# FORMAL VERIFICATION TECHNIQUES FOR SAFETY CRITICAL MEDICAL DEVICE SOFTWARE CONTROL

A Dissertation
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Sana Shuja

In Partial Fulfillment of the Requirements
for the Degree of
DOCTOR OF PHILOSOPHY

Major Department:
Electrical and Computer Engineering

May 2016

Fargo, North Dakota

# NORTH DAKOTA STATE UNIVERSITY

Graduate School

---

**Title**

FORMAL VERIFICATION TECHNIQUES FOR SAFETY CRITICAL MEDICAL DEVICE SOFTWARE CONTROL

---

**By**

Sana Shuja

---

The supervisory committee certifies that this dissertation complies with North Dakota State University's regulations and meets the accepted standards for the degree of

DOCTOR OF PHILOSOPHY

SUPERVISORY COMMITTEE:

Dr. Sudarshan K. Srinivasan
Chair

Dr. Dharmakeerthi Nawarathna

Dr. Na Gong

Dr. Yechun Wang

Approved:

16 May 2016
Date

Dr. Scott C. Smith
Department Chair

# ABSTRACT

Safety-critical medical devices play an important role in improving patients health and lifestyle. Faulty behaviors of such devices can cause harm or even death. Often these faulty behaviors are caused due to bugs in software programs used for digital control of the device. 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 these medical devices. Our methodology is based on the theory of Well-Founded Equivalence Bisimulation (WEB) refinement, where both formal specifications and implementations are treated as transition systems. First, we present formal specification model for the medical device. Second, we develop correctness proof obligations that can be applied to validate object code programs used in these devices. Formal methods are not widely employed for the verification of safety critical medical devices. However, using our methodology we were able to bridge the gap between two very important phases of software life cycle: specification and verification.

# ACKNOWLEDGEMENTS

I would like to thank Dr. Sudarshan K. Srinivasan, Dr. Na Gong, Dr. Dharmakeerthi Nawarathna, and Dr. Yechun Wang for serving in my graduate committee. Dr. Sudarshan K. Srinivasan is my Ph.D. Advisor, and I am very thankful for guidance, and wisdom he gave me throughout my research. He has shown me the true satisfaction of research, and I enjoyed many discussions we had on the subjects of formal verification.

Finally, I like to thank my family and friend Shaista Jabeen for their relentless support in every way.

# DEDICATION

To my family.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# 1. INTRODUCTION

## 1.1. Safety Critical Medical Devices

A medical device is any instrument or apparatus that is used to diagnose, treat or cure any medical condition. These devices are used by patients to improve their quality of life and lifestyle. A medical device can be a simple apparatus like a thermometer or an intelligent system like heart lung machines, infusion pumps and pacemakers. There are so many ailments in todays world whose cure is not possible without the use of a medical device. For example heart lung machine is used to act as heart and lung in order to maintain the circulation of blood and oxygen during surgery. Pacemakers are used to treat irregular heart rhythms. Infusion pump are used to inject medicine in a patient's body. There are several medical devices that are responsible for keeping people alive and healthy. In a nutshell, for some people life is impossible without a medical device.

Safety critical device is meant to have corrected functioning in every possible situation. If in some case the device has a bug or error in it, it can lead to serious consequences like environmental harm or even death. Some of the important safety critical systems are in the scope of medicine, automotive, aviation and recreation. For example parachute is an example of a safety critical system which if fail to open at the right time can lead to death. Therefore, simple looking devices are actually safety critical in nature.

### 1.1.1. Motivation

The major objective of focusing on medical devices for this research is the reason that medical devices are inevitable for the patient dependent on the device. For our research we have chosen two medical devices: pacemaker and insulin pump. There are two reasons behind choosing these devices:

- The U.S Food and Drug Administration (FDA) [1] has issued 38 class I pacemaker recalls during the last decade. There are currently 169,184 pacemaker units that have been recalled due to software related issues. Class I recall indicates that the continuous use of the device can lead to serious health consequences or even death. Similarly since 2001 FDA has issued 54 Class I recalls on infusion pumps due to software issues, a number of these recalls targeted

at insulin pumps. From 2005 to 2009, over 56,000 adverse-event reports and more than 500 deaths were recorded by the FDA due to infusion pump malfunction [69].

- Pacemakers are small implantable devices that are employed to treat bradycardia (slow heart rate). Pacemaker aids in maintaining normal pace of heart, which if not treated can lead to death. Insulin pumps are used to administer regular doses of insulin in diabetes patients. If the insulin pump is not injecting insulin at the required rate for some time, it can lead to hypoglycemia or hyperglycemia which if left untreated can lead to coma or death [28].
- Formal methods are not widely used for verification in medical device industry.

This rationale makes these safety critical devices ideal and valuable for applying effective verification methodologies. It is to be noted that the medical device industry does test their devices before marketing a product. However, despite of testing the medical device, every year there are considerable amount of recalls issued by the FDA. The recalls issued related to the medical devices are both hardware and software related. However, we have employed formal verification techniques for the software encoded in these devices.

### 1.1.2. Why Not Testing

The recent advancements in medical device industry shows an enormous progress in device development. Where, hundreds of million dollars are spent each year for medical device design, testing and verification. In a traditional software life cycle a device is tested for errors after it is manufactured. The process of testing is useful to identify the behavior of the device against certain inputs. Testing is performed by presenting the device with a set of pre-defined inputs, and the output is accessed against the set of expected outputs. If the system fail to provide the correct response the device is turned down. Changes in the design are being made and the entire process is repeated again. This not only consumes time but considerable amount of resources as well. Testing can be employed alone for every day appliances, desktops and other recreational embedded devices, but it is inadequate for safety critical devices. Some of the limitation of testing are:

- It cannot guarantee that the tested system is bug free.
- It does not identify the real reasons and causes responsible to introduce bugs.
- It cannot guarantee that the system function properly under all possible circumstances.

### 1.1.3. Why Verification

Verification is the process of checking the accuracy of a system in any given circumstances. It is performed in order to find hard to find corner case bugs, which testing can fail to capture. Verification is another important phase of software design cycle. It can be applied before or after the device production. A traditional software life cycle has five stages: requirement analysis, design, implementation, testing and deployment. Verification can be performed in the deployment stage. But the true essence of verifying a system is to incorporate verification techniques from the requirement analysis to the deployment of the device. The problem here is the variance in nature of different stages of the software life cycle. For example specification engineers gather the requirements for a device, analyze the requirements and devise specifications. The device is then constructed by design engineers and then tested by testing engineers. These group of engineers have their expertise in the phases of the traditional design cycle. However, formal verification techniques are highly mathematical in nature and require substantial rigor.

### 1.1.4. Problem Statement

The use of formal verification techniques to ensure safety and correctness in a medical device software is an unaddressed problem. Some of the challenges are to deal with a very large state space, high non-determinism and stringent timing requirements. This study aims to employ formal verification techniques for validating and verifying error free operation of safety critical medical devices like pacemakers and insulin pumps.

### 1.1.5. Our Approach

The problem is approached with the concept of refinement based verification. Our approach is divided in following essential steps:

- First, we gather the requirements for the medical device. We make sure that the requirements are taken from authentic clinical literature. It is one of the most important steps of this research. If the specification is not correct or invalid it will not only dissipate the verification effort but it will also induce bugs that are extraneous.

- Second, we express the requirements as a transition system or a timed transition system whatever the requirements call for. We call this system the formal specification model of the medical device.

- Third, the transition system is modeled in a model checking tool. Model checking ensures that the system is free of deadlocks. The transition system is encoded in UPPAAL a model checker. The requirements are encoded in Compositional Tree Logic (CTL) as properties. The properties are verified against the model using UPPAAL verifier.

- Fourth, we devise invariant predicate for the system. The invariant essentially eliminates the non reachable states of the system.

- Fifth, a rank function is formulated that signifies if the implementation is progressing with repsect to the specification.

- Sixth, to bridge the gap between the implementation and speification proof obligations are introduced. They are formulated in Satisfiability Modulo Theory (SMT) [2] and discharged using an SMT solver z3[3]. These proofs form the basis for equivalence verification.

This approach is novel in a way that it incorporates safety and correctness from gathering requirements till verification.

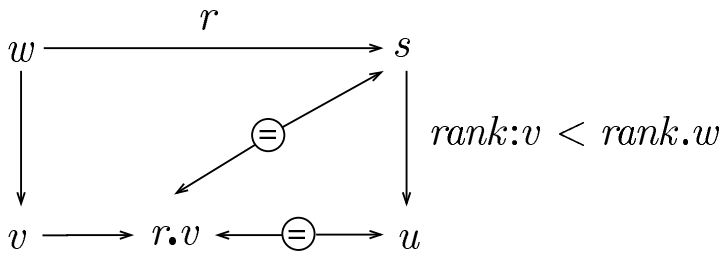


Figure 1.1. WEB refinement

### 1.1.6. Equivalence Verification

Equivalence verification is the concept of matching the requirements of the system in terms of the formal specification and match them with the implementation. We use the notion of Well Founded Equivalence Bisimulation (WEB) refinement. WEB refinement address the problem of verification in a matter that it matches the specification transition system and implementation transition system. It checks where the system progresses and where it is stalled. Both the specification and implementation should be expressed as transition systems. Specification in this study

is the set of clinical requirements for the medical device. On the other hand implementation is the object code level software embedded in the device. Safety critical medical devices have complex software thus making the implementation TS quite complex. A more detailed description on this concept is given in chapter 3. Figure 1.1 depicts the concept of WEB refinement. $w$ and $v$ are implementation states, $s$ and $u$ are specification states, *rank(u)* and *rank(v)* are the ranks of specification and implementation respectively. There is an extensive amount of work done in the area of medical device verification. However the use of formal methods for verifying safety critical medical device is not much evident. Some of the related work in the field is given in next chapter.

# 2. RELATED WORK

## 2.1. Background

Software is playing increasingly important role in medical devices which are responsible for keeping patients healthy and in some cases alive. FDA [1] has been focusing on more efficient strategies to provide bug free software for medical devices. The use of mathematical models for the development of software might be helpful, but there is no standard architecture for medical device software. Also, the need of an integrated engineering tool to model and verify clinical requirements of a life saving medical device is inevitable. There are extensive regulatory processes to ensure correct production of the device, but they are inadequate for assessing and validating the software. There are subtle errors and latent bugs that might exist in the software, which requires formal model checking techniques and exhaustive white box testing. In such cases a model based design is required against which the designer can design and testers can verify their device.

There are certain issues with high confidence, bug free and reliable design of safety critical medical devices. The technologies already available are not reliable and secure enough to produce highly distributed medical device software with assurance of patient safety. Treatment systems and diagnostics are advanced but they do not work well together. These systematic inefficiencies in health care delivery introduces unavoidable medical errors that degrade patient health .The medical devices may need to communicate over a distributed network. These networks propagate messages based on the critical conditions and requests which serve as a feedback of the closed loop. Research is required to enable fast and reliable technologies to model and verify such medical systems. In medical practice, the role of modeling and simulation will improve device and eventually quality of patient care. Medical devices are essentially embedded systems, of which software is a pivotal part. If the device is safety critical, rigorous software development methods are required to ensure a reliable and secure patient health.

As the technology is advancing, and the size of wearable and implantable medical devices is increasing, software is becoming more and more complicated. In such a scenario, testing alone is not sufficient and practical. Lack of formal specifications and requirements adds to this critical issue. Verification and validation of the device before it is marketed is an essential and inevitable

step towards a reliable safety critical medical device. As devices grow more complex and rely more on the software than the hardware, the existing validation and verification techniques seem isolated. The major challenge of the design of a medical device either safety critical or not, is the cross-cutting nature of design. It involves, science, computing, engineering as well as mechanics at times. There is an immense need for devising standards for the architecture and communication of device networks, while assuring quality of service and patient security. Another important challenge is to keep the system's behavior according to the expected response and not degrade at the time of unforeseen and abnormal conditions. The system should be capable of identifying and if possible repairing faults while notifying the concerned monitors.

Medical devices are not only designed as an embedded system but in certain cases are embedded in the patient's body. In patient's body these devices control and regulate critical life functions. The design of medical device should be based on authentic fundamental approach that can integrate functional, computational, and interactive designs in the presence of patient models in normal as well as abnormal conditions. Current design practice employs validation and verification techniques at the end of device design when it is too late to change the design. As medical devices become complex, it is necessary to validate the design at early stages. The verification methodologies also need to be applied to each component and formulate an integrated solution, which sufficiently reduces the risk of closed loop bugs and errors.

More effort is required to verify the real time embedded systems in a closed loop with a physical plant. Most medical devices require the patient to be in the loop, which makes the modeling of such systems more interesting. Huge amount of work has been done in modeling of device design, formal specification of the clinical requirements and use of formal methods for verification of safety critical medical devices. Different approaches have been adopted to bridge the gap between the informal medical requirements and the formal specifications for the device. These approaches can be categorized based on modeling of the medical device or the patient, modeling of the proposed formal specifications, or verification of the closed loop system with either the model of the device, or the patient or both.

## 2.2. Approaches Towards Safe and Reliable Medical Device Design

There are several approaches to seeing the challenge of designing software for a complex device whose formal specifications are not known. Various researchers have targeted this problem with methodologies and models to provide a standard architecture or formal specifications. These approaches can be classified on the basis of the techniques employed and the involvement of the feedback in the close loop.

### 2.2.1. Closed Loop Patient Modeling and Simulation

Closed loop patient modeling and simulation requires the physical plant which may be heart, pancreas or patient himself in the closed loop. Medical devices face a unique challenge in modeling of such closed loop systems is a unique challenge because of variety of patient models and high fidelity simulators for device design. As devices adapt to a variety of medical conditions, developing simulators and models for normal and abnormal conditions of the patient has become increasingly important and challenging. There is an immense need of developing models and simulators at high level of abstraction, ranging from coarse models to highly reliable simulators for model validation and virtual verification.

### 2.2.2. Verification Based on Formal Specifications and Methods

This is a good approach to bridge the gaps between informal requirements and formal specifications. It also provides theorems and proof obligations which serve as a standard for the design. Exhaustive model checking techniques and representation of requirements as properties and formulae in a theorem prover, validates and verifies the requirements against the proposed model.

### 2.2.3. Model Driven and Model Based Verification

This scheme focuses on the model of the device design or the physical plant to validate and test the device. Model based designs build the software based on their understanding of the device requirements. Whereas, model driven designs devise methodologies and algorithms to run on already established authentic models of the device and specification of the requirements. Next we describe the contributing research being done with the approaches discussed in this section.

## 2.3. Closed Loop Patient Modeling and Simulation

An extremely novel approach is devised for the modeling of implantable medical devices with the physical plant, the heart [40]. There has been no attention given towards formulating a

formal methodology or an open platform to test and verify the correct function of medical device software with the patient in the closed loop. A real time virtual heart model (VHM) has been modeled to model the electrophysiological operation of the functioning of the heart. Based on the formal specifications of pacemaker and the functioning of a human heart, timing properties of the pacemaker and heart are extracted. The timing properties help to model the functional behavior, and perform formal testing and verification of the closed loop system. Pacemaker is tested against the VHM in a closed loop for healthy heart conditions, and also by introducing common arrhythmia. The behavior of the pacemaker in this closed loop testing environment validates its correct operation.

Jian et. al [38] has taken modeling of medical devices to a new level by designing a human heart as Virtual Heart Model (VHM), which is capable of imitating the behavior of a human heart and also able to demonstrate heart arrhythmia. The timing properties of heart are devised based on the clinical settings and a timed-automata model is formed. The platform has formal and functional interfaces for validation and verification of implantable cardiac devices. VHM is capable of responding to premature stimuli and an artificial pacemaker. The study contributes to the verification of an implantable pacemaker by observing the correct response of the pacemaker against the arrhythmia introduced by the VHM. The virtual heart model is implemented on FPGA and can be tested against any commercial pacemakers available. This research can aid physicians to diagnose and devise a correct cure for their patients.

Macedo et. al [48] proposed a pragmatic incremental approach to model construction by developing a pacemaker model. Pacemaker specifications include system level requirements which affect hardware as well as software. This effort demonstrates validation of functional and timing requirements for such a cross-disciplinary medical device. A new formal model is proposed for a cardiac pacemaker system to address the grand challenge problem [67]. VDM tool has been employed which allows co-simulation of Matlab/Simulink model of heart with controlled events of the pacemaker. This research emphasizes more on validation by animation then verification by proof.

Verification of closed loop system is a recent and important challenge as more and more people are using implantable medical devices. There is currently no methodology for formal validation and verification of medical device software especially with the patient in loop. A timed

automata based Virtual heart Model (VHM) is presented to act as a platform for pacemaker software validation and verification [35]. VHM provides closed loop interaction with both medical device hardware and software. Pacemaker involves a complex mode switch operation. VHM can be used with the closed loop operation of a pacemaker to validate the bug free functionality of the pacemaker software in this mode. The case study focuses on patients with supraventricular tachycardia, and correct pacemaker operation is demonstrated as the pacemaker switches from one rhythm management algorithm to another.

Implantable cardiac devices respond according to the timing information from the heart. In a closed loop model when the implantable device is interfacing with the heart, it is assumed that the leads are always in place, also pacing in one chamber does not interfere with pacing in another. If this assumption is not true, the patient can suffer severe consequences or even death. For that matter, spatial and temporal properties of electrical conduction system of the heart are investigated using virtual heart model [36]. These spatial properties of the heart are utilized to model the sensing mechanism using clinical cases to show validity of the proposed sensing model. This closed loop evaluation of pacemaker also allows functional testing of pacemaker software. The interface aids in development of new algorithms to cure various arrhythmias and also serves as a tool prospective research.

The challenge of a design for a bug free medical device software is addressed in [38]. In order to model the normal electrophysiological operation of heart and various arrhythmias, a methodology is proposed to construct a timed automata model by extracting timing properties of the heart. The proposed platform provides functional and formal interfaces for the validation and verification. The heart model is capable of generating expected response to premature stimuli and pacemaker signals for a variety of common arrhythmia. Having a closed loop with pacemaker, VHM can be paced and synchronized based on the irregular rhythms introduces in the model. VHM has been implemented on FPGA and is tested against a real pacemaker. This is a productive step towards speeding up medical device certification and safe operation.

### 2.4. Verification Based on Formal Specifications and Methods

Formal model is an intelligent way of modeling a safety critical system. The requirement for a safety critical system to be dead-lock and bug free is inevitable. Tuan et. al [65] proposed a formal model of a pacemaker. The behavior and modeling of the pacemaker is done using real time

formalism. In order to check the critical properties like dead lock and heart time limits a model checker is employed. Process Analysis Tool-kit (PAT) [47] checks for the correctness of the behavior of the pacemaker model. This work is capable of serving as a specification for a real pacemaker implementation.

