Modeling, Design, and Analysis of a Heart + Pacemaker System

Timothy Darrah and William Banks

April 27, 2018

Abstract

Medical device development presents a unique set of challenges to the model based design methodology. While many engineering domains have well defined environments, human physiology is extremely complex and requires significant abstraction to become computationally manageable while still capturing sufficient behavior. Given the criticality of closed-loop medical devices in particular, strict safety and regulatory considerations must be enforced which further complicates the design environment. While medical devices cover a wide range of modalities, from adhesive bandages to artificial joins and surgical robots, patient critical devices such as pacemakers represent a sizable portion. An estimated 3 million individuals worldwide have an implanted pacemaker and that number is increasing by 600,000 annually. Artificial pacemakers therefore present an attractive case study. We utilize a model-based approach to developing a logical control scheme for a heart + pacemaker system. We implement the controller in Simulink®and on Arduino based hardware. We present state observers to enforce liveness and safety properties and then analyze the performance of our model on hardware.

---

1 Introduction

Many engineering domains possess well defined operational environments for their systems (automotive and aeronautical being chief examples), however human physiological models are not well developed. This results in a poor testing environment for model-based design. By comparison, every individual part of a car can be modeled for thermal and mechanical strain, electrical interference, etc. Such fidelity has not been realized for human physiology at the time of this writing.

Medical devices are broken up into 2 primary groups: open-loop and closed-loop. Open-loop devices such as the X-ray machine typically perform a diagnostic action foremost and have the benefit of an operator to 'close the loop.' This operator is a trained professional with years of education in human physiology to interpret test results and administer the appropriate treatment plan. This removes the dependence on a suitable design environment during development. With these systems, diagnostic accuracy must be ensured but safety requirements are satisfied throughout the environment and system (well defined procedures, highly trained technicians, the devices themselves, etc).

Closed-loop devices perform both a diagnostic and a therapeutic action without a human in the loop, and therefore all of the safety requirements must be satisfied by the device alone. Because these devices must diagnose and treat autonomously without room for error, the design complexity drastically increases. A newer model pacemaker, which is intuitively a very simple device, contains over 50,000 lines of code to meet these stringent safety requirements as they must be relied upon to safely and effectively operate in the body over a 5 to 7 -year period (Jiang and Mangharam [2015]). With these systems, reliably accurate diagnostics and safe treatment must be guaranteed.

1.1 Problem Definition

Any medical device must validate design goals in clinical trials with real patients. Because of the extreme safety considerations and need to obtain unambiguous data, clinical trials are limited in scope and represent an expensive, time-consuming process. Since there is no well define physiological model for testing of closed-loop devices, there exists a safety gap between the design phase and the clinical trial phase. The heart + pacemaker system is a prime example of this type of problem that has been addressed (although not to exhaustion) in recent years. Producing a heart model design environment to test a cardiac pacemaker model and deploying the model to hardware for device testing fits well within the framework of the EECE-6321 Cyber-Physical Systems course.

1.2 Goals

In this project our aim is to develop a simulation model of the human heart as well as a simulation model of the cardiac pacemaker, and then to conduct V&V activities to evaluate our models and designs. Both models are necessary because because a system does not exist in a vacuum, but in a very close relationship with its environment. The health state of a heart can be (up to a certain extent) characterized by the ECG signal it emits, so we need to ensure that the level of abstraction for the heart model is appropriate for our system, and also that our system generates appropriate output to the environment. These activities will be discussed below.

1.3 Organization

The paper is organized as follows: A review of the heart functionality and modeling methods are detailed in section (2); modeling of the heart + pacemaker system with Simulink®is discussed in section (3); design and implementation of the model onto hardware is detailed in section (4); verification and validation activities are discussed in section (5); and finally followed up with a discussion and conclusion.

2 Background

Closed-loop devices operate without direct supervision and therefore must be capable of its own decision making. These types of devices incorporate both diagnostic and therapeutic functionality. For example, the cardiac pacemaker must capture timing information from the heart muscle excitation potential and diagnose any arrhythmia prior to pacing the heart to a safe level if needed. However the heart itself is an extremely complex organ, and this process is not as simple as it sounds. The first pacemaker designs did not take into account these complexities, and now that our understanding of the heart has increase, so has the pacemaker technology. Discussed below are the intricate details of the heart and some of the pacemaker designs that have been developed.

2.1 Heart Physiology

The primary function of the heart is to pump blood through the body. This is done by contractions of the atria and ventricles. These contractions are triggered by electrical signals which, on the cellular level, are changes in electrical potential across cell membranes. The *sinoatrial* (SA) node generates an electrical signal which paces the heart naturally. This signal is conducted through the *internodal pathway* into the atrium resulting in contraction. Following this, the signal passes through the *atroventricular* (AV) node allowing blood to empty the atria and fill the ventricles. The electrical signal then quickly propagates through the now filled ventricles, causing contraction to force blood out of the heart. This entire electrical system is the natural pacing mechanism employed by the heart muscle (Chen et al. [2013]). Any abnormality in this electrical system can cause arrhythmias (improper beating) of the heart. Figure (1) illustrates the locations of the atrium, ventricle, SA node, VA node and placement of an artificial pacemaker.

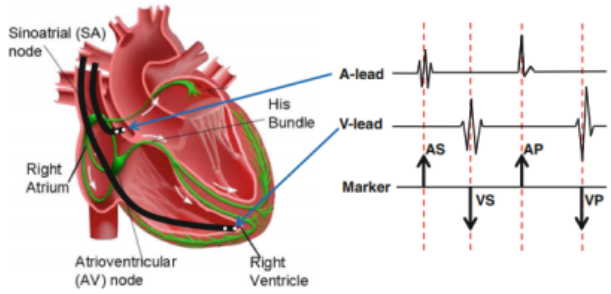

Figure 1: Heart Diagram showing pacemaker contacts and ECG timing events (Jiang et al. [2014])

2.2 Previous Modeling Methods

We reviewed several papers on the modeling, design, and analysis of heart + pacemaker systems spanning 50 years from the very first analog circuit to the very latest logic controllers.

