVI. Whenever a timer reaches a value equal to a value in any of its match registers, an internal interrupt is generated. The pacemaker receives two inputs, which are the atrial sense and the ventricle sense. These inputs are implemented using external interrupts of the LCP1768. We estimate that the object code corresponding to the control program has over 2 million transitions.

To check the correctness of the object code control program, we use the WEB refinement proof obligations. We used the z3 [8] SMT solver for verification. The input language to the z3 solver is the SMT-LIB language [28]. The verification process involves four high-level steps. The first step is to model the object code control program as a TS in the SMT-LIB language. This was achieved by encoding each instruction as a function in the SMT-LIB language (called instruction functions). The instruction functions essentially specify how the instruction modifies the program state. Each of the instructions and hence instruction functions captures a set of transitions of the object code. The transitions corresponding to all the instructions in the program thus give all the transitions of the TS model of the object code program. The second step is to compute the preconditions and postconditions for each instruction. Preconditions and postconditions are predicate conditions that program states preceding and succeeding an instruction must satisfy, respectively. The preconditions and postconditions essentially determine the set of states of program preceding and succeeding an instruction. The third step is to check that each instruction function satisfies at least one of the proof obligations PF01–PF16. This was checked using the z3 solver [8]. If an instruction function did not satisfy any of the proof obligations, this points to a bug. Finally (fourth step), we want to ensure that all the nonstuttering proof obligations (PF07–PF16) were satisfied by at least one instruction function. If there is a nonstuttering proof obligation that was not satisfied by any instruction function, this indicates that there are behaviors of the specification that are not captured by the implementation and also point to a bug in the implementation.

The first, third, and fourth steps of the verification process can be automated. Automation of the first step can be achieved using a tool that can synthesize the instruction functions in SMT-LIB language from the object code. Currently, there are no available tools that can perform such translations. However, we are developing a tool that can handle a subset of the instructions of the ARM Cortex M3 microprocessor. Note that if there are bugs in the translation tool from object code to SMT-LIB, these bugs will generate incorrect instructions functions that will be caught during the verification process. Such bugs will raise spurious counterexamples as the bugs are due to the translation process and not anomalies of the object code. The third step can be automated by running a loop through PF01–PF16 for each instruction function. Each iteration of the loop would call the z3 solver to check if the instruction function satisfies one of the proof obligations PF01–PF16. The fourth step can be automated along with the third step by using flag variables that track if each of the proof obligations PF07–PF16 was satisfied by at least one instruction function.

The pacemaker control program was modeled using 224 instruction functions. Each instruction function required a verification check. Therefore, the proof required 224 verification checks using the z3 solver. Each of the verification checks was completed in less than one second. During verification, we also found a number of functional bugs in the object code. We describe two of the bugs that we found.

Bug1. Pins 3 and 1 of PORT1 of LPC1768 are used for AP and VP. PORT1 is controlled by the FIO1SET and FIO1CLR registers, which are used to set and clear the pin values, respectively. The FIO1CLR register was being updated incorrectly causing the program state to transition incorrectly. Specifically, the program was transitioning from s_3 to s_0 to s_4, when it should be transitioning from s_3 to s_4 directly. The bug was found and fixed. This bug may not be easy to find using testing because the program still seems to behave correctly even though it visits the state s_0 temporarily. However, when the buggy program reaches state s_0, if an external interrupt occurs in this state, the program will react as if it is in state s_0 instead of state s_4.

Bug2. The IO2IntStatR register contains the current status of external interrupts. A value of 1 or 2 of IO2IntStatR indicates that an AS or VS has occurred, respectively. The bug manifests when a VS is followed by an AS. In this case, the IO2IntStatR register value changes from 1 to 3, which is incorrect and is indicating that both an AS and a VS have occurred. Thus the source of the external interrupt is misinterpreted as AS instead of VS. The reason for the bug is that the interrupt status in the IO2IntStatR was not cleared after the occurrence of an AS. This bug was found and fixed.

7. Conclusions

We have developed a methodology for checking the functional correctness of DDD mode pacemaker controllers. Our methodology is targeted at the object code of the controller, which directly corresponds to the processor instructions executed by the microcontroller embedded in the device. The verification methodology is based on the set of safety requirements given in the Boston Scientific clinical literature on pacemakers [29]. Boston Scientific is a leading manufacturer and seller of pacemakers and several other medical solutions [30]. The values used for the critical timing cycles of DDD mode pacemakers are obtained from the actual clinical settings [31]. The Boston Scientific requirements are formalized and presented in [17] based on the authentic clinical literature [29, 32]. In [13] the same set of requirements are modeled and verified in UPPAAL. We have developed a CTTS model that captures all the Boston Scientific safety requirements. Our goal in developing this CTTS specification model is to use it for verification of the object code of real world pacemaker software controller. Our work is unique because it is the first formal verification methodology targeted at verification of safety of object code for DDD mode pacemakers. Our verification methodology was used to efficiently verify a control program with over two million transitions against the CTTS specification. The methodology constituted an invariant that captures the set of reachable states of a pacemaker control program, and a set of proof obligations that when verified guarantee the safety of the control program. Both the invariant and the proof obligations were developed based on the CTTS specification. Pacemaker control being a real-time system has both functional and timing requirements. Our specification CTTS captures both functional and timing requirements. However, in this paper, we have focused on functional verification of object code control programs. For future work, we plan to extend our methods using the theory of timed WEB refinements [33] to address the verification of timing requirements as well.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

References

[1] R. Jetley, S. P. Iyer, and P. L. Jones, "A formal methods approach to medical device review," *Computer*, vol. 39, no. 4, pp. 61–67, 2006.

[2] I. Lee, G. J. Pappas, R. Cleaveland et al., "High-confidence medical device software and systems," *Computer*, vol. 39, no. 4, pp. 33–38, 2006.