Jiang et. al [57] emphasized on formal modeling of real time systems and model's verification at an early stage. Model-Driven Design (MDD) for cyber physical systems support such design procedures. The main contribution of this research apart from the design of a model transition tool UPP2SF is the definition of formal pacemaker requirements and software specifications. The specifications are modeled as timed automata and the requirements are encoded as CTL properties. The timed automata model is translated using the transition tool to a state flow chart in Simulink which is automatically converted into C and tested on FPGA against the formal specifications defined. Conformance testing approach demonstrates that the translation procedure preserves the behavior of the closed loop implantable pacemaker.

A dual chamber implantable pacemaker has been studied as a case study to express the software specifications of the implantable medical device as a timed automata model in UPPAAL [37]. Clinical literature Boston Scientific describes the specifications and algorithms used to model the system in UPPAAL. Safety properties expressed in ATCTL* are also preserved during refinement. Closed loop system is incomplete without a physical plant, so the heart is also modeled in UPPAAL based on its physiology. CEGAR a manual Counter-Example-Guided Abstraction and Refinement framework refines the heart model based on the counter examples obtained whenever there is an unexpected or buggy behavior.

A set of general and patient specific temporal requirements for closed loop heart and pacemaker model are proposed in [39]. A close loop testing environment is formulated based on the requirements. It allows for interactive and physiologically relevant model-based test generation for basic pacemaker operations such as maintaining the heart rate and atrial-ventricle synchronization. The approach also demonstrates the flexibility and efficacy of the testing environment for timing anomalies like reentry circuits, pacemaker mode switch operation and pacemaker-mediated tachycardia. This study is an advanced step towards testing cyber-physical system with the physical plant in loop. Alur et. al [9] proposed the use of formal methods to capture the requirements and to analyze CARA infusion pump control system. Extended Finite State Machine (EFSM) is

employed to convert the informal design requirements in to a set of reference specifications. In this way the safety properties that need to be satisfied are identified and are model checked using Hermes [12]. This allows concluding the properties of the reference specification. The study develops a framework and a methodology for the integrated use of formal methods in the development process of safety critical medical devices such as CARA.

A compositional approach to verify the safety properties of a closed loop generic patient controlled infusion pump is proposed by Anitha et. al [54]. The approach cyber physical systems (CPS) which is organized into multiple abstraction layers. This work contributes to the formal reasoning, multiple modeling formalism, verification paradigms, and associated tools to verify medical CPS. The infusion system includes a patient blood oxygen level monitor whose output is constantly monitored and a controller based on that output decides when to terminate the medication. Two approaches has been combined in this work (i) Safety properties of the closed loop system using timed automata in UPPAAL [17][44] with the model of patient in loop and (ii) critical requirements elaborated on Simulink/Stateflow. The research formally unites distinct verification paradigms by controlling the system hierarchical architectural decomposition. It has shown that properties proved for the system components at the lower level of abstraction can validate more abstract models and remain satisfied for higher level of abstractions.

### 2.5. Model Driven and Model Based Verification

The use of Implantable Cardiovascular Defibrillator (ICD) has exponentially increased [60] and is increasing each year. A patient specific forward simulation model is proposed [63] to predict efficacy of the shock. The focus of the research is to develop a method to measure the ICD surface potential and to use this information for the verification of simulation. A lead selection algorithm helps to develop a surface potential mapping system with recording sites without interfering with the implantation surgery. The discharged recordings obtained are utilized to compare with similar locations to corresponding patient models. The reconstructed surface potentials are nearly error free which proves the algorithm to be effective. This study shows proper simulation for power distributions of ICD potentials.

Chen et. al proposed a model-based framework for quantitative, automated verification of pacemaker software [19]. The electrocardiogram model of Cliffard et. al [23] is adapted which is capable of generating realistic normal and abnormal heart beats. These beats produce a timed

sequence of potential signals which can behave as boundary conditions or test conditions as a pacemaker input. The timed automata model for a pacemaker presented in [32] is employed to develop a methodology for deriving the behavior of heart and the pacemaker together. The study involves checking for correct timing of pacing during bradycardia and for prevention from tachycardia for a range of realistic healthy heart as well as a real patient's heart. One additional interesting contribution is checking for noise and faulty signals which may hinder the conduction of correct sense signals. This idea is implemented using PRISM [43] a model checker and MATLAB. This study is compatible with any pacemaker model and any conditions.

Jiang et. al [32] focused on the modeling and verification of dual chamber implantable pacemaker. The requirements to model the system are taken from Boston Scientific and the model is verified in UPPAAL. This study defines a state space of the closed loop system with heart as the physical plant and introduces a heart model which non-deterministically covers all the state space. In order to ensure correct verification methodology certain unsafe regions are specified in the state space, and the closed loop system is verified against the safety algorithms from clinical literature. Pacemaker mediated tachycardia is the clinical case against which unsafe transitions are tested with proposed correction algorithms. This study contributes to model driven design and code generation from UPPAAL models.

The verification grand challenge [67] motivated researchers to develop toolkit which can verify software according to users' needs. In the field of safety critical systems especially medical devices this challenge has become eminent to create a formal model of a pacemaker based on the informal requirements described by Boston Scientific [4]. Gomes et. al proposed a formal pacemaker model using Z notation [68]. The formal specifications are validated against the informal requirements using ProofPower-Z [56]. This theorem prover formulates proofs of specification-to-model correspondence for a bug free secure system.

Arney et. al [14] presented a model-based approach to conformance testing of medical devices. The approach focuses on reference models which are employed for the application of formal methods to check software conformance and to provide a framework for rigorous testing. A reference model for Generic Patient controlled Analgesic Infusion Pump is developed and using that reference model software conformance is checked and verified. The study also develops a list of hazards and requirements, and also tests the reference model for structural and safety properties.

Pajic et. al [59] proposed a verification approach for the safety properties of closed loop medical devices. A model driven approach is employed to perform the safety analysis of the closed loop system. This approach combines the simulation based analysis of detailed continuous time model of the system in Simulink with continuous patient dynamics modeled with abstract timed-automata. The timing parameters of the Simulink model are used as constants in UPPAAL. This relationship between timing behaviors of these models shows the conservativeness of safety analysis. An important contribution is a system design model that can provide open loop safety under network failure. This study primary focus is on the use of model driven safety analysis of closed loop medical devices.

Design of a UPP2SF model tool for safety critical medical device development is an advanced step towards end to end modeling, verification, code-generation and testing process for complex software controlled embedded systems [58]. UPP2SF harness automatic conversion of verified timed automata in UPPAAL to a model that can be simulated and tested in Simulink/Stateflow. A pacemaker model in UPPAAL is translated automatically to Stateflow for simulation based testing. The tool automatically generates modular code from UPPAAL timed automata for hardware-level testing. An additional feature of UPP2SF is the estimation of worst case execution time at an early stage.

Arney et. al [15] proposes a model driven approach for validation of closed loop medical systems. A continuous time system is modeled in MATLAB and validated with simulations. Another abstract model is proposed using timed automata in a model checker UPPAAL [17] [44]. The key approach is to derive the timing parameters using Matlab simulations and then use these parameters as clocks or constants in UPPAAL. This method is based on the concept of timed lease which is employed in fault-tolerant embedded systems. Model driven engineering is used to establish a secure implementation of Patient-Controlled Analgesic (PCA) based on the generic PCA reference model [41]. The reference model is modeled as a network of timed automata in UPPAAL. FDA provides generic safety requirements for PCA, based on which the safety properties of UPPAAL model are assured. After assuring safety properties, TIMES tool [13] is employed to automatically generate platform independent code. This code along with auxiliary requirements is deployed on a real PCA pump. To assure that no safety properties are violated a test is generated to check the consistency between the reference model and the code through conformance testing.

## 2.6. Discussion and Conclusion

There should be a clearer of the requirements for the safety critical medical devices. There is a huge amount of clinical literature about various medical conditions but there has to be a standard or a reference model for each device and each mode that it can work in. Formal methods have been an effective approach to deal with the software issues of the current medical devices but without a clear set of requirements this is not worthwhile. Closed loop modeling of patient and the device is the latest uproar in the field of safety critical medical device verification, but models and software no matter how fool proof they have been carries a margin of fault which can lead to irreversible consequences or even death. On the other hand modeling a physical plant like heart can mimic human heart's operation but it cannot predict the behavior of every patient, as each individual is different. Similarly model based approach is not a productive step without a clear set of requirements towards the best solution. Whereas, model driven approaches are restricted in a way that there has to be proven, clear and relevant literature presented.

There is still room to see this challenge in a different perspective. All the studies discussed either focus on the modeling of the device, specifications of the requirements or verification of the closed loop system. There has been no attention given to the verification of the real time software operating on the device as a closed or an open loop system. Medical devices due to their nature of operation are expected to respond to every minor change in the feedback, and the feedback being the most intelligent entity (i.e. human body) can react in the most unexpected way possible. Similarly the complex nature of the device makes it hard to present software that strictly conforms to the formal specifications of the device. Efforts need to be done in devising methods, models and proof obligation which serve as a standard to validate and verify the correct and bug free run time operation of any safety critical medical device.

One significant gap in verifying approaches for safety critical medical device is the translation of software written in higher level language to low level language. Software developers design the software based on the specification. Software for embedded systems is written in high level languages like C, Verilog and VHDL etc. These high level languages are converted into low level languages called as source code. The source code is converted into machine specific object code. This translation process is done with the help of assemblers and compilers. During translation from

high abstraction to low abstraction, bugs are introduced in the system. These are the bugs that are not overlooked by the software developers but are errors introduced due to the translation process. The reason of these bugs is level of abstraction. The functions that are performed in one line in C are translated into several numbers of lines of assembly code. In order to address this problem, coherent specification model should be followed. Also, intelligent equivalence techniques like WEB refinement should be employed to match the specification and the object code (implementation). This process will not only catch bugs but will also point to the source that caused the bug.

# 3. FORMAL SPECIFICATION AND VERIFICATION OF DDD MODE PACEMAKERS

## 3.1. Background

Implantable pacemaker is a small device that is placed surgically in the chest or abdomen of the patient. It generates electrical pulses that pace a slow paced heart. A typical pacemaker is (product of Medtronic[5]) shown in Figure 3.2. Pacemaker is normally used for patients that suffer from bradycardia. Bradycardia is a disease when heart beats too slowly. Due to the slow rate of the heart, it does not pump enough blood to the body. If the slow heart beat continues for a longer period of time it can cause damage integral organs of the body. It can also lead to coma or even death in severe conditions. These facts make pacemaker safety critical, whose faulty behaviors can cause serious harm or even death. There can be different forms of faulty behaviors. It might be due to the wires not implanted properly during surgery. But in most cases it is caused due to the bugs in the software underlying the hardware. Software for embedded devices is mostly written in high level languages like C. Developers play special heed to the requirements while writing the software for such safety critical devices. But they do not consider object code while designing the software. When a higher level language like C is encoded on a microprocessor, it is compiled and converted to source code. This source code is converted into object code with the help of assemblers which are platform specific. For example on an ARM processor a specific C instruction converts into 3 lines of object code but the same line in C converts to 5 lines of assembly on an AVR processor. This translation from high to low level language introduces ambiguity or bugs with respect to the actual specification of the system.

We present a formal verification methodology [62] that can be used to check the correctness of object code programs that implement the safety-critical control functions of DDD mode pacemakers. The purpose of these proof obligations is to bridge the differences between the specification for the pacemaker and the actual implementation. ***Specification*** is the formal set of requirements for the pacemaker obtained from the clinical literature. Whereas, ***Implementation*** is the object code obtained after implementing pacemaker specification on an embedded system. For the pur-


Figure 3.1. A typical pacemaker

pose of equivalence of specification and implementation, our methodology is based on the theory of Well-Founded Equivalence Simulation (WFS) refinement, where both formal specifications and implementations 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.

### 3.2. Introduction

Pacemakers are widely used for the treatment of slow heart rate. Each year approximately 6 hundred thousand people are implanted with pacemakers. There are currently approximately 3 million people with implanted pacemaker, improving their quality of life [66]. Pacemakers are not only implanted in adults and elderly but it is also implanted in children. A normal healthy person's heart beats 50-70 times a minute. If a person's heart beats slower than this limit for a longer period of time that person suffers from bradycardia and needs a pacemaker. The heart generates electrical signals to induce heartbeat. These electrical signals decide the pace with which heart is going to beat. The heart's electrical system can become defective due to aging or other causes, leading to a slower heart rate. Such ailment need be treated using pacemakers, which are implantable in the chest thus bugs in the system cannot be tolerated. It is concluded that pacemakers are inevitable for patients with bradycardia and if they do not function properly can cause loss of life. Thus it is emphasized that pacemakers are safety critical in nature [34][45].

A control program executing on a microcontroller embedded in a pacemaker is responsible for implementing the control functions of the device. This control program is the code written in

C by the software developers or in other high level languages. For this research we have written C code for the pacemaker. 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 [49]. 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 [1]. 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 of the patient.

We present a formal verification methodology [22] 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, that is pacing in both atria and ventricles.
- *"Dual"* chamber sensing, that is sensing for signal in both atria and ventricles.
- *"Dual"* chamber activation or inhibition that is stopping or starting further pacing in both 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 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 [25] such as z3 [3].

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.

### 3.3. 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 node (SA), a set of specialized tissues located on the right atrium, it is controlled by the nervous system. This specialized tissue acts as the natural pacemaker of the heart. It is responsible for generating periodic electrical pulses. The blood is initially in the ventricles. The electrical pulses caused by the SA node contract the walls of the atria pushing the blood into the ventricles. The atrioventricular node (AV), which is a bundle of specialized tissues situated between the atria and ventricles do not allow the electrical signals to transmit to the ventricles until the ventricles are filled with blood. The bundle next to the AV node called HIS-Pukinje eventually transmits the electrical pulses to both ventricles with the aid of purkinje fibers, causing the muscles of the ventricles to contract simultaneously and thus pump the blood at a healthy pace to the entire body.

The heart rate is formed due to timely electrical signal contracting the walls of the heart and pumping blood to the body. If a patient suffers from bradycardia these electrical signals are not generated properly. The concept of a pacemaker is to generate these electrical signals externally. The mechanism of the pacemaker revolves around sensing and signaling of electrical pulses.

#### 3.3.1. Sensing

The phenomenon of detecting the electrical signal generated by the muscles of the right atrium or right ventricle is called sensing. Sensing serves as the input from the heart to the

pacemaker. If this input is received in an optimal time, no pacing is performed. But if this input is delayed pacemaker has to take over the heart and pace it on its own. If sensing is done in atria it is called atrial sensing and if done in ventricle is called ventricle sensing.

### 3.3.2. Pacing

The phenomenon of artificially inducing electrical signals in the walls of atria and ventricles is called pacing. The electrical signal makes the muscles of the heart chamber to contract and thus pump the blood. If an atrial sense is not sensed in an optimal time, atrial pacing is done in the atrium which forces the blood to enter the ventricles. Once the blood is in ventricles, and no ventricle sense signal is received for an optimal period of time, ventricle pacing signal is generated by the pacemaker, pushing the blood out of the heart and to the rest of the body. Pacing serves as the output from the pacemaker as input to the heart.

## 3.4. Interface Between Heart and Pacemaker

A DDD mode pacemaker has leads connected to the right atrium and right ventricle [20]. The interface between a heart and a DDD mode pacemaker is shown in Figure 3.2. 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 atrial 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. [16] are given below.

- **Lower rate Interval (LRI)** This interval helps to keep the heart above a certain minimum value below which it can be hazardous for the patient. LRI is the timing spell that is responsible to maintain the longest interval between a ventricle event that can be either ventricle sensing or ventricle pacing, and the subsequent ventricle paced event (VP) without superseding sensed events.

- **Ventricular Refractory Period (VRP)** VRP is initiated by a ventricle event, either ventricle sensing or ventricle pacing. During VRP, LRI cannot be initiated or reset. During

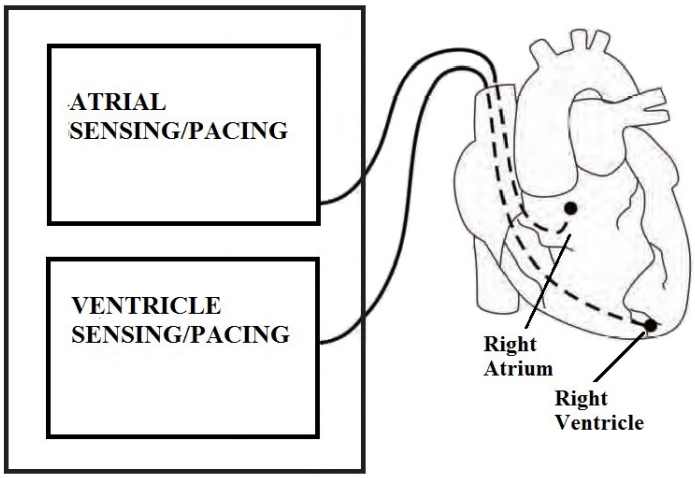


Figure 3.2. The interface between heart and DDD mode pacemaker

this period, a pacemaker does not respond to incoming signals. VRP follows each ventricle event in order to give a compensation window to ignore any noise that can cause unwanted pacemaker behavior.

- **Atrioventricle Interval (AVI)** AVI is the time interval between an atrial event (atrial sensing AS or pacing AP) and the following ventricle event (ventricle sensing VS or pacing VP). The purpose of AVI is to ensure optimal delay between atrial and ventricle events. If during atrioventricle interval, no ventricle signal has been sensed after an atrial sensed or paced event, ventricle pacing will be performed by the pacemaker.

- **Atrial Refractory Period (ARP)** ARP is the interval after a ventricular event (ventricle sensing VS or pacing VP). During this interval no atrial event can initiate a new AVI.

- **Upper Rate Interval (URI)** URI is the interval that prevents the pacemaker from pacing the ventricles too fast. It limits the ventricle pacing rate by imposing a lower limit on consecutive ventricle events either ventricle sensing VS or ventricle pacing VP.

- **Atrial Escape Interval(AEI)** AEI is the interval between a ventricle event VS or VP and the subsequent atrial pacing event (AP) with no intervening sensed events.

$$AEI = LRI - AVI$$

### 3.5. Related Work

Tuam et al. [65] 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 et. al proposed a formal specification of a pacemaker using the Z notation [29]. 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 [32][37]. 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 3.6, 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 verification methodology for object code. We use the theory of WFS refinement for this purpose. In Section 5.5, we have described why we use the WFS 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 WFS refinement. Also, we have developed a specification model that is as simple, clear and high-level as possible, so that the resulting verification methodology is efficient.

Jiang et. al [39] 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 [40] 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.

### 3.6. Formal Specification Model for DDD Mode Pacemakers

The requirements of a DDD mode pacemaker are given in [57]. These requirements are based on two time lines $t_a$ 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 a valid $A_{in}$ is detected, then the pacemaker registers an atrial sense event (AS). If a valid $V_{in}$ is detected, then the pacemaker registers a ventricle sense event (VS). Figure 3.2 shows the interface between the heart and the pacemaker. The requirements from [57] are given below.