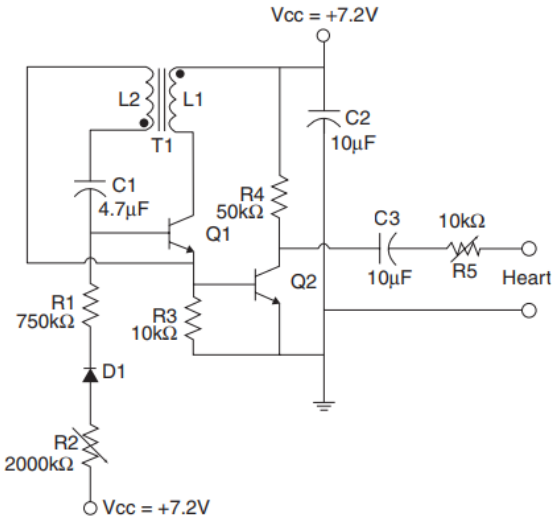

Figure 2: Analog Circuit Diagram from original Electronic Pacemaker

Figure (2) depicts the first analog circuit schematic of an electronic pacemaker (Haddad et al. [2006]). It is very simple, consisting of basic components including three resistors, three capacitors, two variable resistors, two transistors, one diode, and one transformer. For a graduate level course modeling this circuit would seem trivial, however neither of us have a true electrical engineering background and have very limited experience carrying out modeling and simulation activities.

Figure (3) is a model snippet of the pacemaker logic developed more recently by (Jiang et al. [2012]). There method is to abstract the physiological heart environment down to a computational manageable parameter space. This model describes the heart as tissue connected by a conductive pathway, (i.e. as a network of nodes). As one side activates, a signal passes via the pathway and activates the connected tissue. These can be strung together like a string of beads to describe the full signal propagation through the heart.

This is a much simpler system to model than the full physiology of a human heart. Since the pacemaker only cares about the timing between these signal events, this heart model captures all of the necessary behavior. If there is a delay in this signal, then this represents an arrhythmia and the heart must be artificially paced.

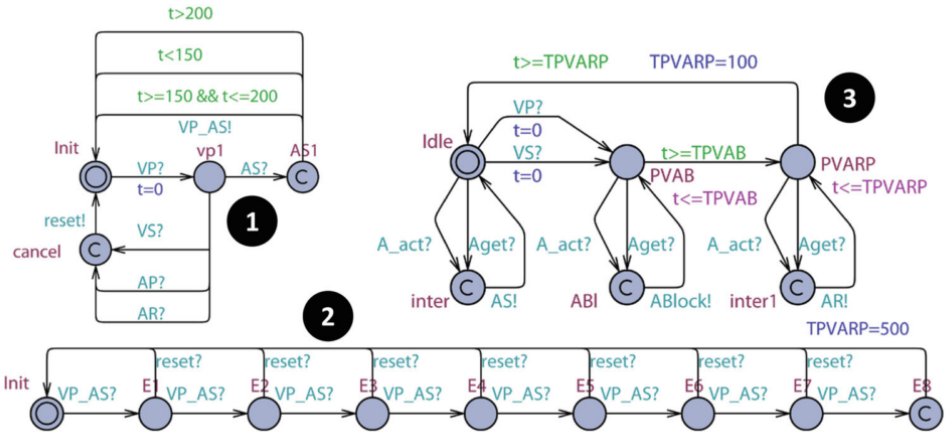

Figure 3: UPPAAL model snippet of pacing logic (Jiang et al. [2014])

Figure (4) depicts a typical ECG impulse, where the locations P, Q, R, S, and T denote critical points of interest that provide 3 basic intervals for use with timing: PQ, which physiologically is atrial depolarization, QRS, which is ventricle excitation, and ST, which is when the heart prepares for the next impulse. This waveform was generated from a MATLAB function given by (McSharry et al. [2003]) where the X-axis is time (s) and the Y-axis is millivolts (mV).

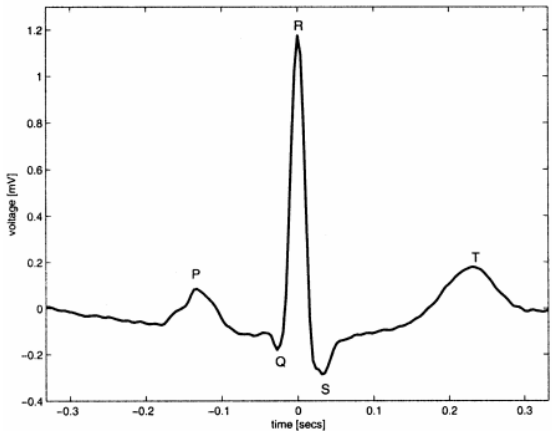

Figure 4: ECG Waveform (McSharry et al. [2003])

3 Modeling

In figure (5) we have the top-level *Simulink* model of our system. The system has 1 Simulink block modeling the heart logic and 1 Simulink block modeling the pace logic. The input block, normalHeart, uses the appropriate input *ecgVals* and *timeVals* to create the correct PQRST complex. The input block, slowHeart, uses *ecgValsSlow* instead of *ecgVal* to simulate an unhealthy heart. The *HeartModel* block takes in the array of inputs and *SampleTime*, and generates the corresponding ECG signal. This signal output becomes the *rawECG* signal input for the *PaceMakerModel*.

The internal logic of the *PaceMakerModel* is nearly identical to the *HeartModel* with one key difference. The *PaceMakerModel* has an added function to watch the timing input events from the *HeartModel*. If a

slow timing event is detected the *PaceMakerModel* will output a *pace_signal* to the *HeartModel* which will "pace" the heart into the R phase, generating the "beat" that we are all familiar with.

The blocks to the right, *PropertyCheck_pacing* and *ProperyCheck_BPM*, are safety and liveness observers. These monitor the *PaceMakerModel* to ensure that it is functioning as designed by checking that certain properties are true.