[3] W. H. Maisel, M. O. Sweeney, W. G. Stevenson, K. E. Ellison, and L. M. Epstein, "Recalls and safety alerts involving pacemakers and implantable cardioverter-defibrillator generators," *The Journal of the American Medical Association*, vol. 286, no. 7, pp. 793–799, 2001.

[4] http://www.fda.gov/MedicalDevices/Safety/ListofRecalls/.

[5] E. M. Clarke and J. M. Wing, "Formal methods: state of the art and future directions," *ACM Computing Surveys*, vol. 28, no. 4, pp. 626–643, 1996.

[6] L. De Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," *Communications of the ACM*, vol. 54, no. 9, pp. 69–77, 2011.

[7] L. De Moura and G. O. Passmore, "The strategy challenge in SMT solving," in *Automated Reasoning and Mathematics*, pp. 15–44, Springer, Berlin, Germany, 2013.

[8] http://z3.codeplex.com/.

[9] A. W. Chow and A. E. Buxton, *Implantable Cardiac Pacemakers and Defibrillators: All You Wanted to Know*, John Wiley & Sons, 2008.

[10] S. S. Barold, R. X. Stroobandt, and A. F. Sinnaeve, *Cardiac Pacemakers and Resynchronization Step by Step: An Illustrated Guide*, John Wiley & Sons, 2010.

[11] L. A. Tuan, M. C. Zheng, and Q. T. Tho, "Modeling and verification of safety critical systems: a case study on pacemaker," in *Proceedings of the 4th IEEE International Conference on Secure Software Integration and Reliability Improvement (SSIRI '10)*, pp. 23–32, IEEE, June 2010.

[12] A. O. Gomes and M. V. M. Oliveira, "Formal specification of a cardiac pacing system," in *FM 2009: Formal Methods*, pp. 692–707, Springer, Berlin, Germany, 2009.

[13] Z. Jiang, M. Pajic, S. Moarref, R. Alur, and R. Mangharam, "Modeling and verification of a dual chamber implantable pacemaker," in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 7214 of *Lecture Notes in Computer Science*, pp. 188–203, Springer, Berlin, Germany, 2012.

[14] Z. Jiang, M. Pajic, R. Alur, and R. Mangharam, "Closed-loop verification of medical devices with model abstraction and refinement," *International Journal on Software Tools for Technology Transfer*, vol. 16, no. 2, pp. 191–213, 2014.

[15] Z. Jiang, M. Pajic, and R. Mangharam, "Model-based closed-loop testing of implantable pacemakers," in *Proceedings of the 2nd International Conference on Cyber-Physical Systems (ICCPS '11)*, pp. 131–140, IEEE, Chicago, Ill, USA, April 2011.

[16] Z. Jiang, M. Pajic, and R. Mangharam, "Cyber-physical modeling of implantable cardiac medical devices," *Proceedings of the IEEE*, vol. 100, no. 1, pp. 122–137, 2012.

[17] M. Pajic, Z. Jiang, I. Lee, O. Sokolsky, and R. Mangharam, "From verification to implementation: a model translation tool and a pacemaker case study," in *Proceedings of the 18th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS '12)*, pp. 173–184, April 2012.

[18] I. Ibrahim, "Implantable medical devices employing capacitive control of high voltage switches," US Patent 5,178,140, January 1993, http://www.google.co.uk/patents/US5178140.

[19] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," *International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1-2, pp. 134–152, 1997.

[20] G. Behrmann, A. David, K. G. Larsen, P. Pettersson, and W. Yi, "Developing UPPAAL over 15 years," *Software: Practice and Experience*, vol. 41, no. 2, pp. 133–142, 2011.

[21] S. Li, S. Balaguer, A. David, K. G. Larsen, B. Nielsen, and S. Pusinskas, "Scenario-based verification of real-time systems using UPPAAL," *Formal Methods in System Design*, vol. 37, no. 2-3, pp. 200–264, 2010.

[22] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Logics of Programs*, vol. 131, pp. 52–71, Springer, Berlin, Germany, 1982.

[23] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.

[24] P. Manolios, "A compositional theory of refinement for branching time," in *Correct Hardware Design and Verification Methods*, pp. 304–318, Springer, Berlin, Germany, 2003.

[25] P. Manolios, *Mechanical verification of reactive systems [Ph.D. thesis]*, University of Texas at Austin, Austin, Tex, USA, 2001.

[26] P. Manolios and S. K. Srinivasan, "Automatic verification of safety and liveness for pipelined machines using WEB refinement," *ACM Transactions on Design Automation of Electronic Systems*, vol. 13, no. 3, p. 45, 2008.

[27] P. Manolios and S. K. Srinivasan, "A refinement-based compositional reasoning framework for pipelined machine verification," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 4, pp. 353–364, 2008.

[28] *The Satisfiability Modulo Theories Library*, 2013, http://www.smtlib.org/.

[29] B. Scientific, *Pacemaker System Specification*, Boston Scientific, 2007.

[30] Boston Scientific, http://bostonscientific.com/.

[31] *The Compass—Technical Guide to Boston Scientific Cardiac Rhythm Management Products*, 2007.

[32] S. S. Barold, R. X. Stroobandt, and A. F. Sinnaeve, *Cardiac Pacemakers Step-by-Step: An Illustrated Guide*, John Wiley & Sons, 2008.

[33] M. A. L. Dubasi, S. K. Srinivasan, and V. Wijayasekara, "Timed refinement for verification of real-time object code programs," in *Verified Software: Theories, Tools and Experiments*, D. Giannakopoulou and D. Kroening, Eds., vol. 8471 of *Lecture Notes in Computer Science*, pp. 252–269, Springer, 2014.