#### 3.6.1. 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 $t_v$;

#### 3.6.2. 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 $t_v$;

#### 3.6.3. Atrial Pacing

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

#### 3.6.4. Ventricle Pacing

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

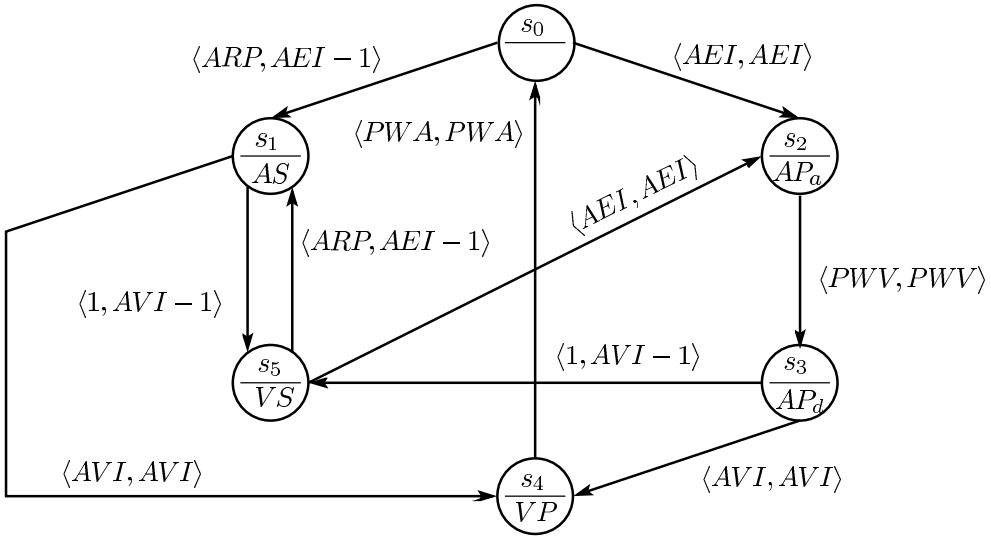


Figure 3.3. Figure depicts the specification TTS $M_{\text{PM}}$

**VP.2** VP cannot be generated within $t_v \in$ (0, URI);

**VP.3** If VS does not occur in intervals $t_a \in$ (0, AVI) and $t_v \geq$ URI, VP should occur at $t_a =$ AVI;

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

We present a formal specification model that captures the above requirements. We use timed transition systems (TTS) to model the pacemaker specification. TTS is defined as follows:
**Definition 1:** A Timed Transition System (TTS) $\mathcal{M}$ 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 $lb, ub \in \Re$. $lb$ 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 3.3. The TTS specification $\mathcal{M}_{PM} = \langle S_{PM}, R_{PM}, L_{PM}\rangle$ has 6 states:

$$S_{PM} = \{s_0, s_1, s_2, s_3, s_4, s_5\}$$

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, $\text{AP}_a$, $\text{AP}_d$, 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 $\text{AP}_a$ and $\text{AP}_d$. $\text{AP}_a$ indicates when the

pacemaker should assert an atrial pacing and $\text{AP}_d$ indicates when the pacemaker should de-assert an atrial pacing. The transition relation of the specification TTS is given below.

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

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 time line $t_a$ and the ventricle time line $t_v$, respectively, and are hence named as such. The labeling function is defined as follows:

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

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

### 3.6.5. State $\mathbf{s_0}$

$s_0$ 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 $t_v \in (0, \text{ARP})$ should be ignored (requirement AS.1 and AS.2). Also, if Ain occurs for $t_v \geq \text{ARP}$, 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 $t_v$= 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 $t_v$=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, 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.

### 3.6.6. 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, 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 $t_a$=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, 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.

### 3.6.7. 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.

### 3.6.8. 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.

### 3.6.9. States $s_2$, $s_4$

From [30], the pulse width for emergency bradychardia pacing is approximately $1.00ms \pm$ 0.02 ms. Therefore, the pulse width of both AP and VP should be $1.00ms \pm 0.02$ ms. AP is asserted in $s_2$ and de-asserted in $s_3$. Therefore, $s_2 \rightarrow s_3$ has a lower bound and upper bound of PWV = $1.00ms \pm 0.02ms$; PWV stands for Pulse Width Ventricle, as the next event is a ventricle event.

Similarly, VP is asserted in $s_4$ and de-asserted in $s_0$. Therefore, $s_4 \rightarrow s_0$ has a lower bound and an upper bound of PWA $= 1.00ms \pm 0.02ms$; 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 $t_v$ time line. 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 $\langle s_0, s_2\rangle$, $\langle s_2, s_3\rangle$, $\langle s_3, s_5\rangle$. 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 $\langle s_i, s_{i+1}, ..., s_n, lb, ub\rangle$ such that for $i \leq j < n$, $\langle s_j, s_{j+1}\rangle \in R$ and $lb, ub \in \Re$. $lb$ and $ub$ indicate the lower bound and the upper bound on the combined time delays of transitions $\langle s_i, s_{i+1}\rangle, ..., \langle s_{n-1}, s_n\rangle$, respectively.

A composite TTS (CTTS) is a 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.

A composite constraint is a constraint that enforces lower or upper bounds on more than one transition. The composite constraints corresponding to requirements and how they induce composite constraint is defined below.

### 3.6.10. First Composite Constraint

This composite constraint is introduced by the following requirements

**VS.1** *VS cannot be generated within the interval $t_v \in$ (0, VRP).*

This requirement signifies that on the time line of $t_v$ there cannot be any ventricle sensed event for as long as $t_v$ is between 0 and VRP.

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

This requirement signifies that if there is a sensed ventricle event for $t_v$ between 0 and VRP it should be disregarded. The purpose of this requirement is to ensure that requirement VS.1 is fulfilled.

**VS.3** *If $V_{in}$ occurs at $t_v \geq$ VRP, VS is to be created at $t_v$.*

This requirement ensures that if $V_{in}$ occurs at $t_v$ greather than VRP, VS should be issued at the same instance of $t_v$.

The composite constraint is enforced because of the fact that transition from $s_0$ to $s_1$ is dependent on the time line $t_v$. The state $s_1$ is an atrial sensed event. After this event pacemaker waits for the ventricle sense. The above requirement VS.1 and VS.2 states that $s_2$ cannot transition to $s_5$ until $t_v$ is between 0 and VRP. This set of bounds on the same time line for two consecutive transitions enforces a composite constraint.

Similarly transition from $s_0$ to $s_2$ and then to $s_3$ is dependent on time line $t_v$. $s_3$ is the state is the atrial pacing de-assert event. After this event pacemaker waits for a ventricle sense. The requirement states that at state $s_3$ no ventricle sensed event can be registered for as long as $t_v$ is between 0 and VRP. As these requirements enforces a bound on $t_v$ for more than two consecutive transitions so it signifies a composite constraint.

$$R_{C1} = \{\langle s_0, s_1, s_5, \text{VRP}, X\rangle, \langle s_0, s_2, s_3, s_5, \text{VRP}, X\rangle\}$$

In the above and in the discussions that follow, $X$ indicates don't care. An $X$ 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.

### 3.6.11. Second Composite Constraint

The second composite constraint is introduced by the following requirement:

**VP.2** *VP cannot be generated within $t_v \in$ (0, URI).*

This requirement ensures that no ventricle pacing can be issued unless $t_v$ is between 0 and URI.

The requirement VP.2 also results in composite constraints. VP.2 gives a lower bound on when VP can be generated, but on the $t_v$ time line. 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 $\langle s_0, s_2\rangle$, $\langle s_2, s_3\rangle$, $\langle s_3, s_4\rangle$.. Therefore, in order to satisfy

VP.2, we need a new transition relation $R_{C2}$ that covers the requirement stated in the property VP.2. Composite constraint $R_{C2}$ is given as below:

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

The composite constraints of the pacemaker specification $R_C$ is given by:

$$R_C = R_{C1} \bigcup R_{C2}$$

### 3.7. UPPAAL Basics

UPPAAL is a tool used for the verification of real time system. In order to verify a system in UPPAAL, it should be expressed as timed automata. UPPAAL serves as a model checker which is based on the theory of timed automata [10]. A timed automaton is a finite set of real valued clocks. All clocks tick at the same rate, synchronously. Real time systems are modeled in UPPAAL as a set of such automata that communicates via broadcast channels and common variables. A model in UPPAAL is also comprised of variables that are part of the state of the system. These variables act just like the variables in C or any other programming languages. They can be used for arithmetic operation and can be taken as output or input. The state of the system encoded in UPPAAL can be extracted from the values of variables, clocks and the location of the automata. Some of the important concept of an UPPAAL model is described below.

#### 3.7.1. Location

Location of timed automata is represented as a circle and called a location. Two locations are connected through edges. Edges are what we called state in our specification.

#### 3.7.2. Edges

Edges connect the locations. Edges are marked with guards, updates and synchronizations in case of broadcast channels.

#### 3.7.3. Guard

Guards are expressed in the form of algebraic expressions. A guard must be a conjunction of simple conditions on clocks, differences between clocks, and Boolean expressions on discrete variables. The limit on the guard is given by an integer expression. An example of a guard can be

$$(y >= 1)\&\&(y <= 6)$$

where y is a discrete variable or the clock. The transition with this guard cannot be taken unless y is between 1 and 6.

### 3.7.4. Updates

An update is a list of expressions that are used to update the value of a clock variable. The updated value on a variable should remain within the limit that the variable was declared with. This expression has no effect on the transition from one location to another. An example of an update expression is

$$x = 1, y = 2 * x$$

where x is set to 1 and y to twice of x, y and x can be clock variables or simple integers.

### 3.7.5. Invariant

An invariant must be a conjunction of simple conditions on clocks, differences between clocks, and Boolean expressions not involving clocks. Locations are labeled with invariants. The bound must be given by an integer expression. Invariant has an effect on the behavior of the system, but they are different from specifying safety properties in the requirements specification language. An important property of invariant is that if a state violates the invariant that state is marked as non reachable and thus is eliminated from the verification process. States which violate the invariants are undefined; by definition, such states do not exist. An example of an invariant can be

$$(x < y)\&\&(y < z)$$

## 3.8. Verification of CTTS Specification Using UPPAAL

We checked that the CTTS specification satisfies all the DDD mode pacemaker requirements from [57] (also given in Section 3.6) using UPPAAL [44][18], which is a standard tool for checking properties of timed systems [46]. UPPAAL can be used to check if a real-time system modeled as a network of timed automata satisfies properties expressed in CTL (Computaional Tree Logic) [21].

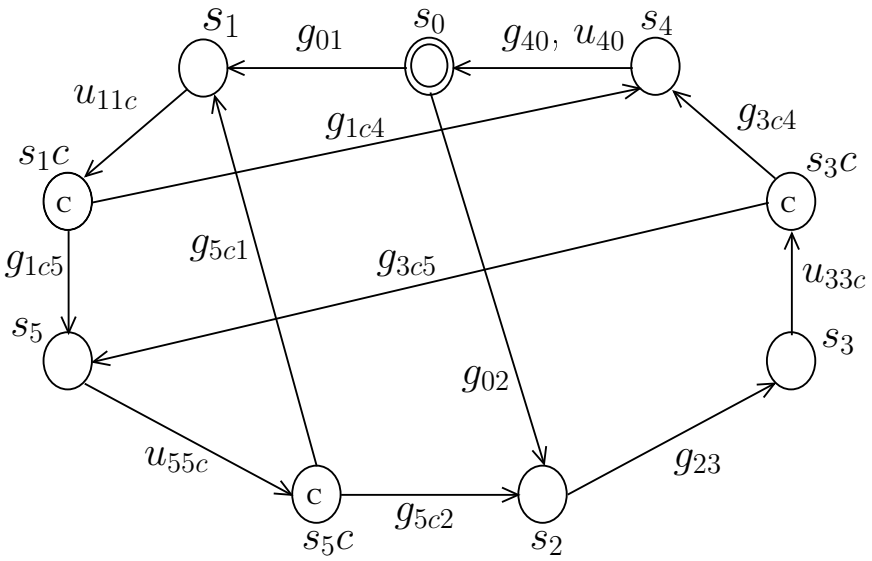


Figure 3.4. Formal specification model in UPPAAL

We encoded the CTTS specification as a timed automaton [11] 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 evaluates to true. An update on an edge is an expression that is executed when the guard evaluates to true. The UPPAAL model of the CTTS specification is shown in Figure 3.4. 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. Time lines $t_v$ and $t_a$, which are described in Section 3.6, are modeled as clocks *clktv* and *clkta* 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 $R_{C2}$.

$g_{01} \leftarrow ((AEI - 1) \geq clktv \geq ARP)$

This is the guard on the transition $s_0$ to $s_1$, this incorporates the requirement AS.1, AS.2, AS.3 and AP.1.

$g_{02} \leftarrow (clktv = AEI)$

This is the guard on the transition $s_0$ to $s_2$, this incorporates the requirement AP.2 and AP.3.

$g_{1c4} \leftarrow (clkta = AVI) \wedge (clktv \geq URI)$

This is the guard on the transition $s_1c$ to $s_4$, this incorporates the requirement VP.2 and VP.3.

$g_{1c5} \leftarrow (clkta \leq (AVI - 1)) \wedge (clktv \geq VRP)$

This is the guard on the transition $s_1c$ to $s_5$, this incorporates the requirement VP.1, VP.4, VS.1, VS.2 and VS.3.

$g_{23} \leftarrow (clktv = PWV)$

This is the guard on the transition $s_2$ to $s_3$, this signifies a delay of PWV between atrial pacing $AP_a$ assertion and deassertion $AP_d$.

$g_{3c4} \leftarrow (clkta = AVI) \wedge (clktv \geq URI)$

This is the guard on the transition $s_3c$ to $s_4$, this incorporates the requirement VP.2 and VP.3

$g_{3c5} \leftarrow (clkta \leq (AVI - 1)) \wedge (clktv \geq VRP)$

This is the guard on the transition $s_3c$ to $s_5$, this incorporates the requirement VP.1, VP.4, VS.1, VS.2 and VS.3.

$g_{40} \leftarrow (clkta = PWA)$

This is the guard on the transition $s_4$ to $s_0$, this signifies a delay of PWV between ventricle pacing VP and reset ($s_0$) which serves as de-we will transit to a committed state assertion of VP.

$g_{5c1} \leftarrow ((AEI - 1) \geq clktv \geq ARP)$

This is the guard on the transition $s_5c$ to $s_1$, this incorporates the requirement AS.1, AS.2, AS.3 and AP.1.

$g_{5c2} \leftarrow (clktv = AEI)$

This is the guard on the transition $s_5c$ to $s_2$, this incorporates the requirement AP.2 and AP.3.

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, time lines 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 $u_{11c} = (clkta = 0)$ on the edge from location $s_1$ to location $s_1c$. $s_3$ and $s_5$ are similarly modeled. The reason that we don't have a committed state for $s_0$ is that the guard $g_{40}$ is dependent on clock $clkta$ while the clock that is reset in this transition is $clktv$. The updates are given below:

$u_{11c} = (clkta = 0)$

This is an update on the transition from $s_1$ to $s_1c$. $s_1$ is an atrial event so we will transit to a committed state $s_1c$ to reset the time $t_a$.

$u_{33c} = (clkta = 0)$

This is an update on the transition from $s_3$ to $s_3c$. $s_3$ is an atrial event so we will transit to a committed state $s_3c$ to reset the time $t_a$.

$u_{40} = (clktv = 0)$

This is an update on the transition from $s_4$ to $s_0$. $s_4$ is an ventricle paced event so we will transit to a reset state $s_0$ to reset the time $t_v$.

$u_{55c} = (clktv = 0)$

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 for all paths emanating from this state. We also use the temporal operator [] (globally), which indicates for all states in the path. We have one property for each requirement. Below we explain three examples:

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

$$A[]\{(clktv < URI) \rightarrow \neg(s_4)\}$$

The above property specifies that no VP can be generated within $tv \in (0, URI)$. 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 $s_1c$ corresponds to AS, $s_2$ corresponds to $\text{AP}_a$ and so on.

AS.1 requires that no AS can be sensed within $tv \in (0, ARP)$, expressed as the following property.

$$A[]\{(0 < clktv < ARP) \rightarrow \neg(s_1 \vee s_{1c})\}$$

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

$$A[]\{(0 < clktv < VRP) \rightarrow \neg(s_5 \vee s_{5c})\}$$

### 3.8.1. Pacemaker Properties in UPPAAL

We have devised a bunch of properties in computational tree logic (CTL) that ensures the requirements of the pacemaker. The properties are given as below:

$$A[]\{(t_a < AVI) \rightarrow (\neg s_4 \wedge t_a < AEI)\}$$

$$A[]\{(t_a = AVI \wedge t_v >= URI \wedge (s_{3c} \vee s_{1c})) \rightarrow s_4\}$$

$$A[]\{(t_v >= 0 \wedge t_v < AEI \wedge s_2) \rightarrow (s_3 \wedge s_{3c} \wedge t_v = AEI)\}$$

$$A[]\{(t_v < URI \wedge t_v > 0) \rightarrow \neg s_4\}$$

$$A[]\{(t_v < ARP \wedge t_v > 0) \rightarrow \neg(s_5 \wedge s_{5c})\}$$

$$A[]\{(t_v < VRP \wedge t_v > 0) \rightarrow \neg(s_5 \wedge s_{5c})\}$$

$$A[]\{t_v >= VRP \rightarrow ((s_5 \wedge s_{5c}) \vee s_2)\}$$

$$A[]\{(t_v >= 0 \wedge t_v < AEI) \rightarrow \neg(s_3 \wedge s_{3c})\}$$

$$A[]\{(tv >= ARP) \rightarrow (s_1 \wedge s_{1c})\}$$

$$A[]\{(t_v > 0 \wedge t_v < ARP) \rightarrow \neg(s_1 \wedge s_{1c})\}$$

$$A[]\{(t_a > 0 \wedge t_a < AVI) \rightarrow \neg s_4\}$$

$$A[]\{(tv < AEI) \rightarrow (\neg(s_3 \wedge s_{3c} \wedge s_2) \wedge t_v < AEI)\}$$

The CTL properties are verified in UPPAAL verifier and all properties are verified against the UPPAAL model shown in Figure 3.4. These properties ensure that all the requirements of pacemaker are covered and fulfill the specification TTS in Figure 3.3.