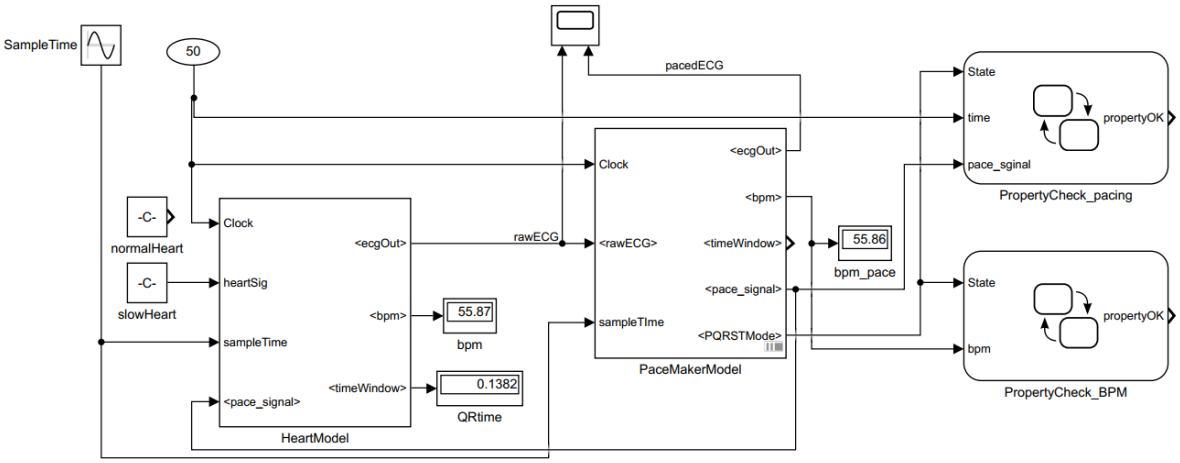

Figure 5: Heart + Pacemaker System in Simulink®

3.1 Heart Model

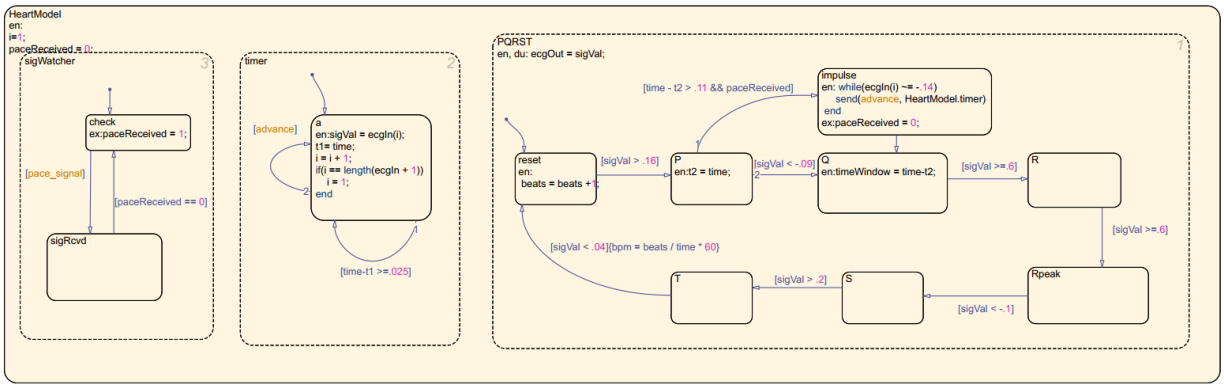

Figure 6: HeartModel Stateflow diagram

The logic of *HeartModel* is given as a *Stateflow* diagram as shown in figure (6). The left most block, sigWatcher, is watching for input pacing events from the *PaceMakerModel*. If such an event is received, it registers the event and resets the tracker, *paceReceived*, to 0. The middle block, timer, converts the input *sigVal* to an ECG signal. The right most block, PQRST, models the PQRST -complex cycle. Transitions between states occur once the correct *sigVal* is received. If a pace signal is received while still in state **P**, this indicates that the heart is not advancing through the PQRST -complex fast enough and has been paced by the pace maker. In this case, the model transitions from state **P** to state **impulse** and then to state **Q** to continue along the cycle.

3.2 Pacemaker Model

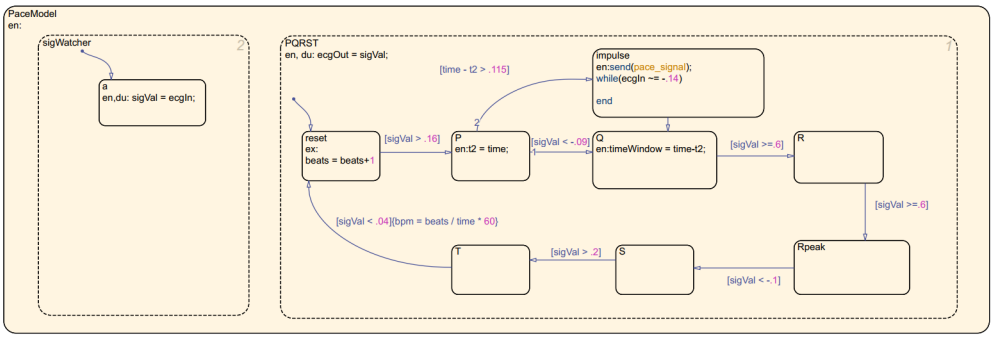

Figure 7: PaceModel Stateflow diagram

The logic of *PaceMakerModel* is very similar to that of *HeartModel* which is by design and is illustrated in figure 7. Only 2 blocks are needed because *PaceMakerModel* does not need to watch for incoming pacing events. The left block, sigWatcher, assigns the incoming ECG signal, *ecgin*, to *sigVal*. The right block, PQRST, follows the same PQRST -complex logic as *HeartModel*. If the model is in state **P** and has exceeded the maximum allowable time of 0.115 seconds, it transitions to state **impulse** which fires the pace signal for use by the Heart Model to correct the ECG signal.