### 3.9. 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 towards the validation of the control programs at the object code level. For the verification methodology, we employ the theory of Well-Founded Equivalence Simulation (WFS) refinement [51], 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 3.6). In the context of WFS refinement, both the implementation and specification are modeled as transition systems (TSs). In Section 3.6, 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 mod-

eled 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, $\text{AP}_a$, VS, VP, $\text{AP}_d$). 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 WFS refinement [51] 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 WFS refinement. Below are the definitions for WFSs and WFS refinement. In [51] [50][52][53] a more detailed description of WFS refinement is provided. ( *Well-Founded Simulation (WFS)* $B \subseteq S \times S$ is a *well-founded simulation* on TS $\mathcal{M} = \langle S, R, L \rangle$ iff:

(Wfs1) $\langle \forall s, w \in S : sBw \ : \ L(s) = L(w) \rangle$

(Wfs2) There exists functions, $rankt : S \times S \to W$,

$rankl : S \times S \times S \to \mathbb{N}$,

such that $\langle W, \prec \rangle$ is well-founded, and

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

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

(b)$(uBw \ \wedge \ rankt(u, w) \prec rankt(s, w)) \ \vee$

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

$\rangle$

**Definition 5:**(*Simulation Refinement*) Let $M$=$\langle S,R,L\rangle$, $M'$=$\langle S', R', L'\rangle$, and $r : S \to S'$. We say that $M$ is a simulation refinement of $M'$ with respect to refinement map $r$, written $M \sqsubseteq_r M'$, if there exists a relation, $B$, such that $\langle \forall s \in S :: sB(r.s)\rangle$ and $B$ is an STS on the TS $\langle S \uplus S', R \uplus R', \mathcal{L}\rangle$, where $\mathcal{L}.s = L'(s)$ for $s$ an $S'$ state and $\mathcal{L}.s = L'(r.s)$ otherwise.

In the above definitions, $\mathcal{M}$ is the implementation TS and and $\mathcal{M}'$ is the specification 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 $\langle W, \lessdot\rangle$, where $W$ is a set and $\lessdot$ is a binary relation on $W$ such that there are no infinitely decreasing sequences on $W$, with respect to $\lessdot$. We employ the well-founded structure consisting of the set of natural numbers and the 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 WFS 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 WFS refinement proof obligations.

There are several challenges to applying an SMT solver for this problem. The first challenge is that the WFS 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 WFS 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 1. The second challenge is that of reachability. The WFS 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 form the initial states), this would lead to spurious counter examples, 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:

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

The invariant uses the refinement map function $r$, which we define as a function that projects the values AS, $\text{AP}_a$, VS, VP, and $\text{AP}_d$ 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. $w.td_a$ and $w.td_v$ 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 $\mathcal{M}_{\mathcal{PM}}$, 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 3.3) is non-deterministic. For

the pacemaker specification, we need 16 proof obligations, where 10 proof obligations represent the non-stuttering 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 (implementations are also non-deterministic). $\mathrm{A_{in}}$ and $\mathrm{V_{in}}$ correspond to the inputs to the pacemaker from the atrium and ventricle, respectively. $\mathrm{A_{in}}$ and $\mathrm{V_{in}}$ 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:** $\big((r(w) = s_0) \wedge (\mathrm{ARP} \leq w.tdv \leq \mathrm{AEI} - 1) \wedge (\mathrm{A_{in}} = 0)\big) \rightarrow \big(r(v) = s_0\big)$

This property signifies stutter on state $s_0$ based on time line $tdv$ and $\mathrm{A_{in}}$.

**PF02:** $\big((r(w) = s_1) \wedge (w.tda \leq \mathrm{AVI} - 1) \wedge ((\mathrm{V_{in}} = 0) \wedge (w.tdv \geq \mathrm{VRP}\big) \rightarrow \big(r(v) = s_1\big)$

This property signifies stutter on state $s_1$ based on time line $tda$, $tdv$ and $\mathrm{V_{in}}$.

**PF03:** $\big((r(w) = s_2) \wedge \neg(w.tdv = \mathrm{PWV}) \rightarrow \big((r(v) = s_2)\big)$

This property signifies stutter on state $s_2$ based on time line $tdv$.

**PF04:** $\big((r(w) = s_3) \wedge (w.tda \leq \mathrm{AVI} - 1) \wedge (\mathrm{V_{in}} = 0) \wedge (w.tdv \geq \mathrm{VRP}\big) \rightarrow \big(r(v) = s_3\big)$

This property signifies stutter on state $s_3$ based on time line $tda$, $tdv$ and $\mathrm{V_{in}}$.

**PF05:** $\big((r(w) = s_4) \wedge \neg(w.tda = \mathrm{PWA})\big) \rightarrow \big(r(v) = s_4\big)$

This property signifies stutter on state $s_4$ based on time line $tda$.

**PF06:** $\big((r(w) = s_5) \wedge (\mathrm{ARP} \leq w.tdv \leq \mathrm{AEI} - 1) \wedge (\mathrm{A_{in}} = 0)\big) \rightarrow \big(r(v) = s_5\big)$

This property signifies stutter on state $s_5$ based on time line $tdv$ and $\mathrm{A_{in}}$.

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. $Rank_a$ and $Rank_v$ are the ranks for the states where the active counter is w.$tda$ and w.$tdv$, respectively. 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.

$$Rank_a : rank(w) = max - w.tda$$

$$Rank_v : rank(w) = max - w.tdv$$

PF7-PF16 give the proof obligations corresponding to the non-stuttering cases.

**PF07:** $\big((r(w) = s_0) \wedge (\text{ARP} \leq w.tdv \leq \text{AEI} - 1) \wedge (\text{A}_{\text{in}} = 1)\big) \rightarrow \big(r(v) = s_1\big)$

The proof calls for the transition from state $s_0$ to $s_1$ based on time line $tdv$ and $\text{A}_{\text{in}}$.

**PF08:** $\big((r(w) = s_0) \wedge (w.tdv = \text{AEI})\big) \rightarrow \big(r(v) = s_2\big)$

The proof calls for the transition from state $s_0$ to $s_2$ based on time line $tdv$.

**PF09:** $\big((r(w) = s_1) \wedge (w.tda \leq \text{AVI} - 1) \wedge (\text{V}_{\text{in}} = 1) \wedge (w.tdv \geq \text{VRP})\big) \rightarrow \big(r(v) = s_5\big)$

The proof calls for the transition from state $s_1$ to $s_5$ based on time line $tdv$, $tda$ and $\text{V}_{\text{in}}$.

**PF10:** $\big((r(w) = s_1) \wedge (w.tda = \text{AVI}) \wedge (tdv \geq \text{URI})\big) \rightarrow \big(r(v) = s_4\big)$

The proof calls for the transition from state $s_1$ to $s_4$ based on time line $tdv$ and $tda$.

**PF11:** $\big((r(w) = s_2) \wedge (w.tdv = \text{PWV})\big) \rightarrow \big(r(v) = s_3\big)$

The proof calls for the transition from state $s_2$ to $s_3$ based on time line $tdv$.

**PF12:** $\big((r(w) = s_3) \wedge (w.tda \leq \text{AVI} - 1) \wedge (\text{V}_{\text{in}} = 1) \wedge (w.tdv \geq \text{VRP}\big) \rightarrow \big(r(v) = s_5)\big)$

The proof calls for the transition from state $s_3$ to $s_5$ based on time line $tdv$, $tda$ and $\text{V}_{\text{in}}$.

**PF13:** $\big((r(w) = s_3) \wedge (w.tda = \text{AVI}) \wedge (tdv \geq \text{URI})\big) \rightarrow \big(r(v) = s_4\big)$

The proof calls for the transition from state $s_3$ to $s_4$ based on time line $tda$ and $tdv$.

**PF14:** $\big((r(w) = s_4) \wedge (w.tda = \text{PWA})\big) \rightarrow \big(r(v) = s_0\big)$

The proof calls for the transition from state $s_4$ to $s_0$ based on time line $tda$.

**PF15:** $\big((r(w) = s_5) \wedge (\text{ARP} \leq w.tdv \leq \text{AEI} - 1) \wedge (\text{A}_{\text{in}} = 1)\big) \rightarrow \big(r(v) = s_1\big)$

The proof calls for the transition from state $s_5$ to $s_1$ based on time line $tdv$ and $\text{A}_{\text{in}}$.

**PF16:** $\big((r(w) = s_5) \wedge (w.tdv = \text{AEI})\big) \rightarrow \big(r(v) = s_2\big)$

The proof calls for the transition from state $s_5$ to $s_2$ based on time line $tdv$.

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.
Let $\mathcal{M}' = \mathcal{M}_{PM}$. Let $\mathcal{M}$ be an implementation of $\mathcal{M}'$. $\mathcal{M}$ is WFS refinement of $\mathcal{M}'$ if every transition of $\mathcal{M}$ satisfies one of PF01-PF16.

For an implementation (object code) to be a WFS refinement of a specification (CTTS model), as per the definition of WFS refinement, every transition of the implementation has to match a transition of the specification, 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, i.e., all the transitions of the implementation and specification are accounted for. PF7-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 invariant 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 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 PF7-PF16 satisfy case (a) of Definition 4 (non-stuttering transitions). Each of PF01-PF06 satisfy 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 WFS refinement of the pacemaker specification $\mathcal{M}_{PM}$.

### 3.10. 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 WFS refinement proof

obligations. We used the z3 [3] SMT solver for verification. The input language to the z3 solver is the SMT-LIB language [2]. The verification process involves four high-level steps. The first step is to model the object code control program as 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 instruction and hence each instruction function 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 pre-conditions and post-conditions 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 PF1-PF16. This was checked using the z3 solver [3]. 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 non-stuttering proof obligations (PF7-PF16) were satisfied by at least one instruction function. If there is a non-stuttering 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 counter examples 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 PF1-PF16 for each instruction function. Each iteration of the loop would call the z3 solver to check if the instruction function satisfies one of PF1-PF16. The fourth step can be automated along with the third step by using flag variables that track if each of PF7-PF16 were 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 were 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 below:

**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 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 indicate that an AS or VS has occurred, respectively. The bug manifests when VS is followed by an AS. In this case, the IO2IntStatR register value changes from 1 to 3, which is incorrect and indicating that both an AS and a VS has 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.

### 3.11. Conclusion

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 micro controller embedded in the device. The verification methodology is based on the set of safety requirements given in the Boston Scientific (a clinical literature on pacemakers [61]). Boston Scientific is a leading manufacturer and seller of pacemakers and several other medical solutions [6]. The values used for the critical timing cycles of DDD mode pacemakers are obtained from the actual clinical settings [4]. The Boston Scientific requirements are formalized and presented in [57] based on the authentic clinical literature [61]. In [32] 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 research we have focused on functional verification of object code control programs. For future work, we plan to extend our methods to address the verification of timing requirements using the theory of Timed refinements [26] .

# 4. REFINEMENT CHECKING FOR DC MOTOR-BASED INSULIN PUMP

## 4.1. Introduction

Diabetes is a medical condition in which the amount of glucose in the blood is extremely high. Glucose is the energy that we get from the food that we eat. The glucose that we consume enters the cells and gives them energy with the help of insulin. There are two types of diabetes, type I and type II. In type I diabetes patients no insulin is made. Whereas, in type II diabetes, body cannot use insulin; and in some cases cannot make insulin well enough. Insulin is inevitable, in the absence of which glucose stays in the blood. Without enough insulin, the glucose stays in your blood. If a person has too much blood glucose for a longer period of time, it can lead to serious health problems. Diabetes can also cause heart disease, coma and even death in severe cases if left untreated.

Insulin pump is a medical device used to treat diabetes by injecting insulin doses into a patient's body. An insulin pump consists of an insulin reservoir that is connected to a piston on one end and an infusion set (consisting of a tube with a fine needle or cannula) on the other end. The piston's motion is controlled by a motor. A microcontroller is used to control the rotation of the motor, which in turn impacts the motion of the piston. The insulin pump monitors the blood glucose level of the patient. If the blood glucose level is higher than a threshold, the microcontroller activates the piston to deliver the required insulin dosage.

Insulin pump is a safety critical device because a dosage of insulin higher than required can lead to hypoglycemia (low blood sugar), which can in turn lead to coma or even death. Hypoglycemia should never occur and as its occurrence can be instantaneously very harmful to the patient. If the insulin pump delivers lower dosages of insulin than required , then this results in hyperglycemia (high blood sugar), which is harmful to the patients health in the long term. A typical insulin pump delivers different functionality and as such the control software is quite complex and easily prone to defects. Since the control software is responsible for ensuring the correct insulin dosage, defects in the software can cause hyperglycemia or hypoglycemia. For example in March

2014, Medtronic [7] a leading manufacturer of insulin pumps, issued a recall to notify its users of a button pressing error that can result in unintended high insulin delivery. The Food and Drug Administration also issued a considerable number of insulin pump recalls during the last decade [1].

Each year over 18000 youth are diagnosed with Type I diabetes and over 5000 with Type II diabetes [27].It has been estimated that 1 out of 3 children born in 2000 will develop diabetes in their lifetime [55]. In order to prevent serious health consequences of Type I diabetes, people and children with Type I diabetes must have insulin delivered by a pump or injection [27]. The use of insulin pump in children puts the delivery of unwanted insulin dosages at high risk, as they may play with their pump and inject unwanted insulin accidently. This can lead to DKA (diabetic ketoacidosis), DKA is a metabolic state resulting from acute hyperglycemia and if left untreated can cause death [42]. DKA is also the most frequent cause of death caused by diabetes [24]. Where as if pump is not used correctly in Type I patients especially children, it can lead to complications like hypoglycemia [24]. Extreme hypoglycemia can cause severe consequences even death in young children due to their high glucose consumption[28].

### 4.2. Related Work

Zhang et al., [69] and Jetley & Jones [33] have identified safety requirements for insulin pumps and infusion pumps in general, respectively. The status quo on formal methods for infusion/insulin pumps is model-driven software development [41], where formal verification tools like UPPAAL [18] and Kronos [64] are used to check that higher-level models satisfy safety requirements. However, there is quite a large gap between the high-level models and the actual code that is executed in the device. This gap is bridged using a model-driven approach [41], where platform-independent source code is generated from high-level models using synthesis tools. The platform-independent source code is then manually augmented with glue code, which is used to configure hardware interrupts and interface the software controller with low-level peripheral units and devices such as timers, communication channels, ports, etc. The glue-code-augmented source code is then compiled and assembled to generate object code. There are numerous sources of error that can compromise the safety of the object code in this process. The synthesis tools, glue code, compilers, and assemblers can be buggy. The process of integrating the platform-independent source code with the glue code is manual and can be very error prone as it involves programming

and configuring many low-level parameters associated with peripheral units located inside and outside the microcontroller. Our work is targeted at guaranteeing that the software controller at the object-code-level is safe by developing formal verification techniques that can verify the object code against higher-level system models.

Generic safety requirements for a highly abstract generic insulin infusion pump (GIIP) are presented by Zhang et. al [69]. These safety requirements address the GIIP hazards identified by consulting with manufacturers, pump users, and clinicians; and by reports collected by the FDA [1]. These requirements cover all the possible options that are provided in the insulin pumps for example two basal profiles, temporary basal injection, extended bolus delivery, drug reservoir and alarms, to name a few. The aim of this research article is to verify and ensure correct and accurate delivery of insulin units in both basal and bolus mode. Therefore, we have gathered only those requirements from [69] which corresponds to the administration and delivery of basal and bolus insulin. Requirements 1.1.2, 1.2.1, 1.3.1, 1.3.2 and 1.3.5 are the requirements that are covered by our specifications. Insulin pump vendors use stepper or DC motor to inject insulin in patient's body [8]. In this work, specifications for DC motor with two modes, Basal mode and Bolus mode is presented. Each specification is represented as a transition system. Similarly the verification methodology for both cases is devised.

### 4.3. Use of DC Motor for Insulin Pumps

A DC motor is a simple form of motor that rotates with the principle of torque. The torque is produced by electricity and a magnetic field that causes the motor to rotate. A DC motor can be easily controlled by a pulse width modulated signal. The speed of the motor is an essential feature in applications where some tasks have to be done at a certain speed within certain time limits. However for insulin pumps, the most important requirement is to inject insulin dosages at correct time and correct dosages. The amount of insulin is the number of insulin units to be administered. Whereas, the rate of insulin delivery is the speed with which insulin is being injected. It is to be noted that the amount of insulin is the most crucial thing to be considered in insulin pumps. On the other hand, the rate at which insulin is injected is also important but if the rate is a little lower or higher than the required rate it will not affect much. There are several insulin pump vendors that use dc motors in their insulin pump because of the ease of use. These motors come with ratings of rotations per minute (rpm), operating voltage, maximum allowable frequency.

DC motors are driven by a pulse width modulated signal; at hundred percent duty cycle of the operating voltage motor will have the maximum rotations per minute. Similarly when the pulse width modulated signals duty cycle is reduced and the speed of motor is decreased, the resulting rotations per minute will also increase. We have employed dc motors with a constant pulse width modulated signal, with constant speed and hence constant rotations per minute. The required number of rotations to inject a given number of insulin is made by applying pulse width signal only for the amount of time in which the rotations will be complete. For example if the insulin dosage requires 100 rotations the motor will execute 100 rotations and then stopped. This can be done by calculating the number of pulses required to rotate the motor for a required number of rotations. One important thing to note here is that insulin is injected at a very slow pace, because if insulin is injected rapidly it cannot be absorbed by the body completely. A typical insulin injection can take anywhere from 5 to 10 minutes or even more depending on the amount of units. Thus the speed of motor needs to be extremely slow, which helps to convert the number of rotations to number of pulses required to inject the required dosage of insulin. The formula that we have devised is given below:

$$\text{Number of pulses} = \text{Number of rotations} \times \frac{60f}{speed(rpm)}$$

This relation converts the number of rotations of motor to number of pulses. The number of rotations of motor required is patient specific, for that the basics of insulin pump should be known.

### 4.3.1. Insulin Pump Basics

Insulin pumps typically consist of three main components that include a reservoir, a control unit, and a disposable infusion set. The reservoir is like a syringe or a cartridge that stores around 300 units of insulin (3 ml), and can be refilled when empty. The control unit includes a microcontroller that controls the position of the piston in the insulin reservoir using an actuator such as a DC motor or stepper motor. The infusion set has a tube with a very fine needle (cannula). The cannula is inserted under the skin. The tube connects the insulin reservoir to the needle under the skin, thereby enabling the injection of insulin to blood vessels. Insulin is injected in the form of units where 100 units of insulin is equivalent to 1 ml. Insulin pumps provide two types of insulin doses, basal and bolus. A basal insulin dose is required by the body to keep the blood glucose

under control during the day and night depending on the type and severeness of diabetes hence it is patient specific. For a bolus dose, the user enters his current blood glucose in the insulin pump which calculates the bolus insulin dose of the patient. Below we describe basal and bolus dose.

### 4.3.2. Basal Dose

A basal dose is the insulin dose injected throughout the day at regular intervals between meals and during the night. This is the dose that is recommended by a physician based on the patient's disease. Basal dose is programmed in the pump at the time of pump setting (the time of first use of the pump). This dose will be injected at regular intervals without prompting the insulin pump to inject it. If a patient's basal dose is altered by the physician, user can re-program the pump easily for the new dose.

### 4.3.3. Bolus Dose

A bolus dose is a user initiated insulin dose used to combat higher blood glucose levels that typically occur after a meal. This type of dose is decided and initiated by the user himself. Before taking a meal, patient will calculate the amount of insulin units required to compensate for the glucose he will take with the meal. This is done by using formulas and with the amount of carbohydrates patient is going to consume in the meal.

### 4.3.4. Calculation of Dosage

There is a threshold for the blood glucose for every patient; the blood glucose of the patient should remain within this threshold. If the patient's blood glucose exceeds this threshold for a longer period of time, then that can lead to serious health consequences. For a basal dose the number of insulin units are fixed and already known. For the bolus mode user himself calculates the insulin units. Once the insulin units are known, they are converted into insulin units based on the correction factor and rotation factor.

### 4.3.5. Rotation Factor ($\mathcal{F}_{rot}$)

The rotation factor ($\mathcal{F}_{rot}$) indicates how much rotations are required to inject one unit of insulin.

### 4.3.6. Correction Factor ($\mathcal{F}_{corr}$)

The correction factor is the amount of blood glucose decreased with one unit of insulin. Based on the design and the rotation factor of the insulin pump, insulin units are converted in the number of rotations of the motor. The number of rotations based on the insulin units that are

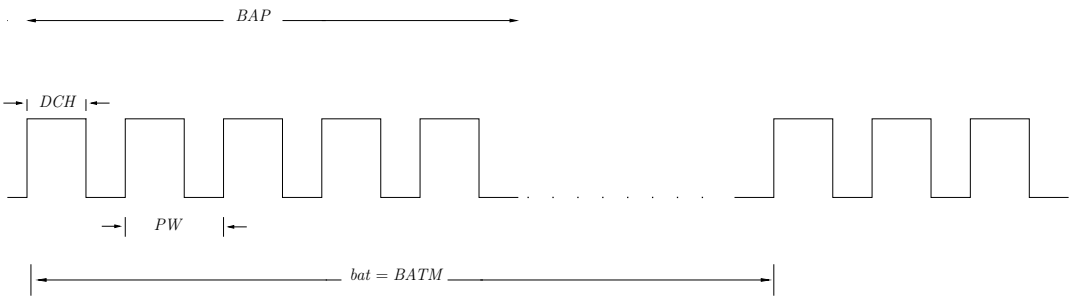


Figure 4.1. Pulse width modulated signal representing $pv_{basal}$

required to be injected is as follows.

$$riu = no\ of\ insulin\ units \times \mathcal{F}_{corr}$$

The reservoir of the insulin pump has a piston attached to it which is pressed forward by the rotation of a stepper motor. For example for a bolus dose if the difference of patient's current and target blood glucose is 100 and the correction factor is 50, the patient requires 2 units of insulin. The number of rotations require to inject 2 units of insulin is 200, if the rotation factor is 100. The role of the insulin pump is to inject the calculated amount of insulin with the calculated rotations of the motor. If the rotations exceed or lag behind the number of rotations required for a continuous period of time, it may cause adverse health consequences.

### 4.4. Specification of DC Motor Control for Basal Mode

In this section we present formal specification model for a DC motor control for Basal mode. DC motors require a pulse width modulated signal for their operation. The speed of the motor can be controlled by changing the duty cycle of the pulse. In a DC motor based insulin pump, the required number of insulin units are translated in number of rotations. Basal mode is always injecting same pre-defined amount of insulin units so the relative number of rotations is constant. We have translated number of rotations of the DC motor in number of basal pulses (BAP) that the pulse width modulated signal should generate. Figure 4.1 shows the pulse modulated signal to drive the DC motor. When basal insulin is injected the insulin pump should be in Basal mode, and the Basal mode flag $Basal_{mode}$ should be high. Below we explain the state components and constants.

#### 4.4.1. Duty Cycle-High (DCH)

DCH is the time the pulse remains high, so this parameter essentially decides the duty cycle of the pulse. DCH should be 100 in order to have a hundred percent duty cycle and 50 for a 50 percent duty cycle. The value of DCH is a constant that can be changed but for our specification we our employing the concept of number of pulses instead of the duty cycle.

#### 4.4.2. Pulse Width (PW)

PW is the width of one pulse. The width of pulse is the time between the high duty cycle and low duty cycle. The width of the pulse decides the number of waves generated in a fixed period of time. It can be changed but this specification calls for fixed pulse width PW.

#### 4.4.3. Basal Pulses (BAP)

BAP is the total number of pulses. The number of pulses is derived based on the number of rotations. The number of rotations $riu_{basal}$ help to estimate the number of pulses required to inject the required unit of insulin. The number of pulses (BAP) is derived with the following relation, where $f$ is the allowable frequency for the DC motor.

$$\text{BAP} = riu_{basal} \times \frac{60f}{speed(rpm)}$$

#### 4.4.4. State Components ($\mathcal{S}_{basal}$)

We have expressed the specifications for DC motor based insulin pump in the form of a Transition system (TS).

**Definition 1:** A Transition System (TS) $\mathcal{M}$ 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.

The specification TS is comprised of a large number of states which cannot be modeled as a transition system. So instead of modeling that we have described the following state components.

$$\mathcal{S}_{basal} = \{bat, Basal_{mode}, pv_{basal}, pc_{basal}, pwt\}$$

#### 4.4.5. Basal Flag ($Basal_{mode}$)

$Basal_{mode}$ is a Boolean value, which if true tells that the insulin pump is in Basal mode. This is implemented as a flag that should remain asserted for as long as basal insulin is being injected. When a request for a bolus injection is being made, basal flag $Basal_{mode}$ will be de-asserted and $Bolus_{mode}$ will be asserted.

#### 4.4.6. Basal Counter (*bat*)

*bat* is the basal mode counter; it counts to the maximum value BATM. BATM signifies the interval between two consecutive basal insulin injections. For eexample if a person needs a basal injection every 2 hours the value of BATM will be set to 2 hours. *bat* will count to BATM gets reset and gives a prompt to inject basal insulin and then counts back to BATM to inject insulin every 2 hour. If a user needs to change the frequency of basal injection the value of BATM should be changed.

$$bat' \leftarrow if\ (bat = \text{BATM}) \qquad bat' = 0$$
$$else \quad bat' = (bat + 1)$$

#### 4.4.7. Pulse Width Timer ($pwt$)

$pwt$ is the timer that implements one pulse. For example if the length of 1 pulse is 1 $\mu$sec, $pwt$ will count to 1 $\mu$sec. We have already defined the length of one pulse as PW.

$$pwt' \leftarrow if\ (pwt = \text{PW}) \qquad pwt = 0$$
$$else \quad pwt = (pwt + 1)$$

#### 4.4.8. Pulse Width Modulated Value ($pv_{basal}$)

The specification is implemented on an embedded platform. The external port pin of the platform is utilized to generate the pulse width modulated value and thus rotate the motor. The microcontroller pin that generates the pulse has the Pulse Width Modulated Value called $pv_{basal}$. The duty cycle of the pulse is controlled by giving '1' to $pv_{basal}$ for a a period equals DCH. Therefore, in order to have a 50% duty cycle DCH should be 50% of PW. We use the timer $pwt$ that resets when the time equals PW. $pv_{basal}$ changes from '1' to '0' and '0' to '1' thus generating the pulse.

As PW is the width of one pulse, so every time $pwt$ resets, $pv_{basal}$ gets a value '1' as shown in Figure 4.1. $pv_{basal}$ will remain high based on the duty cycle time specified by DCH. When $pwt$ reaches the time DCH, $pv_{basal}$ is de-asserted. This sequence of 1's and 0's continues until and unless the required number of pulses BAP is generated. The definition of $pv_{basal}$ is given below

$$
\begin{aligned}
pv_{basal}{}' \leftarrow & if\ (bat = \text{BATM}) && pv_{basal} = Basal_{mode} \\
& if\ (pc_{basal} = 0) && pv_{basal} = 0 \\
& if\ \big((pwt = \text{PW}) \wedge \neg(\text{PW} \times \text{BAP} \leq bat \leq \text{BATM})\big) && pv_{basal} = Basal_{mode} \\
& if\ (pwt = \text{DCH}) && pv_{basal} = 0 \\
& else\ pv_{basal}
\end{aligned}
$$

$pv_{basal}$ will get asserted for the first time when $bat$ equals BATM and there is a time for basal injection. It will remain asserted for as long as pulse width timer $pwt$ has reached DCH that's when $pv_{basal}$ is de-asserted. It will remain de-asserted until pulse width timer $pwt$ reaches pulse width PW and then asserted again to start the new pulse. This will continue until all the required number of pulses is generated and the pulse counter $pc_{basal}$ has become zero

### 4.4.9. Pulse Counter ($pc_{basal}$)

We use a counter $pc_{basal}$ that keeps a track of the number of pulse. When the basal timer $bat$ hits BATM, $pc_{basal}$ gets the value of BAP which is the number of pulses to be generated. Once $pc_{basal}$ has BAP, the motor will start injecting insulin based on the pulse width modulated signal $pv_{basal}$. $pc_{basal}$ is decremented based on the value of $pwt$. Whenever pulse width timer $pwt$ hit the value of length of the pulse PW, that is when one pulse is generated and one rotation of motor is complete. The definition of $pc_{basal}$ is given below

$$
\begin{aligned}
pc_{basal}{}' \leftarrow & if\ (bat = \text{BATM}) && \text{BAP} \\
& elseif\ [(pc_{basal} \neq 0) \wedge (pwt = \text{PW})] && (pc_{basal} - 1) \\
& else\ pc_{basal}
\end{aligned}
$$

### 4.4.10. Basal Invariants (*inv*)

Insulin pump is a real time system where timer keeps counting throughout while the system is on, which makes the state space quite large. As we have a large number of states, so we need invariants to describe the states. Invariants are the range of values of the state components $\mathcal{S}$ which cannot be exceeded in any state of the system. Invariants are used to restrict our verification effort to the reachable states of the implementation. Unreachable states can often be inconsistent and trigger spurious bugs. These invariants should be satisfied by the implementation model. The basal invariants are given below:

$$\textbf{\textit{inv1:}} \qquad 0 \leq pwt \leq \text{PW}$$

*inv1* signifies that for all states of the system the pulse width timer *pwt* will remain between 0 and PW. This should be true for all the reachable states of the system. It also makes sure that the length of one pulse does not exceed or lag behind the pre-defined length.

$$\textbf{\textit{inv2:}} \qquad 0 \leq bat \leq \text{BATM}$$

*inv2* signifies that for all states of the system, basal timer *bat* will remain between 0 and BATM. This should be satisfied by all the reachable states of the system. This invariant ensures that the rate and frequency of the basal injection is not affected. If for some reason this invariant is not satisfied, it can result in a delay or over dose on the basal insulin.

$$\textbf{\textit{inv3:}} \qquad 0 \leq pc_{basal} \leq \text{BAP}$$

*inv3* signifies that for all states of the system, the pulse counter $pc_{basal}$ will remain between 0 and BAP. This invariant enforces that for no reachable state of the system, should the $pc_{basal}$ lags or lead behind the specified range. If this invariant is not satisfied, it can lead to the injection of more or less insulin than required. The reason behind the significance of this invariant is the fact that any problem with the limits of the invariant can cause the motor rotate more or less thus causing states of the system which are not valid according to the formal specification of basal mode insulin injection.

## 4.5. Specification of DC Motor Control for Bolus Mode

This specification model is for the bolus mode. Bolus mode insulin is injected by the patient when he takes a meal or exercise. Bolus dose insulin injection is initiated by the insulin pump user. Like basal mode, bolus mode has a flag called $Bolus_{mode}$ associated with it. Bolus insulin cannot be injected while $Bolus_{mode}$ is de-asserted. Whenever bolus insulin is requested and a basal injection is underway, bolus request will be registered and put on hold. Once basal injection is complete, basal flag $Basal_{mode}$ will be de-asserted, bolus flag $Bolus_{mode}$ will be asserted and control will be given to the bolus mode. $Bolus_{mode}$ will remain high for as long as bolus injection is in process and will be de-asserted when it is completed. Control is then given back to the basal mode. The specification for bolus mode is similar to the basal mode except the calculation for dosage. Below we explain the state components and constants.

### 4.5.1. Bolus Pulses (BOP)

The basal dosage of insulin is programmed by the user which is converted into number of rotations $riu_{bolus}$ by using equation 4.3.6. The number of bolus mode pulses BOPis given by the following relation.

$$\text{BOP} = riu_{bolus} \times \frac{60f}{speed(rpm)}$$

### 4.5.2. State Components ($\mathcal{S}_{bolus}$)

Bolus mode functions exactly like basal mode except for the fixed insulin units at regular intervals of time. The specification TS is comprised of a large number of states, se have described the following state components.

$$\mathcal{S}_{bolus} = \{bot, Bolus_{mode}, pv_{bolus}, pc_{bolus}, pwt\}$$

Bolus mode insulin is also injected with the help of a pulse width modulated pulse as shown in Figure 4.1. Note that the parameters and counters like DCH, PW, $pwt$ will be the same for bolus mode. $Bolus_{mode}$ is a Boolean value, which if true tells that the insulin pump is in Bolus mode.

### 4.5.3. Pulse Width Modulated Value ($pv_{bolus}$)

The microcontroller pin that generates the pulse has the Pulse Width Modulated Value called $pv_{bolus}$. The duty cycle of the pulse is controlled by giving '1' to $pv_{bolus}$ for a a period equals DCH. The definition of $pv_{bolus}$ is given below

$$
\begin{aligned}
pv_{bolus}{}' \leftarrow & if\ (pwt = \text{DCH}) && pv_{bolus} = 0 \\
& if\ (pwt = \text{PW}) && pv_{bolus} = Bolus_{mode} \\
& if\ (pc_{bolus} = 0) && pv_{bolus} = 0 \\
& else\ pv_{basal}
\end{aligned}
$$

$pv_{bolus}$ will get asserted based on the value of the bolus flag. When the bolus flag $Bolus_{mode}$ gets asserted on a bolus request, $pv_{basal}$ will get asserted the instant $pwt$ hits PW to mark the start of a new pulse. It will continue to be asserted and de-asserted based on the value of DCH and $pwt$. The pulse width modulated signal will be stopped when $pc_{bolus}$ has become zero. Bolus mode flag $Bolus_{mode}$ will also be asserted at this instant and control is given back to the basal mode.

### 4.5.4. Pulse Counter ($pc_{bolus}$)

We use a counter $pc_{bolus}$ that counts the number of pulses generated. This is different from the pulse counter for basal mode ($pc_{basal}$) because it does not get the value of bolus pulses based on a pre-set timer like the basal timer in basal mode. It rather gets the value of bolus pulses BOP when the flag for bolus mode $Bolus_{mode}$ is asserted and the current value of pulses $pc_{bolus}$ is zero. Below is the definition of $pc_{bolus}$:

$$
\begin{aligned}
pc_{bolus}{}' \leftarrow & if(Bolus_{mode} \wedge pc_{bolus} = 0) && \text{BOP} \\
& elseif[(pc_{bolus} \neq 0) \wedge (pwt = \text{PW})] && (pc_{bolus} - 1) \\
& else\ pc_{bolus}
\end{aligned}
$$

The value of $pc_{bolus}$ is decremented based on the current value of $pc_{bolus}$ and the value of $pwt$. When the pulse width timer $pwt$ has completed one pulse by hitting PW and the current value of $pc_{bolus}$ is greater than zero, $pc_{bolus}$ will be decremented by one. $pc_{bolus}$ will remain zero once it has become

zero and the $Bolus_{mode}$ will be de-asserted to mark the end of bolus injection. Pulse counter will only be active during insulin delivery to keep track of the number of pulses of the motor.

### 4.5.5. Bolus Invariants (*inv*)

Invariants for the bolus mode are given below:

$$inv1: \qquad 0 \leq pwt \leq \text{PW}$$

*inv1* signifies that for all states of the system the pulse width timer *pwt* will remain between 0 and PW. This should be true for all the reachable states of the system. It also makes sure that the length of one pulse does not exceed or lag behind the pre-defined length.

$$inv2: \qquad 0 \leq pc_{bolus} \leq \text{BOP}$$

*inv2* signifies that the for all states of the system, the pulse counter $pc_{bolus}$ will remain between 0 and BOP. This invariant enforces that for no reachable state of the system, should the $pc_{bolus}$ lags or lead behind the specified range. If this invariant is not satisfied, it can lead to the injection of more or less insulin than required. The reason behind the significance of this invariant is the fact that any problem with the limits of the invariant can cause the motor rotate more or less thus causing states of the system which are not valid according to the formal specification of bolus mode insulin injection.

## 4.6. Formal Verification Methodology for Object Code Control Programs

In this section, we develop a methodology for formal verification of control programs for DDD mode insulin pumps. The control program is validated with a methodology that targets at the object code level. The theory of Well-Founded Equivalence Bisimulation (WEB) refinement [51] is employed for the verification, 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 4.4 and 4.5). In the context of WEB refinement, both the implementation and specification are modeled as transition systems (TSs). In Section 4.4 and 4.5, we have developed a TS specification for insulin pump control. The object code program is also modeled as TS. The control program is comprised of a number of assembly instructions. Each

assembly instruction captures the transition of the program and alters the state of the program. The instructions are thus modeled as functions that keep a track of the program state. Considering the first instruction of the program, all the parameters, registers, variables and the state of the processor is at reset state. The first instruction of the program will take the reset values as the current state of the system and calculate the next state values of the program. Therefore, each instruction will take its current state from the next state values of its predecessor instruction function, and then calculate the next state depending on the kind of the assembly instruction.

An important thing to consider here is the difference between specification TS and implementation TS (object code) when examined at a high level. The specification TS is a transition system expressed as set of relations, with no set of clearly defined states. The system progresses and changes state based on these equation. Where as in the implementation, state of the system at any given instance is the state of the object code program comprising of all the general purpose and special registers, and all the peripherals of the embedded system that are utilized for insulin pump specifications. The implementation progresses with the help of instruction functions, therefore each instruction function should match to the set of equations in the specifications. The other difference is the complexity of the implementation as compared to the specification. The specification is explicit enough in the form of equations where a single counter is expressed as a simple equation; on the other hand in the implementation this counter is implemented with a peripheral timer with its own set of registers and other complex requirements. The theory of WEB refinement [51] 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. The specification of insulin pump is expressed as TS encoded in the form of equations, but there are as many states in the implementation as the number of the program states the control program can take. There cannot be a formal refinement map that can be used to extract the similarities between the specification and the implementation TS. The concept of mapping each implementation state to a specification state without a refinement map requires manual efforts to come up with proofs that unfold the similarities between the two. These proof obligations enable the comparison of implementation states and specification states, even if these states look very different. 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 180 billion transitions for the basal mode, whereas the

specification TS is only set of states. 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 WEB refinement. In [51] [50][52][53] a more detailed description of WEB refinement is provided.

**Definition 5:**[50] Let $\mathcal{M} = \langle S, R, L\rangle$, $\mathcal{M}' = \langle S', R', L'\rangle$, and $r : S \to S'$. We say that $\mathcal{M}$ is a WEB refinement of $\mathcal{M}'$ with respect to refinement map $r$, written $\mathcal{M} \approx_r \mathcal{M}'$, if there exists a relation, $B$, such that $\langle \forall s \in S :: sBr(s)\rangle$ and $B$ is a WEB on the TS $\langle S \uplus S', R \uplus R', \mathcal{L}\rangle$, where $\mathcal{L}(s) = L'(s)$ for $s$ an $S'$ state and $\mathcal{L}(s) = L'(r(s))$ otherwise.