4 Design

After developing the simulation models of our system under test (pacemaker) and environment (heart) the next step is to design and implement the hardware controller. We chose the *Arduino* platform because its programming environment is ubiquitous across different models, and, the *Due* and *Mega* models meet our simple requirements of being single threaded, and containing analog resolution of at least 10 bits. The *Due* is unique among any of the *Arduino* boards in that it contains a 32bit ARM microprocessor rather than an 8bit AVR based microcontroller. The *Due* is suited for the heart since it contains a 12 bit digital-to-analog (DAC) converter, which will give us 4096 values to characterize a single ECG impulse. The *Mega* controls the pacemaker logic, and has a 10 bit analog-to-digital (ADC) converter. We could have used the *Due* for both since it also contains an ADC, but we could not have used two *Mega* controllers since the *Mega* model does not have a DAC. The *Mega* is nearly half the price of the *Due*, so we went with one of each.

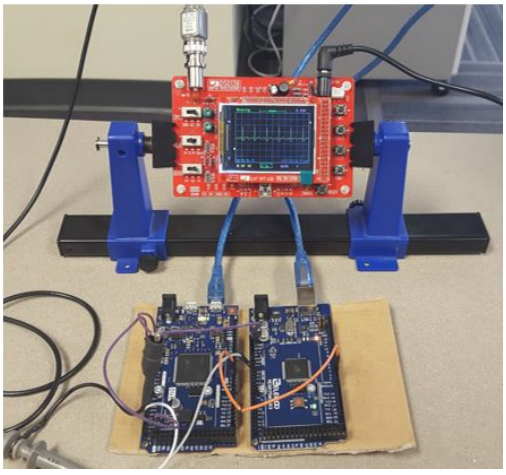

Figure 8: Hardware Setup

Figure 8 shows the 2 units hooked up to an oscilloscope for testing (left: *Due* "heart", right: *Mega* "pacemaker").

Since the ECG signal has 5 distinct phases, these phases must be contained within our simplified design. The signal on the left of figure (9) is output by the heart, and is an adequate abstraction of the signal depicted in figure (4). On the right is how this signal looks like to the pacemaker, and there are some key points to be made. First, the signal on the right is of a slow ECG signal that has been paced, this is evident by the gradual slope from the peak of the P phase leading to a sudden drop (where the pace signal was fired and subsequently received). The signal on the left is of a normal ECG signal, which is evident again by the down slope of the P phase. Second, not only is the number of bits in the converters different, but the I/O reference voltages are different as well. This explains why the Y-axis scales are different, and presented a unique challenge that will be discussed in section (6).

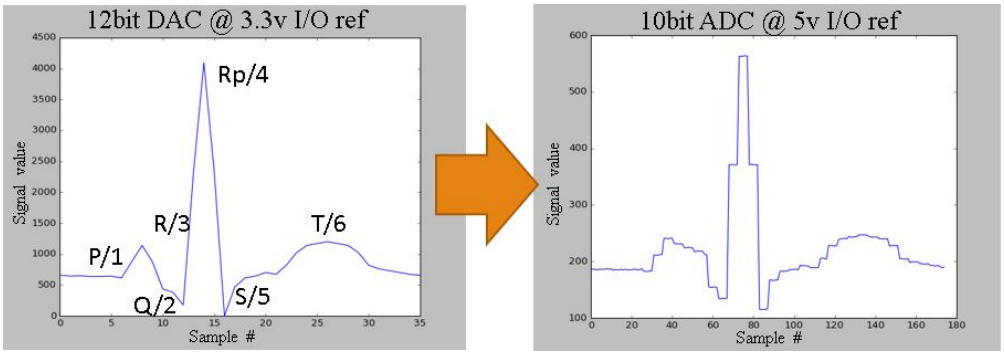

Figure 9: ECG Signal from Heart (left) to Pacemaker (right)

4.1 Heart Device

Using a similar approach to model the heart functionality as (McSharry et al. [2003]), we decided to abstract the heart as a time-series signal generator which outputs an ECG signal on the DAC1 (SAM3x PB16) output pin of the *Due* (figure (9[left])). The heart takes only one input on digital pin 2 (SAM3x PB25) configured as a RISING edge interrupt which is the *Pace_Signal*. When the signal is received, the heart "jumps" the waveform series to the end of the Q phase and returns to the main on the first value of the R phase. As mentioned before, this activity is an abstraction of the way the heart truly works, and is suited for our purposes.

Within the heart we have two signal vectors, one for a "normal" ECG signal, and one for a "slow" ECG signal. A defining characteristic that distinguishes these two signals is the transition time from the Q to R phase. For a normal signal, this transition is approximately 105 - 110 ms, and anything above 115 ms is considered "slow". Since each signal vector is for one cycle, the signals can be composed to generate bradycardia (slow heart rate using slow signal only), arrhythmia (fluctuation between slow and normal), or a normal heart rate as mentioned earlier.

Figure (10) is a code snippet showing the implementation of heart arrhythmia on the heart device. This was implemented with two random variables, on lines 59-60. The first RV determines which signal will be selected and the second RV determines the number of times this signal will be repeated. The numbers were partially chosen arbitrarily; we wanted a sufficient range of values to maximize the randomness for the *ecgTypeRV*, and we also didn't want such a high range that a signal would be repeated for a substantially long time. Also as can be seen on lines 69 and 82, there is a 26 millisecond delay in signal output. This delay time is related to the number of signal points in our vectors, so if we decided to double the number of points in them (i.e. double the resolution), this delay time would be halved. We did not follow a formal approach to derive this relationship, however this relationship became apparent through our literature review as well as trial-and-error during the modeling phase of development.

```
56 //************************************************************************** MAIN ARHYTHMIA
57 void MainArhythmia()
58 {
59     ecgTypeRV = random(1000) - 500;
60     timesRepeatRV = random(70) % 7 + 1;
61     if(ecgTypeRV >= 0)
62     {
63         for(int repeat = 0; repeat < timesRepeatRV; repeat++)
64         {
65             while(j < len)
66             {
67                 analogWrite(DAC1,newEcgVals[j]);
68                 j++;
69                 delay(26);
70             }
71             j = 0;
72         }
73     }
74     else
75     {
76         for(int repeat = 0; repeat < timesRepeatRV; repeat++)
77         {
78             while(j < len2)
79             {
80                 analogWrite(DAC1,newEcgValsSlow[j]);
81                 j++;
82                 delay(26);
83             }
84             j = 0;
85         }
86     }
87 }
88
```