In the above definitions, $\mathcal{M}$ is the implementation TS and and $\mathcal{M}'$ 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 w.r.t. the low-level object code controller TS.

WEB refinement has the ability to reason about single transitions of the implementation and specification to devise a correctness proof. This can be done easily on the specification side, as the specification is only set of equations. Whereas, the object code control program TS have millions of transitions. Therefore in order to check the WEB refinement proof obligation, a decision procedure (SMT solver) is employed .

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 TS 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.

One of the challenges to apply SMT solver for insulin pump is of reachability. If all the states are considered while checking the WEB refinement proofs including those that are not reachable from the initial state of the system, it will lead to deceptive counter examples and make the

verification difficult. Therefore we should restrict are verification efforts only to the reachable states of the system. Hence we have derived invariants that should be satisfied by every instruction of 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.

### 4.6.1. Implementation of Invariants

The invariants should be satisfied by all the states of the implementation. So these invariants are checked for every possible state. The invariant property is written as

$$inv1 : [0 \leq w.pc_{basal} \leq \text{BAP}] \longrightarrow [0 \leq v.pc_{basal} \leq \text{BAP}]$$

$$inv2 : [0 \leq w.bat \leq \text{BATM}] \longrightarrow [0 \leq v.bat \leq \text{BATM}]$$

$$inv3 : [0 \leq w.pwt \leq \text{PW}] \longrightarrow [0 \leq v.pwt \leq \text{PW}]$$

$$inv4 : [0 \leq w.pc_{basal} \leq \text{BOP}] \longrightarrow [0 \leq v.pc_{basal} \leq \text{BOP}]$$

In the above $w$ is an implementation state, and $v$ is the successor of $w$. The invariant demands that the reachable states of the implementation will map to one of the specification states. The insulin pump specification is comprised of counters that are updated based on a set of conditions. In the implementation, these counters are implemented with the help of peripheral timers that counts 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. The invariants for basal and bolus modes are given as:

$$inv^{basal} = \bigwedge_{n=1}^{3} inv_n; \qquad inv^{bolus} = \bigwedge_{n=3}^{4} inv_n$$

### 4.6.2. Proof Obligations

Next we derive the proof obligations. The verification for basal and bolus modes are done separately. For the verification of the basal mode, the $Basal_{mode}$ should be high; similarly $Bolus_{mode}$ should be high when verifying the bolus mode. As the basal and bolus specifications are closely related because they use some common functions for insulin delivery, therefore the same procedure is followed to write the proof obligations for both modes. The first thing in the proof is to check if

the current and the next value of corresponding flag is same and is asserted. For example for the basal proof we will check if the current and next value of the $Basal_{mode}$ is the same and is asserted. This part of the object proof looks like the following

$$(w.Basal_{mode} \wedge v.Basal_{mode})$$

$$(w.Bolus_{mode} \wedge v.Bolus_{mode})$$

w.$Basal_{mode}$ is the current value of the $Basal_{mode}$ and v.$Basal_{mode}$is the next state value of the $Basal_{mode}$ obtained after applying the instruction function. The specification is encoded in SMT, and each state component is given the corresponding value from the implementation. For example the counter *bat* is implemented via a timer in implementation. The current value of the timer is in a register called T0TC whose value is mapped onto the current value of *bat*. The next value of *bat*is determined by the definition of *bat*. On the other hand T0TC gets its next value based on the values in T0TCR, T0MR0, and T0MCR registers. These differences are mapped by carefully writing the proofs and matching the state components in the specifications with their corresponding counterparts in the implementation. An important point to note here is that there are no significant stuttering cases in this case study. If one of the flags is high, that means the corresponding mode is the acting mode and is not stuttering. If any of the flags is low that means that mode is currently off. During the verification process, we came across some interesting cases. Despite of the fact that there are no stuttering cases, the implementation can stutter for a bunch of instruction functions. The reason for this stutter is the difference in behavior of how the specification and the implementation are expressed. A generic proof looks like following

$$s = r(w) \wedge v = impl - step(w) \wedge u = spec - step(s) \rightarrow u = r(v)$$

The actual basal proof is given below.

$$\big[(w.Basal_{mode} = 1) \wedge (v.Basal_{mode} = 1) \wedge (0 \leq w.pwt \leq \text{PW})$$
$$\wedge (0 \leq w.bat \leq \text{BATM}) \wedge (0 \leq w.pc_{basal} \leq \text{BAP})\big]$$
$$\longrightarrow$$
$$\big[(0 \leq v.pwt \leq \text{PW}) \wedge (0 \leq v.bat \leq \text{BATM})$$
$$\wedge (0 \leq v.pc_{basal} \leq \text{BAP}) \wedge (v.\text{T0TC} = v.bat)$$
$$\wedge (v.\text{T1TC} = v.pwt) \wedge (v.\text{PWMCC} = v.pc_{basal}) \wedge \wedge (v.FIO1SET = v.pv_{basal})\big]$$

T0TC and T1TC are the timer registers that contain the values corresponding to the counters *bat* and *pwt*. PWMCCis the variable in the object code that contains the number of basal pulses in the basal mode. The bolus proof obligation is given as:

$$\big[(w.Bolus_{mode} = 1) \wedge (v.Bolus_{mode} = 1) \wedge (0 \leq w.pwt \leq \text{PW}) \wedge (0 \leq w.pc_{bolus} \leq \text{BOP})\big]$$
$$\longrightarrow$$
$$\big[(0 \leq v.pwt \leq \text{PW}) \wedge (0 \leq v.pc_{bolus} \leq \text{BOP})$$
$$\wedge (v.\text{T1TC} = v.pwt) \wedge (v.\text{PWMCC} = v.pc_{bolus}) \wedge \wedge (v.FIO1SET = v.pv_{bolus})\big]$$

### 4.7. Experimental Results

Timer0 is used as *bat* with one match registers T0MR1 having value of BATM. Similarly Timer1 is used as *pwt*with two match register T1MR0 and T1MR1 having value of DCH and PW. Whenever a timer reaches a value equal to a value in any of its match registers, an internal interrupt is generated. The insulin pump receives one input, which is the request for a bolus injection and the number of units of insulin to be administered. These inputs are implemented using external interrupts of the LCP1768. We estimate that the object code corresponding to the control program has over 180 billion transitions for the basal mode and over 12 million transitions for the bolus mode.

To check the correctness of the object code control program, we use the WEB refinement proof obligations. We used the z3 [3] SMT solver for verification. The input language to the z3

solver is the SMT-LIB language [2]. The verification process involves four high-level steps. The first step is to model the object code control program as 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 instruction and hence each instruction function captures a set of transitions of the object code. The transitions corresponding to all the instructions in the program thus gives all the transitions of the TS model of the object code program. The second step is to compute the pre-conditions and post-conditions for each instruction. Pre-conditions and post-conditions are predicate conditions that program states preceding and succeeding an instruction must satisfy, respectively. The pre-conditions and post-conditions essentially determine the set of states of program preceding and succeeding an instruction. The third step is to check that each instruction in a specific mode satisfies the corresponding proof obligation for that mode. For example if the insulin pump is injecting bolus insulin, the object code that it executes must satisfy the bolus mode proof obligation. This was checked using the z3 solver [3]. If an instruction function did not satisfy any of the proof obligations, this points to a bug.

The first, and third 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 counter examples as the bugs are due to the translation process and not anomalies of the object code. The third step can be automated by checking the proof obligation for each instruction function in a loop. Each iteration of the loop would call the z3 solver to check if the instruction function satisfies the proof.

The pacemaker control program was modeled using 216 instruction functions for basal mode and 155 instruction functions for bolus mode. Each instruction function required a verification check. Therefore, the proof required 371 verification checks using the z3 solver. Each of the verification checks were completed in less than one second. During verification, we also found a number of functional bugs in the object code.

# 5. REFINEMENT CHECKING FOR STEPPER MOTOR-BASED INSULIN PUMP

## 5.1. Introduction

Insulin pump is a medical device used to treat diabetes by injecting insulin in a patient's body. The dosage of insulin is a critical factor, which if not administered accurately can lead to serious health consequences or even death, thus making insulin pumps safety critical. We present a formal verification methodology for the object code of control software used in insulin pumps. The methodology is based on the theory of Well-Founded Equivalence Bisimulation (WEB) refinement and includes formal specifications and verification proof obligation for stepper motor-based insulin pumps. The proof obligation was used to validate the object code program with millions of transitions. Our methodology effectively caught several hard to catch bugs.

## 5.2. Insulin Pumps with Stepper Motors

Stepper motors are like dc motor that take discrete steps in order to rotate. Stepper motors are widely used for their accurate positioning. These motors are a good choice for the insulin delivery because their accurate steps make up for correct number of rotations. A very effective formal methods based approach for the verification of a stepper motor is devised in [31]. The approach is also based on WEB refinement. We have extended that verification efforts to insulin pumps based on stepper motors. The functional behavior of a stepper motor expressed as a transition system (TS) as given in [31] is given below.

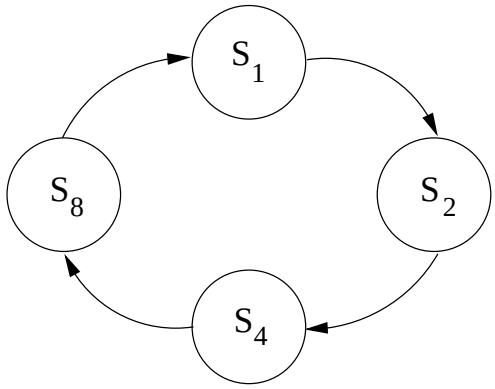


Figure 5.1. Functional behavior of stepper motor expressed as TS

We have used a 4 lead stepper motor. For a motor with 4 leads (a,b,c,d), application of the following repeating sequence of values causes the motor to rotate with values:

$$< abcd = 0001, 0010, 0100, 1000, 0001, .... >$$

All the leads of the motor are in reset state $s_0$ with values $< 0000 >$ before moving to $s_1$. When the motor should start the transition system will enter in $s_1$ from reset state $s_0$. Assignments of the states are given below:

$$< s_1 = 0001 >, < s_2 = 0010 >, < s_4 = 0100 >, < s_8 = 1000 >$$

Each value in the sequence causes the motor to rotate by a discrete angle. Stepper motor has the rating of step angle and steps/rotations. Step angle is the angle with which the motor rotates when it takes one step. Steps/rotation is the amount of rotations required to take one step. Insulin injection is done in terms of units of insulin. The unit of insulin is converted into number of units based on the rotation factor and correction factor explained in the previous chapter. Therefore the steps of stepper motor should be calculated for the required number of rotations. The motor that we have employed has the rating of 1.8° step angle (200 steps/rotation). The transition system in Figure 5.1 has four states. One transition from one state to another identifies one step. There are 200 steps required to make one rotation so in order to inject x units of insulin 50 times x number of rotations should be made.

The concept of basal and bolus insulin is same for stepper motor based insulin pump. The major difference is the way that the insulin units are converted and the unit that it is converted to. For dc motor the number of insulin units was converted to number of pulses. For stepper motor based insulin pump the number of units of insulin is first converted into number of rotations. Those rotations are multiplied with 50 and that many rotations of stepper motor will be required to inject the required insulin.

### 5.3. Specification of Stepper Motor Control for Basal Mode

In this section we present a formal specification model for a stepper motor control for basal mode. This specification model serves for the injection of fixed amount of insulin units at regular

intervals of time. The specification is expressed as a Transition System (TS). Below is the definition of TS:

**Definition 1:** A Transition System (TS) $\mathcal{M}^{\alpha}$ is a 3-tuple $\langle S, R, L \rangle$, where $S$ is the set of states, $R$ is the transition relation that defines the state transitions, and $L$ is a labeling function that defines what is visible at each state.

### 5.3.1. Basal Flag ($\mathcal{F}^{\alpha}$)

The amount of basal insulin required by a patient is programmed in the pump in terms of insulin units. The pump is by default in basal mode, and the basal flag $\mathcal{F}^{\alpha}$ is asserted until there is request for a bolus injection. The user programs basal insulin units and the frequency of basal injection in the pump.$\mathcal{F}^{\alpha}$ will be high throughout except when a bolus injection is underway.

### 5.3.2. Basal Timer ($\mathcal{T}^{\alpha}_{max}$)

The dose of insulin is injected at regular intervals of $\mathcal{T}^{\alpha}_{max}$ based on the basal timer $\mathcal{T}^{\alpha}$

$$\mathcal{T}^{\alpha\prime} \leftarrow if\ (\mathcal{T}^{\alpha} = \mathcal{T}^{\alpha}_{max}) \quad \mathcal{T}^{\alpha} = 0$$
$$else \quad \mathcal{T}^{\alpha} + 1$$

A basal injection will be initiated every time $\mathcal{T}^{\alpha}$ hits $\mathcal{T}^{\alpha}_{max}$.

### 5.3.3. Number of Rotations ($\mathcal{R}^{\alpha,\beta}_{\theta}$)

The number of insulin units is converted into number of rotations of the stepper motor. The number of rotations is calculated based on the correction factor $\mathcal{F}_{corr}$. The correction factor $\mathcal{F}_{corr}$ indicates how much rotations are required to inject one unit of insulin. Below is how the number of rotations is calculated:

$$\mathcal{R}^{\alpha,\beta}_{\theta} = N^{\alpha,\beta}_{iu} \times \mathcal{F}_{corr} \times 50 \tag{5.3.1}$$

$N^{\alpha,\beta}_{iu}$ is the number of insulin units for basal mode. The stepper motor has a 1.8° step angle (200 steps/rotation) and there are four leads so the transition system in Figure 5.1 should be executed 50 times for one rotation of the motor.

### 5.3.4. Stepper Motor Timer ($\mathcal{T}_{sm}^{\alpha,\beta}$)

A timer $\mathcal{T}_{sm}^{\alpha,\beta}$ controls the speed of the stepper motor based on the maximum speed $\tau_{rot}$

$$\begin{aligned}\mathcal{T}_{sm}^{\alpha,\beta\prime} \leftarrow & if\ (\mathcal{T}_{sm}^{\alpha,\beta} = \tau_{rot}) \quad \mathcal{T}_{sm}^{\alpha,\beta} = 0 \\ & if\ (\mathcal{N}_{rot}^{\alpha,\beta} = 0) \quad \mathcal{T}_{sm}^{\alpha,\beta} = 0 \\ & else \quad \mathcal{T}_{sm}^{\alpha,\beta} + 1\end{aligned}$$

This timer helps to maintain the speed of the stepper motor. With reference to Figure 5.1 $\tau_{rot}$ is the time taken to complete one complete sequence from $s_0$ to $s_8$. Timer $\mathcal{T}_{sm}^{\alpha,\beta}$ will ensure that this cycle takes the same amount of time always thus keeping the speed of the insulin pump constant.

### 5.3.5. Rotations Counter ($\mathcal{N}_{rot}^{\alpha,\beta}$)

$\mathcal{N}_{rot}^{\alpha,\beta}$ is the counter that counts the number of rotations to be executed by the motor, $\mathcal{N}_{rot}^{\alpha,\beta}$ is given as:

$$\begin{aligned}\mathcal{N}_{rot}^{\alpha,\beta\prime} \leftarrow & if\ (\mathcal{T}^{\alpha} = \mathcal{T}_{max}^{\alpha}) & \mathcal{R}_{\theta}^{\alpha,\beta} \\ & elseif\ (\mathcal{N}_{rot}^{\alpha,\beta} > 0 \wedge \mathcal{T}_{sm}^{\alpha,\beta} = \tau_{rot}) & \mathcal{N}_{rot}^{\alpha,\beta} - 1 \\ & else \quad \mathcal{N}_{rot}^{\alpha,\beta}\end{aligned}$$

$\mathcal{N}_{rot}^{\alpha,\beta}$ gets the value of $\mathcal{R}_{\theta}^{\alpha,\beta}$ whenever $\mathcal{T}^{\alpha}$ hits $\mathcal{T}_{max}^{\alpha}$ and a basal injection is initiated. Otherwise if it is greater than zero, it will be decremented when $\mathcal{T}_{sm}^{\alpha,\beta}$ equals $\tau_{rot}$.

### 5.3.6. Stepper Motor Delay ($\tau_{rot}$) and Step Delay ($\tau_{step}$)

$\tau_{rot}$ is the amount of time taken to complete one sequence of transition system in Figure 5.1. Stepper motor timer $\mathcal{T}_{sm}^{\alpha,\beta}$ will count to $\tau_{rot}$to mark completion of one sequence. In order to calculate the time between each step, $\tau_{step}$ is introduced. $\tau_{step}$ is the constant time taken by each transition of the transition system. Therefore, $\tau_{rot}$ should be four times of the step delay $\tau_{step}$.

### 5.3.7. Stepper Motor Control for Basal Mode ($\mathcal{S}^{\alpha}_{abcd}$)

The stepper motor control $\mathcal{S}^{\alpha,\beta}_{abcd}$ is a 4 bit vector, where $a$, $b$, $c$ and $d$ are representing 4 leads of the stepper motor. At reset, all leads have a value of zero.

$$
\begin{aligned}
&\mathcal{S}^{\alpha,\beta\,\prime}_{abcd} \leftarrow if(\mathcal{N}^{\alpha,\beta}_{rot} = 0)\\
&\quad \mathcal{S}^{\alpha,\beta}_{abcd} = s_0\\
&\mathcal{S}^{\alpha,\beta\,\prime}_{abcd} \leftarrow if(\mathcal{S}^{\alpha,\beta}_{abcd} = s_0 \wedge \mathcal{N}^{\alpha,\beta}_{rot} > 0 \wedge \mathcal{T}^{\alpha} = \mathcal{T}^{\alpha}_{max})\\
&\quad \mathcal{S}^{\alpha,\beta}_{abcd} = s_1\\
&\mathcal{S}^{\alpha,\beta\,\prime}_{abcd} \leftarrow if(\mathcal{S}^{\alpha,\beta}_{abcd} = s_1 \wedge \mathcal{T}^{\alpha,\beta}_{sm} = \tau_{step} \wedge \mathcal{N}^{\alpha,\beta}_{rot} > 0)\\
&\quad \mathcal{S}^{\alpha,\beta}_{abcd} = s_2\\
&elseif(\mathcal{S}^{\alpha,\beta}_{abcd} = s_2 \wedge \mathcal{T}^{\alpha,\beta}_{sm} = \tau_{step} \wedge \mathcal{N}^{\alpha,\beta}_{rot} > 0)\\
&\quad \mathcal{S}^{\alpha,\beta}_{abcd} = s_4\\
&elseif(\mathcal{S}^{\alpha,\beta}_{abcd} = s_4 \wedge \mathcal{T}^{\alpha,\beta}_{sm} = \tau_{step} \wedge \mathcal{N}^{\alpha,\beta}_{rot} > 0)\\
&\quad \mathcal{S}^{\alpha,\beta}_{abcd} = s_8\\
&elseif(\mathcal{S}^{\alpha,\beta}_{abcd} = s_8 \wedge \mathcal{T}^{\alpha,\beta}_{sm} = \tau_{step} \wedge \mathcal{N}^{\alpha,\beta}_{rot} > 0)\\
&\quad \mathcal{S}^{\alpha,\beta}_{abcd} = s_1\\
&elseif(\mathcal{S}^{\alpha,\beta}_{abcd} = s_8 \wedge \mathcal{T}^{\alpha,\beta}_{sm} = \tau_{rot} \wedge \mathcal{N}^{\alpha,\beta}_{rot} = 0)\\
&\quad \mathcal{S}^{\alpha,\beta}_{abcd} = s_0\\
&else \quad \mathcal{S}^{\alpha,\beta}_{abcd}
\end{aligned}
$$

where $\tau_{step}$ is the maximum delay of one step of the motor, it is a quarter of $\tau_{rot}$. An example of the step is the transition of state $s_1$ to $s_2$ of Figure 1(a) in [31]. $s_0, s_1, s_2, s_4, s_8$ are states of motor leads with values $\langle 0000, 0001, 0010, 0100, 1000\rangle$ respectively. The definition of $\mathcal{S}^{\alpha,\beta}_{abcd}$ guarantees that the motor will remain at the reset state until its time for a basal injection or a bolus request has been made. The state of the stepper motor control helps to identify the undefined states. Whenever $\mathcal{S}^{\alpha,\beta}_{abcd}$ gets a value which has not been defined in the definition of $\mathcal{S}^{\alpha,\beta}_{abcd}$, it points to a bug.

### 5.3.8. State Components for Basal Mode ($\mathcal{S}^{\alpha}$)

Insulin pump specification has millions of states because of its nature of operation. Therefore modeling such an extensive state space is impossible, rather we have identified the state components $\mathcal{S}^{\alpha}$ of the system

$$\mathcal{S}^{\alpha} = \{\mathcal{F}^{\alpha}, \mathcal{T}^{\alpha}, \mathcal{T}_{sm}^{\alpha,\beta}, \mathcal{N}_{rot}^{\alpha,\beta}, \mathcal{S}_{abcd}^{\alpha,\beta}\}$$

### 5.3.9. Basal Invariants (*inv*)

The invariants for this specification are given as:

$$\textbf{\textit{inv1:}} \qquad 0 \leq \mathcal{T}_{sm}^{\alpha,\beta} \leq \tau_{rot}$$

*inv1* signifies that for all states of the system the stepper motor timer $\mathcal{T}_{sm}^{\alpha,\beta}$ will remain between 0 and $\tau_{rot}$. This should be true for all the reachable states of the system. It also makes sure that the length of one step remains constant. If this invariant is not met and the length of one step leads or lags behind the specified limit; it can lead to unwanted insulin delivery.

$$\textbf{\textit{inv2:}} \qquad 0 \leq \mathcal{N}_{rot}^{\alpha,\beta} \leq \mathcal{R}_{\theta}^{\alpha,\beta}$$

*inv2* signifies that for all states of the system, basal timer $\mathcal{N}_{rot}^{\alpha,\beta}$ will remain between 0 and $\mathcal{R}_{\theta}^{\alpha,\beta}$. This should be satisfied by all the reachable states of the system. This invariant ensures that the number of rotations should not exceed the limit assigned. This is a critical invariant as if it is not met can result in a delay or over dose on the basal insulin.

## 5.4. Specification of Stepper Motor Control for Bolus Mode

In this section, we present a formal specification model for a stepper motor control for bolus mode. We use transition system (TS) to model the specification. TS is defined in section 5.3.

### 5.4.1. Bolus Flag ($\mathcal{F}^{\beta}$)

Bolus insulin injection is initiated by user's request. Upon request the bolus flag $\mathcal{F}^{\beta}$ is asserted and remain asserted only until the injection is completed. Once it is completed the control

is given back to the basal mode, $\mathcal{F}^{\beta}$ is de-asserted and $\mathcal{F}^{\alpha}$ is asserted. $\mathcal{F}^{\beta}$ is given as:

$$\begin{aligned} \mathcal{F}^{\beta\prime} \leftarrow & if\ (\mathcal{N}^{\beta}_{rot} > 0) && \mathcal{F}^{\beta} = 1 \\ & elseif\ (\mathcal{N}^{\beta}_{rot} <= 0) && \mathcal{F}^{\beta} = 0 \\ & else \quad \mathcal{F}^{\beta} \end{aligned}$$

### 5.4.2. Stepper Motor Control for Bolus Mode ($\mathcal{S}^{\beta}_{abcd}$)

Stepper motor control conditions for bolus mode are given as:

$$\begin{aligned} & \mathcal{S}^{\beta}_{abcd}{}' \leftarrow if(\mathcal{N}^{\beta}_{rot} = 0) \\ & \quad \mathcal{S}^{\beta}_{abcd} = s_0 \\ & \mathcal{S}^{\beta}_{abcd}{}' \leftarrow if(\mathcal{S}^{\beta}_{abcd} = s_0 \wedge \mathcal{N}^{\beta}_{rot} > 0 \wedge \mathcal{F}^{\beta}) \\ & \quad \mathcal{S}^{\beta}_{abcd} = s_1 \\ & \mathcal{S}^{\beta}_{abcd}{}' \leftarrow if(\mathcal{S}^{\beta}_{abcd} = s_1 \wedge \mathcal{T}^{\alpha,\beta}_{sm} = \tau_{step} \wedge \mathcal{N}^{\beta}_{rot} > 0) \\ & \quad \mathcal{S}^{\beta}_{abcd} = s_2 \\ & elseif(\mathcal{S}^{\beta}_{abcd} = s_2 \wedge \mathcal{T}^{\alpha,\beta}_{sm} = \tau_{step} \wedge \mathcal{N}^{\beta}_{rot} > 0) \\ & \quad \mathcal{S}^{\beta}_{abcd} = s_4 \\ & elseif(\mathcal{S}^{\beta}_{abcd} = s_4 \wedge \mathcal{T}^{\alpha,\beta}_{sm} = \tau_{step} \wedge \mathcal{N}^{\beta}_{rot} > 0) \\ & \quad \mathcal{S}^{\beta}_{abcd} = s_8 \\ & elseif(\mathcal{S}^{\beta}_{abcd} = s_8 \wedge \mathcal{T}^{\alpha,\beta}_{sm} = \tau_{step} \wedge \mathcal{N}^{\beta}_{rot} > 0) \\ & \quad \mathcal{S}^{\beta}_{abcd} = s_1 \\ & elseif(\mathcal{S}^{\beta}_{abcd} = s_8 \wedge \mathcal{T}^{\alpha,\beta}_{sm} = \tau_{rot} \wedge \mathcal{N}^{\beta}_{rot} = 0) \\ & \quad \mathcal{S}^{\beta}_{abcd} = s_0 \\ & else \quad \mathcal{S}^{\beta}_{abcd} \end{aligned}$$

$\mathcal{S}^{\beta}_{abcd}$ ensures that the motor will initiate stepping only when $\mathcal{F}^{\beta}$ is asserted and the current number of rotations $\mathcal{N}^{\beta}_{rot}$ is greater than zero.

### 5.4.3. Number of Rotations ($\mathcal{R}_\theta^\beta$)

The specification for bolus mode is similar to the basal mode except the calculation for dosage. The user programs number of bolus insulin units, which is converted into number of rotations

$$\mathcal{R}_\theta^\beta = N_{iu}^\beta \times \mathcal{F}_{corr} \times 50 \tag{5.4.1}$$

$\mathcal{R}_\theta^\beta$ is the number of rotations of the stepper motor required to inject the required number of bolus insulin units. The same stepper control that injects the basal insulin is employed to inject the bolus insulin at the same rate of $\mathcal{T}_{sm}^{\alpha,\beta}$.

### 5.4.4. Rotations Counter ($\mathcal{N}_{rot}^\beta$)

$\mathcal{N}_{rot}^\beta$ is the counter that counts the number of rotations to be executed by the motor, $\mathcal{N}_{rot}^\beta$ is given as:

$$\begin{aligned} \mathcal{N}_{rot}^{\beta\,\prime} \leftarrow & if\ (\mathcal{N}_{rot}^\beta = 0 \wedge \mathcal{F}^\beta) && \mathcal{R}_\theta^\beta \\ & elseif\ (\mathcal{N}_{rot}^\beta > 0 \wedge \mathcal{T}_{sm}^{\alpha,\beta} = \tau_{rot}) && \mathcal{N}_{rot}^\beta - 1 \\ & else && \mathcal{N}_{rot}^\beta \end{aligned}$$

### 5.4.5. State Components for Bolus Mode ($\mathcal{S}^\alpha$)

Due to an extensive state space, the system cannot be expresses as a transition system; hence we have identified the state components. The state components $\mathcal{S}^\beta$ for bolus mode are:

$$\mathcal{S}^\beta = \{\mathcal{F}^\beta, \mathcal{N}_{rot}^\beta, \mathcal{T}_{sm}^{\alpha,\beta}, \mathcal{S}_{abcd}^\beta\}$$

### 5.4.6. Bolus Invariants (*inv*)

The invariants for this specification are given as:

$$\textbf{\textit{inv1:}} \qquad 0 \leq \mathcal{T}_{sm}^{\alpha,\beta} \leq \tau_{rot}$$

*inv1* signifies that for all states of the system the stepper motor timer $\mathcal{T}_{sm}^{\alpha,\beta}$ will remain between 0 and $\tau_{rot}$. This should be true for all the reachable states of the system. It also makes sure that the

length of one step remain constant. If this invariant is not met and the length of one step leads or lags behind the specified limit; it can lead to unwanted insulin delivery.

$$\textbf{\textit{inv2:}} \qquad 0 \leq \mathcal{N}_{rot}^{\alpha,\beta} \leq \mathcal{R}_{\theta}^{\beta}$$

*inv2* signifies that for all states of the system, basal timer $\mathcal{N}_{rot}^{\beta}$ will remain between 0 and $\mathcal{R}_{\theta}^{\beta}$. This should be satisfied by all the reachable states of the system. This invariant ensures that the number of rotations should not exceed the limit assigned. This is a critical invariant as if it is not met can result in a delay or over dose on the basal insulin.

### 5.5. Formal Verification Methodology for Object Code Control Programs

In this section, we develop a methodology for formal verification of control programs of stepper motor based insulin pumps. The control program is validated with a methodology that targets at the object code level. The theory of Well-Founded Equivalence Bisimulation (WEB) refinement [50] is employed, 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 5.3. In the context of WEB refinement, both the implementation and specification are modeled as transition systems (TSs). In Section 5.3, we have developed a TS specification for insulin pump control. The object code program is also modeled as TS. The control program is comprised of a number of assembly instructions. Each assembly instruction captures the transition and alters the state of the program. The instructions are thus modeled as functions that keep a track of the program state. At boot up, all the parameters, registers, variables and the state of the processor is at reset state. The first instruction function of the program will take the reset values as the current state of the system and calculate the next state values. Therefore, each instruction will take its current state from the next state values of its predecessor instruction function, and then calculate the next state depending on the kind of the assembly instruction.

An important thing to consider here is the difference between specification TS and implementation TS (object code) when examined at a high level. The specification TS is a transition system expressed as set of relations, without a set of clearly defined states. The system progresses and changes state based on the specification. Whereas in the implementation; state of the system at any given instance is derived from state of the object code program. The object code is

comprised of general purpose, special purpose, and the peripherals registers. The implementation progresses with the help of instruction functions, therefore each instruction function should match the set of specifications. The other difference is the complexity of the implementation as compared to the specification. The specification is explicit enough in the form of equations where a single counter is expressed as a simple equation; on the other hand in the implementation this counter is implemnted with a peripheral timer with its own set of registers and other complex requirements.

The definitions for WEBs and WEB refinement are given below: In [50][52] a detailed description of WEB refinement is provided.

**Definition 4:**[50] $B \subseteq S \times S$ is a WEB on TS
$\mathcal{M} = \langle S, R, L\rangle$ iff:

(1) $B$ is an equivalence relation on $S$; and

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

(3) There exist functions $erankl : S \times S \rightarrow \mathbb{N}$,

$erankt : S \rightarrow W,$ such that $\langle W, \prec\rangle$ is well-founded, and

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

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

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

**Definition 5:**[50] Let $\mathcal{M} = \langle S, R, L\rangle$, $\mathcal{M}' = \langle S', R', L'\rangle$, and $r : S \rightarrow S'$. We say that $\mathcal{M}$ is a WEB refinement of $\mathcal{M}'$ with respect to refinement map $r$, written $\mathcal{M} \approx_r \mathcal{M}'$, if there exists a relation, $B$, such that $\langle \forall s \in S :: sBr(s)\rangle$ and $B$ is a WEB on the TS $\langle S \uplus S', R \uplus R', \mathcal{L}\rangle$, where $\mathcal{L}(s) = L'(s)$ for $s$ an $S'$ state and $\mathcal{L}(s) = L'(r(s))$ otherwise. In the above definitions, $\mathcal{M}$ is the implementation TS and $\mathcal{M}'$ is the specification TS.

The theory of WEB refinement [50] employs refinement maps, which are functions that map implementation states to specification states. The specification of insulin pump is expressed as TS encoded in the form of equations, but there are as many states in the implementation as the number of the program states the control program can take. There cannot be a formal refinement map that can be used to exract the similarities between the specification and the implementation TS. The concept of mapping each implementation state to a specification state without a refinement map

requires manual efforts to come up with proofs that unfold the similarities between the two. These proof obligations enable the comparison of implementation states and specification states, even if these states look very different. The object code program TS has many more transitions than the specification. For this case study, the object code program has more than 180 billion transitions for the basal mode, whereas the specification TS is only set of state components. 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. WEB refinement has the ability to reason about single transitions of the implementation and specification to devise a correctness proof. This can be done easily on the specification side, as the specification is only set of equations. Whereas, the object code control program TS have millions of transitions. Therefore in order to check the WEB refinement proof obligation, a decision procedure (SMT solver) is employed.

One of the challenges to apply SMT solver for insulin pump is of reachability. If all the states are considered while checking the WEB refinement proofs including those that are not reachable from the initial state of the system, it will lead to deceptive counter examples and make the verification difficult. Therefore we should restrict are verification efforts only to the reachable states of the system. Hence we have derived invariants that should be satisfied by every instruction of 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 properties are given below:

$$inv1 : [0 \leq w.\mathcal{T}^{\alpha} \leq \mathcal{T}^{\alpha}_{max}] \rightarrow [0 \leq v.\mathcal{T}^{\alpha} \leq \mathcal{T}^{\alpha}_{max}]$$

$$inv2 : [0 \leq w.\mathcal{T}^{\alpha,\beta}_{sm} \leq \tau_{rot} \rightarrow [0 \leq v.\mathcal{T}^{\alpha,\beta}_{sm} \leq \tau_{rot}]$$

$$inv3 : [0 \leq w.\mathcal{N}^{\alpha,\beta}_{rot} \leq \mathcal{R}^{\alpha,\beta}_{\theta}] \rightarrow [0 \leq v.\mathcal{N}^{\alpha,\beta}_{rot} \leq \mathcal{R}^{\alpha,\beta}_{\theta}]$$

The invariants for basal and bolus mode are given as:

$$inv^{\alpha} = \bigwedge_{n=1}^{3} inv_n; \qquad inv^{\beta} = \bigwedge_{n=2}^{3} inv_n$$

In the above $w$ is an implementation state, and $v$ is the successor of $w$. The invariant demands

that the reachable states of the implementation will map to one of the specification states. Next we derive the proof obligation. The verification for basal and bolus modes are done seperately. For the verification of the basal mode, $\mathcal{F}^{\alpha}$ should be high; similarly $\mathcal{F}^{\beta}$ should be high when verifying the bolus mode. As the basal and bolus specifications are closely related because they use common resources for insulin delivery, therefore the same procedure is followed to device the proof obligations for both modes.

Table 5.1. Proof obligation predicates

| **Predicate symbol** | **Predicate definition** |
|---|---|
| $\mathcal{P}1_{\alpha}$ | $(v.T0TC = v.\mathcal{T}^{\alpha})$ |
| $\mathcal{P}1_{\alpha,\beta}$ | $(v.T1TC = v.\mathcal{T}_{sm}^{\alpha,\beta})$ |
| $\mathcal{P}2_{\alpha,\beta}$ | $(v.R0 = v.\mathcal{N}_{rot}^{\alpha,\beta})$ |
| $\mathcal{P}3_{\alpha,\beta}$ | $(v.FIO1SET = v.\mathcal{S}_{abcd}^{\alpha,\beta})$ |

The first thing in the proof is to check if the current and the next value of corresponding flag is same and is asserted which signifies the non-stuttering case. Stuttering occurs when the implementation does not progress with respect to specification. The specification is encoded in SMT, and each state component is mapped to the corresponding value from the implementation. For example the counter $\mathcal{T}^{\alpha}$ is implemented via a timer in implementation. The current value of the timer is in a register called *T0TC* whose value is mapped onto the current value of $\mathcal{T}^{\alpha}$. The next value of $\mathcal{T}^{\alpha}$is determined by the definition of $\mathcal{T}^{\alpha}$. On the other hand *T0TC* gets its next value based on the values in *T0TCR*, *T0MR0*, and *T0MCR* registers. These differences are mapped by carefuly writing the proof and matching the state components in the specifications with their corresponding counterparts in the implementation. An important point to note here is that there are no significant stuttering cases in this case study. If one of the flags is high, that means the corresponding mode is the acting mode and is not stuttering. If any of the flags is low that means that mode is currently off. A generic proof looks like following

$$[s{=}r(w) \wedge v{=}\textit{impl-step}(w) \wedge u{=}\textit{spec-step}(s)] \rightarrow u{=}r(v)$$

The basal proof is given below:

$$\mathbf{PF}_{\alpha} : \left[ (w.\mathcal{F}^{\alpha}) \wedge (v.\mathcal{F}^{\alpha}) \rightarrow \mathcal{P}1_{\alpha} \wedge \bigwedge_{i=1}^{3} \mathcal{P}i_{\alpha,\beta} \right] \wedge \bigwedge_{n=1}^{3} inv_n$$

The bolus proof is given below:

$$\mathbf{PF}_{\beta} : \left[ (w.\mathcal{F}^{\beta}) \wedge (v.\mathcal{F}^{\beta}) \rightarrow \bigwedge_{i=1}^{3} \mathcal{P}i_{\alpha,\beta} \right] \wedge \bigwedge_{n=2}^{3} inv_n$$

The predicates $\mathcal{P}1_{\alpha}$ and $\mathcal{P}i_{\alpha,\beta}$ are given in table 5.1. T0TC and T1TC are the timer registers that contain the values corresponding to the counters $\mathcal{T}^{\alpha}$ and $\mathcal{T}_{sm}^{\alpha,\beta}$. R0 is the register in the object code that contains the number of basal pulses in the basal mode. Same kind of proof obligations is devised for the bolus mode.