Figure 10: Implementation of heart arrhythmia in C++

4.2 Pacemaker Device

The pacemaker device is designed as a finite state machine with 8 states corresponding to the 5 phases of an ECG signal plus **R-peak**, as well as an initialize **State A**, and an error **State E**. The device contains the following functions:

- Poll()
- Step()
- Pace()
- SetState(State)
- UpdateStateVariables(&cycleCounter, &stateCounter, timesArray)

The main loop of the device Polls the input signal line, which is the ECG signal coming from the heart, then Steps the state machine. Initially, the device is in **State A**, and when an ECG signal is detected it will then change to **State P** and return to the main. Subsequent state changes will call the SetState function, which advances to the next state (line 155), and resets the timer (lines 160/166). See figure (11) below. Then, the UpdateStateVariables is called, passing a counter for recording the number of cycles to track (tracks 10 cycles before resetting, this is for information purposes only), its *stateCounter* (the number of times the device has consecutively remained in a given state after a step), and its *timesArray* (holds the last 10 *stateCounter* values).

```
69 //******************************************************** PACE
70 void Pace()
71 {
72     paced = true;
73     if(millis() - lastPace > 879)
74     {
75         digitalWrite(PACE_SIGNAL_PIN, HIGH);
76         errorTime = millis();
77         while(reading > 160 || reading < 120)
78         {
79             Poll();
80             if(millis() - errorTime > 20)
81             {
82                 state = E;
83                 paced = false;
84                 break;
85             }
86         }
87         digitalWrite(PACE_SIGNAL_PIN, LOW);
88         lastPace = millis();
89     }
90     return;
91 }
```

```
151 //******************************************************** SetState
152 void SetState(int nextState)
153 {
154     //PrintHeader();
155     state = State(nextState);
156     if(!timerReady)
157     {
158         t2 = millis();
159         duration = (t2-t1);
160         timerReady = true;
161         //PrintBody();
162     }
163     if(timerReady)
164     {
165         t1 = millis();
166         timerReady = false;
167     }
168     return;
169 }
```

Figure 11: Pace() and SetState() Functions

The Pace function (figure (11[left])) is only called from within the Step function (figure 12) *IFF* the device is in **State Q** *AND* either the maximum alloted time has elapsed (line 200) *OR* the stateCounter has exceeded the maximum count (line 205). If Pace function is called, it first checks to see if it is too soon to send the **pace_signal** (line 73), and then it will fire the signal if it is not. After which, it waits for the heart to respond by continuously polling the input line until the loop condition is satisfied or until it has been waiting for longer than 20ms, after which it will transition to **State E** (error state) and break out of the loop (lines 77 - 84). This is the only place where **State E** can be entered, and it will exit to **State A** on the very next step. **State E** should never be reached, however it is included for completeness. In **State R** the BPM is calculated (line 235) as well.

```
196     if(state == Q)
197     {
198         //Serial.println("Q");
199         qC++;
200         if(millis() - t1 > 140 && !triggered)
201         {
202             Pace();
203             triggered = true;
204         }
205         else if(qC > 29 && !triggered)
206         {
207             Pace();
208             triggered = true;
209         }
210         if(triggered || (reading < 140 && reading > 130))
211         {
212             UpdateStateVariables(q, qC, timesq);
213             if(paced)
214             {
215                 SetState(state+1);
216             }
217         }
218         return;
219     }
```

```
220     if(state == R)
221     {
222         //Serial.println("R");
223         if(triggered)
224         {
225             triggered = false;
226         }
227         rC++;
228         if(reading > 550)
229         {
230             UpdateStateVariables(r, rC, timesr);
231             SetState(state+1);
232             beat++;
233             if(beat == 12)
234             {
235                 bpm = int(60 / double((millis() - beatTime) / beat) * 1000);
236                 //Serial.print("BPM: ");
237                 //Serial.println(bpm);
238                 beat = 0;
239                 beatTime = millis();
240             }
241         }
242         return;
243     }
```

Figure 12: Step() Function Snippet

5 Analysis

The analysis activities for this project includes validation of our model by enforcing liveness and safety properties with state-machine observers, and verification of the hardware implementation and timing analysis for various portions of the code.

5.1 Model Analysis

We evaluated several methods to perform the Verification and Validation portion of our project before settling on our actual approach. First, literature suggested that the tool *UPPAAL* is well suited for time-based systems (Jiang et al. [2012]). Because *UPPAAL* presented a totally new toolset to learn, and with limited time, we did not feel confident that we could learn and implement it effectively. Second, since we built the Heart+Pace model in *Simulink*, we looked into *Simulink Design Verifier*. This is a robust toolsuite that allows complete tracking of requirements, automatic report generation, formal verification, static code analysis and other functions to Verify and Validate your system. Again, we didnt feel we would have enough time to learn how to implement this method.

We settled on using the technique described in (Balasubramanian et al. [2011]). This paper outlines how a simple state automata can be used to check for defined properties. These can be inserted into *Simulink* as *Stateflow* models. The paper describes a simple paradigm to construct these models consisting of 'Scopes' and 'Patterns'. Scopes describe when a particular property should hold during execution, and Patterns defines what conditions must be satisfied in that time. An example Scope and Pattern is shown here.