### 5.6. Experimental Results

Insulin pump injects insulin in one mode at a time. This concept enforces to check the correctness of both modes separately, and hence devise separate proof obligations. We have applied our verification methodology to the object code of insulin pump with basal and bolus modes control program implemented on an ARM Cortex M3 based NXP LPC 1768. $\mathcal{T}_{sm}^{\alpha,\beta}$ and $\mathcal{T}^{\alpha}$ in the specification are implemented using two timer interrupts of LPC 1768. The insulin pump receives request for a bolus injection using external interrupts of the LCP1768. We estimate that the object code corresponding to the control program has approximately over 180 billion transitions for the basal mode and over 12 million transitions for the bolus mode.

To check the correctness of the object code control program, we use the WEB refinement proof obligations. We used the z3 [3] SMT solver for verification. The input language to the z3 solver is the SMT-LIB language [2]. The verification process involves four high-level steps. The first step is to model the object code control program as TS. This was achieved by encoding each instruction as a function in the SMT-LIB language (called instruction functions). The instruction functions essentially modifies the program state. Each instruction and hence each instruction function captures a set of transitions of the object code. The transitions corresponding to all the instructions in the program thus gives all the transitions of the TS model of the object code

program. The second step is to compute the pre-conditions and post-conditions for each instruction. Pre-conditions and post-conditions are predicate conditions that program states preceding and succeeding an instruction must satisfy, respectively. The third step is to check that each instruction in a specific mode satisfies the corresponding proof obligation for that mode. For example if the insulin pump is injecting bolus insulin, the object code that it executes must satisfy the bolus mode proof obligation. This was checked using the z3 solver [3]. If an instruction function did not satisfy the proof obligations, this points to a bug.

The insulin pump control program was modeled using 175 instruction functions for one rotation of motor. Each instruction function required a verification check. Therefore, the proof required 175 verification checks using the z3 solver. Each of the verification checks were completed in less than one second. During verification, we also found a number of functional bugs in the object code; each bug was caught in milliseconds. One of the bugs caught was due to the priority of external interrupts over internal interrupts. Stepper motor is rotated with the help of internal timer interrupts. When an external request for bolus injection is made during a basal injection, the basal injection is halted and the request for bolus is executed. This prevents basal insulin to be completed and can result in less insulin administered for the bolus mode too. This bug was introduced by setting priorities and using flags to ensure complete and accurate delivery of insulin. Another interesting bug was seen in the state of the stepper motor. The stepper motor starts from a reset $s_0$ and then cycles between $s_1$ to $s_8$. The bug was seen where one of the internal timer interrupts was not cleared correctly, halting the motor.

### 5.7. Conclusion

We have developed a formal set of specifications for insulin delivery mechanism. The specification is implemented on an embedded system and verified with the proof obligations devised using WEB refinement. During the verification process, we were not only able to identify several bugs in the implementation but also refine the specification. This verification effort is distinctive as it bridges the gap between two separate stages of software life cycle: specification and verification. Our approach can be automated easily using SMT solvers.

# 6. CONCLUSION

This chapter concludes the work in this study. Medical devices are playing an important role for critically ill patients. These devices are not only improving patient's quality of life but they are also keeping them alive. In some cases these devices are implanted in patient's body and thus errors in such a device cannot be tolerated. The dissertation focused on developing formal verification techniques for software control of safety critical medical devices. The devices focused in this dissertation are:

(a) Pacemakers,

(b) Insulin Pumps.

Pacemakers and insulin pumps are safety critical devices because bugs in any of these can result in serious health consequences or even death. Two types of bugs can be found in these devices, hardware defects and software defects. Medical device industry spends millions of dollar each year for testing of their devices. There are efficient verification techniques employed in the industry for the verification of hardware but effective methodologies are still required for the verification of software defects. Software defects are unforeseen and are often over looked by testing. Testing of software helps to find bugs but it does not ensure that the system is bug free. On the other hand verification ensures that the system is bug free in every possible situation. Hence by only testing a device there is a sound possibility that the system is prone to bugs and errors. Medical devices are recalled if bugs are found in them after they have hit the shelves. This is a huge downturn for the medical device industry. So an intelligent approach should be to properly verify the device before it is actually marketed.

In this work we have employed formal verification techniques for the verification of pacemakers and insulin pumps. The verification methodology is based on the notion of correctness. We have used the notion of WEB (Well Founded Equivalence Bisimulation). WEB helps to ensure that the high level specification matches the low level implementation. Commercially medical devices software verification is often based on Model checking and Model driven approach. There is no evidence of applying formal verification techniques at the level of object code. Bugs are often introduced when high level C code is translated into low level assembly code; these bugs are overlooked

by testing and other conventional verification techniques. Whereas, WEB helps to ensure that each and every state in specification matches every state in implementation. If there is any state in specification that does not match to any state in implementation and vice-versa, that points to a bug. An important aspect of this concept is that it also helps to refine the specification.

Our verification methodology was used to efficiently verify a control program with over millions of transitions against the specification in the form of a transition system. The methodology constituted invariants that captures the set of reachable states of 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 specification. Pacemaker and insulin pump control being a real-time system has both functional and timing requirements. Our specification 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 [26] to address the verification of timing requirements as well.

# REFERENCES


[1] U.S. Food and Drug Administration. (2015, Aug 20). List of Device Recalls [Online]. Available:http://www.fda.gov/medicaldevices/safety/ListofRecalls/default.htm.

[2] SMT-LIB. (2013, Nov). The Satisfiability Modulo Theories Library [Online]. Available: http://http://smtlib.cs.uiowa.edu/.

[3] Codeplex. (2015, Apr 13). Z3 [Online]. Available: http://http://z3.codeplex.com.

[4] The Compass - Technical Guide to Boston Scientific Cardiac Rhythm Management Products. 2007.

[5] Medtronic. (2014, July). Revo MRI SureScan Pacing System [Online]. Available: http://www.medtronic.com/patients/bradycardia/pacemaker/our-pacemakers/revo-mri-surescan/index.htm.

[6] Boston Scientific. (2016, May). Boston Scientific Advancing Science for Life [Online]. Available: http://www.bostonscientific.com/en-US/Home.html.

[7] Medtronic. (2015, Oct). Important Medical Device Safety Information [Online]. Available: http://www.medtronicdiabetes.com/customer-support/product-and-service-updates/Medtronic-Insulin-Pumps.

[8] Diabtesenet. (2016, July). Comparison of Current Insulin Pumps [Online]. Available: http://www.diabetesnet.com/diabetes-technology/insulin-pumps/current-pumps/pump-comparison.

[9] Rajeev Alur, David Arney, Elsa L Gunter, Insup Lee, Jaime Lee, Wonhong Nam, Frederick Pearce, Steve Van Albert, and Jiaxiang Zhou. Formal specifications and analysis of the computer-assisted resuscitation algorithm (cara) infusion pump control system. *International Journal on Software Tools for Technology Transfer*, 5(4):308–319, 2004.

[10] Rajeev Alur and David Dill. Automata for modeling real-time systems. In *International Colloquium on Automata, Languages, and Programming*, pages 322–335. Springer, 1990.

[11] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[12] Rajeev Alur, Radu Grosu, and Michael McDougall. Efficient reachability analysis of hierarchical reactive machines. In *International Conference on Computer Aided Verification*, pages 280–295. Springer, 2000.

[13] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times: a tool for schedulability analysis and code generation of real-time systems. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 60–72. Springer, 2003.

[14] David Arney, Raoul Jetley, Paul Jones, Insup Lee, and Oleg Sokolsky. Formal methods based development of a pca infusion pump reference model: Generic infusion pump (gip) project. In *High Confidence Medical Devices, Software, and Systems and Medical Device Plug-and-Play Interoperability, 2007. HCMDSS-MDPnP. Joint Workshop on*, pages 23–33. IEEE, 2007.

[15] David Arney, Miroslav Pajic, Julian M Goldman, Insup Lee, Rahul Mangharam, and Oleg Sokolsky. Toward patient safety in closed-loop medical device systems. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*, pages 139–148. ACM, 2010.

[16] S Serge Barold, Roland X Stroobandt, and Alfons F Sinnaeve. *Cardiac pacemakers and resynchronization step by step: An illustrated guide*. John Wiley & Sons, 2010.

[17] Gerd Behrmann, Alexandre David, and Kim G Larsen. A tutorial on uppaal. In *Formal methods for the design of real-time systems*, pages 200–236. Springer, 2004.

[18] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Developing uppaal over 15 years. *Software: Practice and Experience*, 41(2):133–142, 2011.

[19] Taolue Chen, Marco Diciolla, Marta Kwiatkowska, and Alexandru Mereacre. Quantitative verification of implantable cardiac pacemakers. In *Real-Time Systems Symposium (RTSS), 2012 IEEE 33rd*, pages 263–272. IEEE, 2012.

[20] Anthony WC Chow and Alfred E Buxton. *Implantable cardiac pacemakers and defibrillators: All you wanted to know*. John Wiley & Sons, 2008.

[21] Edmund M Clarke and E Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981.

[22] Edmund M Clarke and Jeannette M Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys (CSUR)*, 28(4):626–643, 1996.

[23] Gari D Clifford, Shamim Nemati, and Reza Sameni. An artificial vector model for generating abnormal electrocardiographic rhythms. *Physiological measurement*, 31(5):595, 2010.

[24] Robert Couch, Mary Jetha, Donna M Dryden, Nicola Hooton, Yuanyuan Liang, Tamara Durec, Elizabeth Sumamo, Carol Spooner, Andrea Milne, Kate O'Gorman, et al. Diabetes education for children with type 1 diabetes mellitus and their families. 2008.

[25] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.

[26] Mohana Asha Latha Dubasi, Sudarshan K Srinivasan, and Vidura Wijayasekara. Timed refinement for verification of real-time object code programs. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 252–269. Springer, 2014.

[27] Centers for Disease Control, Prevention, et al. National diabetes statistics report: estimates of diabetes and its burden in the united states, 2014. *Atlanta, GA: US Department of Health and Human Services*, 2014, 2014.

[28] David E Goldstein, Jack D England, Randall Hess, Sharon S Rawlings, and Beth Walker. A prospective study of symptomatic hypoglycemia in young diabetic patients. *Diabetes Care*, 4(6):601–605, 1981.

[29] Artur Oliveira Gomes and Marcel Vinicius Medeiros Oliveira. Formal specification of a cardiac pacing system. In *International Symposium on Formal Methods*, pages 692–707. Springer, 2009.

[30] Ibrahim H Ibrahim. Implantable medical devices employing capacitive control of high voltage switches, January 12 1993. US Patent 5,178,140.

[31] Shaista Jabeen, Sudarshan K Srinivasan, Sana Shuja, and Mohana Asha Latha Dubasi. A formal verification methodology for fpga-based stepper motor control. *IEEE Embedded Systems Letters*, 7(3):85–88, 2015.

[32] Kurt Jensen and Andreas Podelski. Tools and algorithms for the construction and analysis of systems. *International Journal on Software Tools for Technology Transfer*, 8(3):177–179, 2006.

[33] R Jetley and P Jones. Safety requirements based analysis of infusion pump software. *IEEE RTSS/SMDS*, pages 310–325, 2007.

[34] Raoul Jetley, S Purushothaman Iyer, and Paul L Jones. A formal methods approach to medical device review. *IEEE Computer*, 39(4):61–67, 2006.

[35] Zhihao Jiang, Allison Connolly, and Rahul Mangharam. Using the virtual heart model to validate the mode-switch pacemaker operation. In *2010 Annual International Conference of the IEEE Engineering in Medicine and Biology*, pages 6690–6693. IEEE, 2010.

[36] Zhihao Jiang and Rahul Mangharam. Modeling cardiac pacemaker malfunctions with the virtual heart model. In *2011 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 263–266. IEEE, 2011.

[37] Zhihao Jiang, Miroslav Pajic, Rajeev Alur, and Rahul Mangharam. Closed-loop verification of medical devices with model abstraction and refinement. *International Journal on Software Tools for Technology Transfer*, 16(2):191–213, 2014.

[38] Zhihao Jiang, Miroslav Pajic, Allison Connolly, Sanjay Dixit, and Rahul Mangharam. Real-time heart model for implantable cardiac device validation and verification. In *2010 22nd Euromicro Conference on Real-Time Systems*, pages 239–248. IEEE, 2010.

[39] Zhihao Jiang, Miroslav Pajic, and Rahul Mangharam. Model-based closed-loop testing of implantable pacemakers. In *Proceedings of the 2011 IEEE/ACM Second International Conference on Cyber-Physical Systems*, pages 131–140. IEEE Computer Society, 2011.

[40] Zhihao Jiang, Miroslav Pajic, and Rahul Mangharam. Cyber–physical modeling of implantable cardiac medical devices. *Proceedings of the IEEE*, 100(1):122–137, 2012.

[41] BaekGyu Kim, Anaheed Ayoub, Oleg Sokolsky, Insup Lee, Paul Jones, Yi Zhang, and Raoul Jetley. Safety-assured development of the gpca infusion pump software. In *Proceedings of the ninth ACM international conference on Embedded software*, pages 155–164, 2011.

[42] Vinay Kumar, Abul K Abbas, Nelson Fausto, and Jon C Aster. *Robbins and Cotran pathologic basis of disease*. Elsevier Health Sciences, 2014.

[43] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism 4.0: Verification of probabilistic real-time systems. In *International Conference on Computer Aided Verification*, pages 585–591. Springer, 2011.

[44] Kim G Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152, 1997.

[45] Insup Lee, George J Pappas, Rance Cleaveland, John Hatcliff, Bruce H Krogh, Peter Lee, Harvey Rubin, and Lui Sha. High-confidence medical device software and systems. *Computer*, 39(4):33–38, 2006.

[46] Shuhao Li, Sandie Balaguer, Alexandre David, Kim G Larsen, Brian Nielsen, and Saulius Pusinskas. Scenario-based verification of real-time systems using uppaal. *Formal Methods in System Design*, 37(2-3):200–264, 2010.

[47] Yang Liu, Jun Sun, and Jin Song Dong. Developing model checkers using pat. In *International Symposium on Automated Technology for Verification and Analysis*, pages 371–377. Springer, 2010.

[48] Hugo Daniel Macedo, Peter Gorm Larsen, and John Fitzgerald. Incremental development of a distributed real-time model of a cardiac pacing system using vdm. In *International Symposium on Formal Methods*, pages 181–197. Springer, 2008.

[49] William H Maisel, Michael O Sweeney, William G Stevenson, Kristin E Ellison, and Laurence M Epstein. Recalls and safety alerts involving pacemakers and implantable cardioverter-defibrillator generators. *Jama*, 286(7):793–799, 2001.

[50] Panagiotis Manolios. *Mechanical verification of reactive systems*. PhD thesis, University of Texas, 2001.

[51] Panagiotis Manolios. A compositional theory of refinement for branching time. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 304–318. Springer, 2003.

[52] Panagiotis Manolios and Sudarshan K Srinivasan. Automatic verification of safety and liveness for pipelined machines using web refinement. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 13(3):45, 2008.

[53] Panagiotis Manolios and Sudarshan K Srinivasan. A refinement-based compositional reasoning framework for pipelined machine verification. *IEEE transactions on very large scale integration (VLSI) systems*, 16(4):353–364, 2008.

[54] Anitha Murugesan, Oleg Sokolsky, Sanjai Rayadurgam, Michael Whalen, Mats Heimdahl, and Insup Lee. Linking abstract analysis to concrete design: A hierarchical approach to verify medical cps safety. In *Cyber-Physical Systems (ICCPS), 2014 ACM/IEEE International Conference on*, pages 139–150. IEEE, 2014.

[55] KM Venkat Narayan, James P Boyle, Theodore J Thompson, Stephen W Sorensen, and David F Williamson. Lifetime risk for diabetes mellitus in the united states. *Jama*, 290(14):1884–1890, 2003.

[56] Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. Unifying theories in proofpower-z. In *International Symposium on Unifying Theories of Programming*, pages 123–140. Springer, 2006.

[57] Miroslav Pajic, Zhihao Jiang, Insup Lee, Oleg Sokolsky, and Rahul Mangharam. From verification to implementation: A model translation tool and a pacemaker case study. In *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, pages 173–184. IEEE, 2012.

[58] Miroslav Pajic, Zhihao Jiang, Insup Lee, Oleg Sokolsky, and Rahul Mangharam. Safety-critical medical device development using the upp2sf model translation tool. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):127, 2014.

[59] Miroslav Pajic, Rahul Mangharam, Oleg Sokolsky, David Arney, Julian Goldman, and Insup Lee. Model-driven safety analysis of closed-loop medical systems. *IEEE Transactions on Industrial Informatics*, 10(1):3–16, 2014.

[60] Véronique L Roger, Alan S Go, Donald M Lloyd-Jones, Emelia J Benjamin, Jarett D Berry, William B Borden, Dawn M Bravata, Shifan Dai, Earl S Ford, Caroline S Fox, et al. Heart disease and stroke statistics—2012 update a report from the american heart association. *Circulation*, 125(1):e2–e220, 2012.

[61] Boston Scientific. Pacemaker system specification. *Boston Scientific*, 2007.

[62] Sana Shuja, Sudarshan K Srinivasan, Shaista Jabeen, and Dharmakeerthi Nawarathna. A formal verification methodology for ddd mode pacemaker control programs. *Journal of Electrical and Computer Engineering*, 2015:57, 2015.

[63] JD Tate, JG Stinstra, T Pilcher, and RS MacLeod. Measuring implantable cardioverter defibrillators (icds) during implantation surgery: Verification of a simulation. In *2009 36th Annual Computers in Cardiology Conference (CinC)*, pages 473–476. IEEE, 2009.

[64] Stavros Tripakis, Sergio Yovine, and Ahmed Bouajjani. Checking timed büchi automata emptiness efficiently. *Formal Methods in System Design*, 26(3):267–292, 2005.

[65] Luu Anh Tuan, Man Chun Zheng, and Quan Thanh Tho. Modeling and verification of safety critical systems: A case study on pacemaker. In *Proceedings of the 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement*, pages 23–32. IEEE Computer Society, 2010.

[66] Mark A Wood and Kenneth A Ellenbogen. Cardiac pacemakers from the patient's perspective. *Circulation*, 105(18):2136–2138, 2002.

[67] Jim Woodcock. First steps in the verified software grand challenge. *Computer*, 39(10):57–64, 2006.

[68] Jim Woodcock and Jim Davies. *Using Z: specification, refinement, and proof*, volume 39. Prentice Hall Englewood Cliffs, 1996.

[69] Yi Zhang, Raoul Jetley, Paul L Jones, and Arnab Ray. Generic safety requirements for developing safe insulin pump software. *Journal of diabetes science and technology*, 5(6):1403–1419, 2011.

# APPENDIX. LIST OF PUBLICATIONS

- Shuja, S., Srinivasan, S.K., Jabeen, S. and Nawarathna, D., 2015. A formal verification methodology for DDD mode pacemaker control programs. Journal of Electrical and Computer Engineering, 2015, p.57.

- Jabeen, S., Srinivasan, S.K., Shuja, S. & Dubasi, M.A.L., 2015. A Formal Verification Methodology for FPGA-Based Stepper Motor Control. IEEE Embedded Systems Letters, 7(3), pp.85-88.