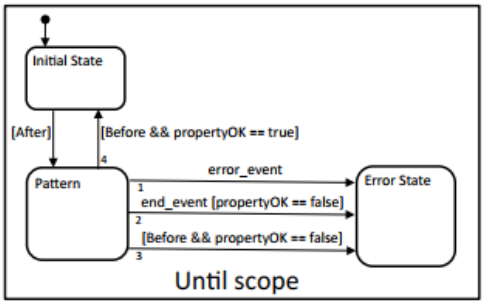

Figure 13: Scope automaton example

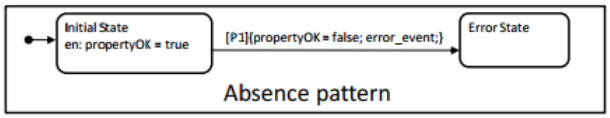

Figure 14: Pattern automaton example

5.2 Liveness Property Validation

We define a particular liveness property as follows: *PaceMakerModel* must always fire a *pace_signal* if time spent in state **P** exceeds 0.115 seconds. The property is implemented on the model level by specifying a Pattern within a Scope. This is represented as a *Stateflow* model in *Simulink* as shown in figure (16). Its behavior is described as follows:

The checker receives 3 inputs. The input variable *State* is the enumeration of the state *PaceMakerModel* is in: None = 0, **reset** = 1, **P** = 2, **Q** = 3, **R** = 4, **R_peak** = 5, **S** = 6, **T** = 7, and **impulse** = 8. The input variable *time* is from the clock used by the *HeartModel* and *PaceMakerModel*. The input variable *pace_signal* is the enumeration of the *PaceMakerModel pace_signal* event.

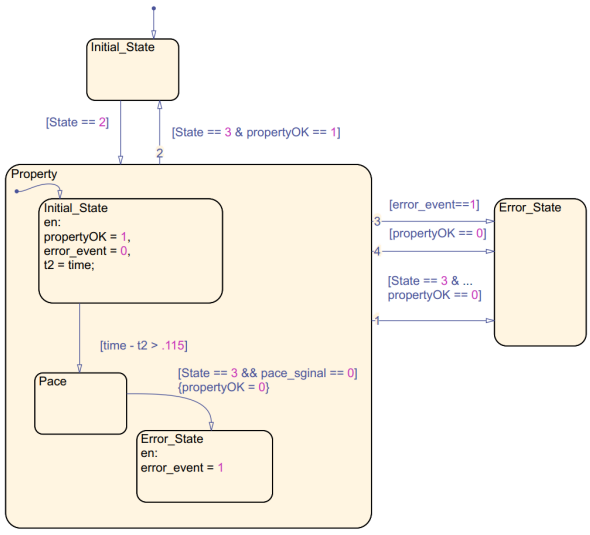

Figure 15: Stateflow diagram of liveness property enforcer

First, the checker is in **Initial_State** and is waiting for *PaceMakerModel* to enter State **P**. Once the guard condition, [**State** == 2], is true (2 corresponds to **P**), the transition into **Property** occurs and now waits in **Initial_State**. It starts keeping track of time spent in State **P** by setting: $t2 = time$. While time is ticking on, it waits for $time - t2 > .115$ and transitions into **Pace**. This transitions reflects that *PaceMakerModel* must have also entered the **Pace** state. The expectation is that the *PaceMakerModel* will fire a *pace_signal* and then enter State **Q**. This is monitored by the transition $[State == 3 \ \ \& \ \ pace\_signal == 0]$. If both conditions are true, the property has not been held and $PropertyOK = FALSE$. This variable can be monitored to show that the property is upheld. Similarly, if the property is violated (say through testing), the time that *PropertyOK* switches to FALSE can be used to determine when and therefore where in the model it occurs.

5.3 Safety Property verification

We define a particular safety property as follows: *PaceMakerModel* must never allow *BPM* to exceed 68 beats-per-minute (BPM). Recall that BPM is calculated at the end of the PQRST cycle (beat indexes in state **reset** and BPM is calculated during transition from state **T** to **reset**.

This monitor functions in a very similar way to the liveness monitor by using the same Pattern/Scope paradigm. It takes in 2 input variables. The input variable, *State* captures the enumeration of the current state *PaceMakerModel* is in. The input variable *bpm* holds the BPM calculated at that time cycle.

First, the monitor waits in **Initial_State** and checks the input *State* values. If it detects that the *PaceMakerModel* is in state **T**, then the guard $[State == 7]$ is TRUE and it transitions into **Pattern** where it waits in **Initial_State**. The monitor begins to watch *BPM*. If it exceeds 68 BPM, the monitor sets $PropertyOK = FALSE$ and the monitor transitions out of **Pattern** and into **Error_State**. The variable *PropertyOK* can be used in the same manner as previously described to demonstrate verification of this property and for testing purposes.

5.4 Hardware Implementation

Detailed below are the results of three different cases: 1) normal ECG signal (figure 17; 2) slow ECG signal (figure 18; and 3) paced ECG signal (figure 19. Each figure depicts the BPM (A), the *stateCounter* (B), the

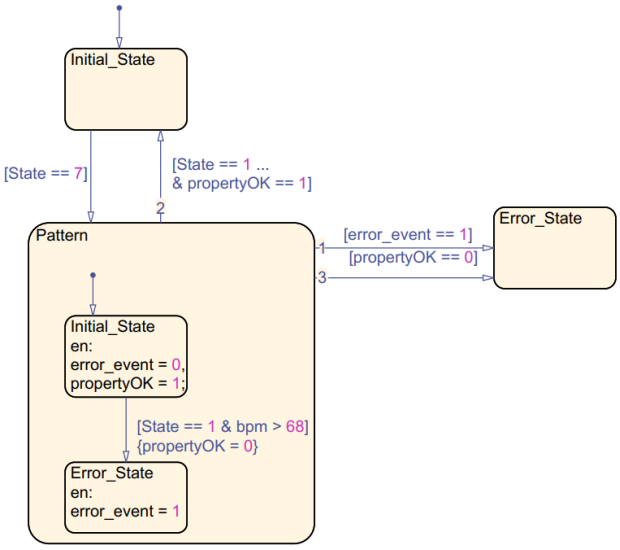

Figure 16: Stateflow diagram of safety property enforcer

Q phase of the signal on the scope (C), the time spent in each state in milliseconds (D), and finally some random variations can be seen in (E). We can see in the first figure that the machine remains in state 2, corresponding to Q, for about 128ms, and its *stateCounter* remains around 25-26 steps. This is our healthy heart. One key thing to take away from this (not highlighted in subsequent figures) is that there is always some level of randomness (E) that is nearly impossible to eliminate, even on single threaded microcontrollers. Should our system require millisecond level accuracy, another implementation would be needed, as we can't guarantee much less than 5 millisecond accuracy.

Figure 17: Normal ECG Signal. A:BPM, B:stateCount, C:Qstate duration (ms), D:stateTime, E:random variations

Next is our slow heart. These 2 signals, while from the same prototype, are quite different in their timing

characteristics. We can see that its BPM is 17 beats lower than a normal heart (A), and that it remains in state Q for a substantially longer amount of time, 91 *stateCounts* and 466 milliseconds. Interestingly enough, the time ratios and *stateCount* ratios are exactly 3.64 (466/128 == 91/25), and so we can say with certainty that the slow heart stays in state Q 3.64 times longer than the normal heart.

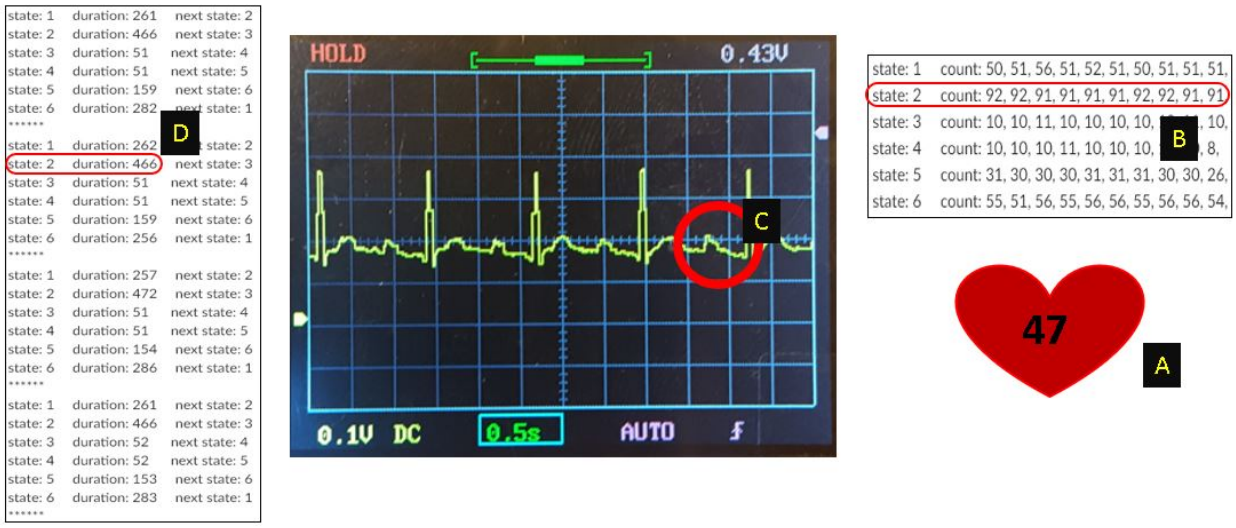

Figure 18: Slow ECG Signal. A:BPM, B:stateCount, C:Qstate duration (ms), D:stateTime

Finally we have our paced heart, with a BPM of 59 (A), within acceptable range (±3 BPM) of our target BPM of 60. Looking at the *stateCounter* (B), it went from 91 to 30, only 5 steps off of our original heart rate, and its duration is 154 milliseconds, less than 30ms off of the original. With these numbers in mind, it is also important to know that our benchmark ECG gave a BPM that was 4 beats *over* our target, and therefore this ECG signal is closer to ideal than the original normal heart.

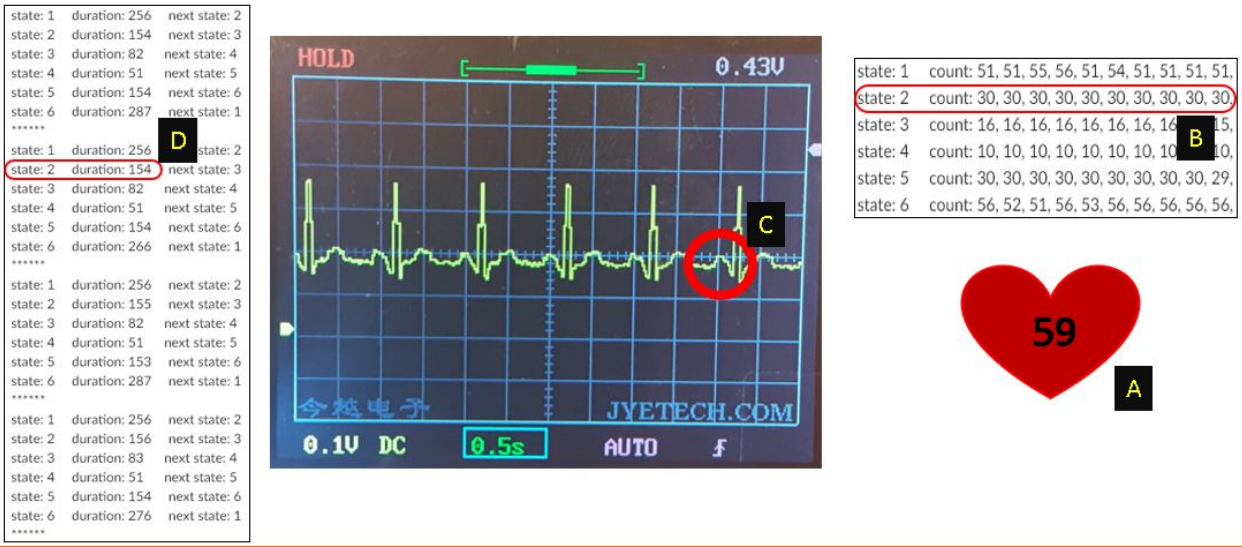

Figure 19: Paced ECG Signal. A:BPM, B:stateCount, C:Qstate duration (ms), D:stateTime

5.5 Hardware Performance

We also performed execution timing analysis on the pacemaker to identify the true sampling frequency, and to see if there were any results of interest. First, looking at figure 20 (top) we time just the execution of the main, disregarding the preprogrammed 5ms delay and come up with 112 microseconds. Then, we time everything including the delay, expecting a result of 5112 microseconds, but come up with 5128 microseconds. This "phantom" 16 microseconds still remains a mystery, and goes to show the complexities involved in designing real-time systems.

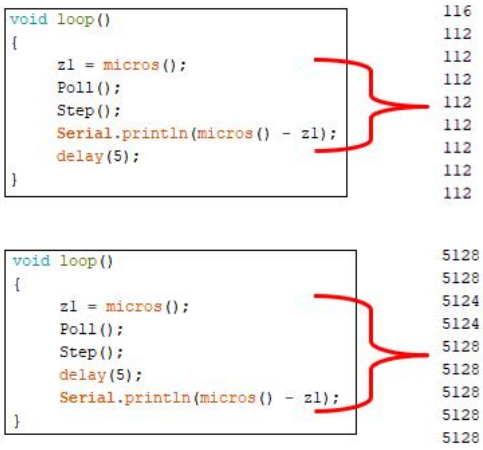

Figure 20: Execution Timing Analysis

Code available at: https://github.com/darrahts/PaceMaker

6 Discussion

We faced several challenges with this project, which ultimately pushed us to the next level by overcoming them and figuring them out. It is frustrating when accepted answers on popular forums from the 'admins' themselves include try pressing the reset button before you upload, try it a few times to figure out the timing. I remember back in the army working on the mike model Blackhawk, a fly-by-wire helicopter with all digital instrumentation. Sometimes we would do what we called a "mike model" reset when we couldnt figure out a persistent error that troubleshooting results made no sense for. Most of the time, the error went away. Again, safety critical system, life and death impact, and we have to reset it to make an error go away? These are common issues that plague engineers and are often times inescapable.

Also a challenge was how to deal with all the different scales. Yes, it turns to a simple mapping conversions problem, but getting to that point of realization was a challenge. Not only was the resolution of the converters different, but their I/O reference voltages were different as well. Since we havent dealt with this before, we had to work through it very carefully.

Direct coding versus code deployment present another unexpected challenge. We spent a lot of time trying to figure out how to get our model deployed, and were able to deploy very simple led blinking examples, but when it came to dealing with time, we couldn't figure out how to implement that with code deployment. *Arduino* uses a built in function called millis(), and we couldnt add custom code to *Simulink* that would make this work. So we decided to code the hardware from scratch, which, at first, was a huge leap, but ultimately paid off in terms of dealing with these challenges and understanding the timing properties of these systems, as well as implementing a FSM on hardware, which we had never done before. If we were able to direct deploy the code, we would have missed out on a huge learning opportunity.

Lastly, not all challenges were particularly complicated but nevertheless resulted in deleterious effects.

While implementing the BPM property checker, we found that the BPM calculating function initially spikes to 140 BPM before quickly reaching a steady state at the designed rate. The BPM calculator is a simple function: $BPM = beats/time * 60$. However, at time less than 1, the quotient inflates before resuming the normal behavior. The function is correct and the function inputs are correct, so how to handle this? A low-pass filter could be applied to smooth out the high frequency spike but this requires additional components to fix (and added complexity which is something to avoid). After much trial and error, it occurred to us to simply move the execution of the BPM calculation to the end of the heart and pace PQRST cycle. Instead of calculating the BPM during **Rpeak**, we calculate BPM during the transition from **T** to **Reset**. This allows time to increment enough to not cause the initial spike.

7 Conclusion

In this project we set out to address the high-level problem of designing systems that must interact with the human body which is an extremely complex environment and models of the human physiology are not well developed. The specific problem of the heart + pacemaker system fit well within this context and within the framework of our class, so we chose to model, design, and analyze this system using *Simulink* and two microcontrollers. In addition, this exposed us to common issues in cPS design, as discussed above. Two key requirements of our pacemaker were that it did not over pace the heart and that it paced the heart when necessary. We have demonstrated these properties hold using formal methods implemented in *Simulink*, as well as in hardware. Our results show that our system successfully paces a slow heart to within range of our goal, and that it should never enter an error state.

References

D. Balasubramanian, G. Pap, H. Nine, G. Karsai, M. Lowry, C. Psreanu, and T. Pressburger. Rapid property specification and checking for model-based formalisms. *Proceedings of the International Workshop on Rapid System Prototyping*, pages 121–127, 2011.

T. Chen, M. Diciolla, M. Kwiatkowska, and A. Mereacre. A simulink hybrid heart model for quantitative verification of cardiac pacemakers. *Proceedings of the 16th International Conference on Hybrid Systems*, pages 131–136, 2013.

S. A. P. Haddad, R. P. M. Houben, and W. A. Serdijn. The evolution of pacemakers. *IEEE Engineering in Medicine and Biology Magazine*, 25:38–48, 2006.

Z. Jiang and R. Mangharam. High-confidence medical device software development. *Foundations and Trends in Electronic Design Automation*, 9:309–391, 2015.

Z. Jiang, M. Pajic, and R. Mangharam. Cyber-physical modeling of implantable cardiac medical devices. *in Proceedings of the IEEE*, 100:122–137, 2012.

Z. Jiang, M. Pajic, R. Alur, and R. Mangharam. Closed-loop verification of medical devices with model abstraction and refinement. *International Journal on Software Tools for Technology Transfer*, 16:191–213, 2014.

P. McSharry, G. Clifford, L. Tarassenko, and L. Smith. A dynamical model for generating synthetic electrocardiogram signals. *IEEE Transactions on Biomedical Engineering*, 50, 2003.