Milestone 3 - Report

(Muhammad Fahad Asghar(masghar), Kwame Owusu(owusua), Henry Garant(henrygar), Max Brown(maxbrown))

Project Roadmap

- **Mbed Model Set up**

The model was set up as shown below. We connected the two Mbeds on the breadboard and with the heart model we also inculcate two LEDs. Each of them lights up when a warning is generated in either of the two scenarios given to us. The wireless module was connected to the heart as shown.

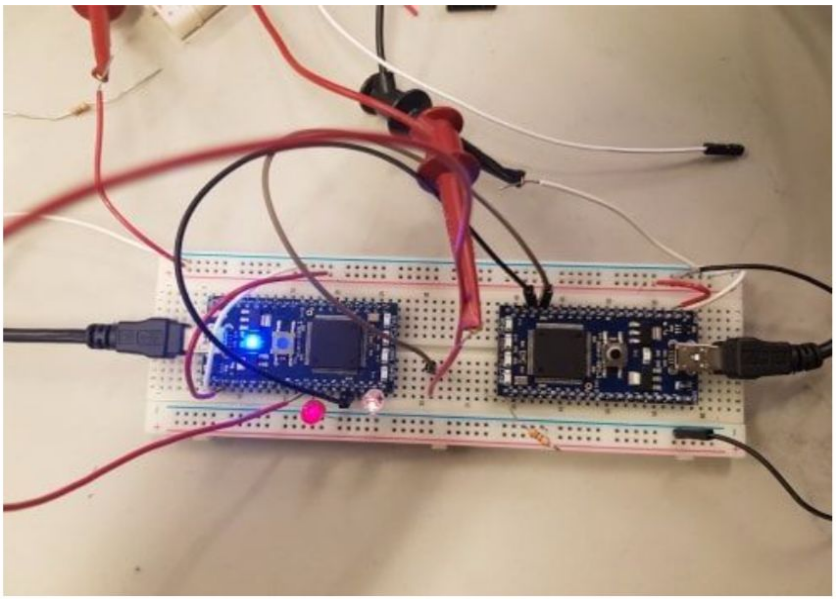


Figure.1

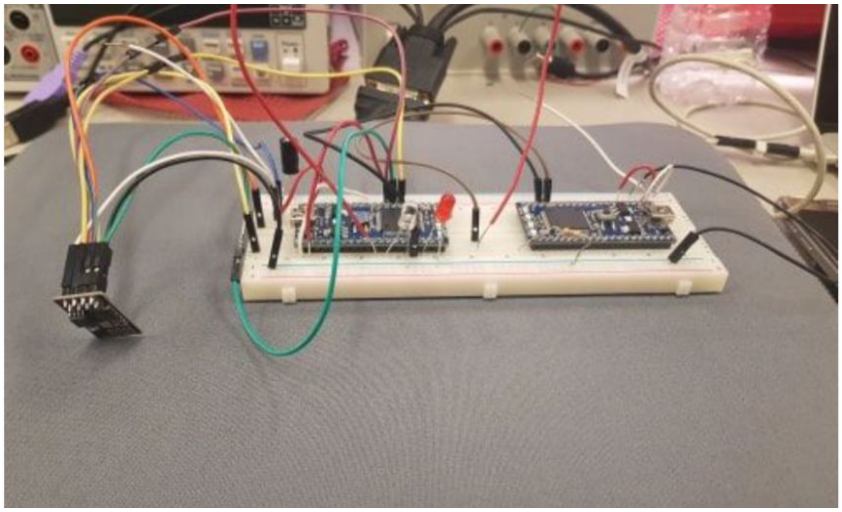


Figure.2

- **UPPAAL Model**

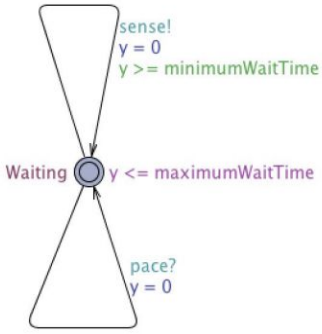
We started off the project by creating UPPAAL models for the Heart and PaceMaker in VVI mode using UPPAAL software for timed automata. Specifications of the model were derived from Boston Scientific's system specification for a pacemaker. The model consists of two timed automata: one is a Random-Heart, and the other is a Ventricle controller. The Random-Heart mimics any heart behaviors (e.g., beating just randomly, in a given range.). A Random-Heart in VVI mode receives only pacing signals to the ventricle and sends signals to the controller when a (spontaneous) heartbeat occurs. The Ventricle controller observes the heart by sensing the heart signal and responds by sending a pacing signal to the heart if the heart fails to beat by itself for a certain amount of time.

Some of the bounds that we used for our heart and pacemaker models were basically the following:

LRI: This is the lower rate limit. This determines the length between a Ventricular event (V) and the following ventricular pace and then makes a decision about lower bound of the heart rate.
URL: This is the upper rate limit and it decides the time between a ventricular event and the ventricular pace that is generated, thus, deciding the upper bound of the heart.
VRP: This is the ventricular refractory period. This is a time interval following a ventricular event during ventricular neither senses or triggers pacing. This is necessary to prevent use of a single resource by more than one process.

Below mentioned are our models:

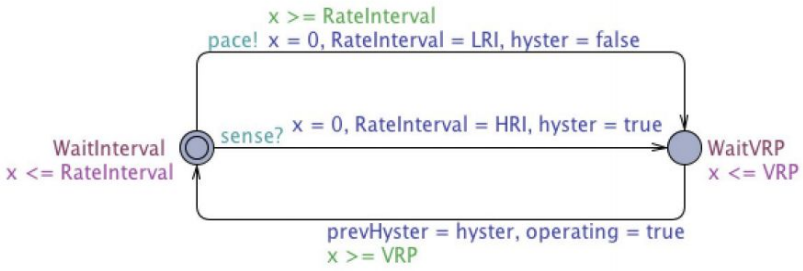
Heart:



The entire goal of the heart model is to provide testing for the Ventricle model by mimicking a heart that requires a pacemaker. The single state of the heart, Waiting, is where the model waits for a pacing event from the Pacemaker or sends a sensing event to the Pacemaker. The sensing event sent from the Heart model is between the time constraint bounds of the minimumWaitTime(LRL) and the

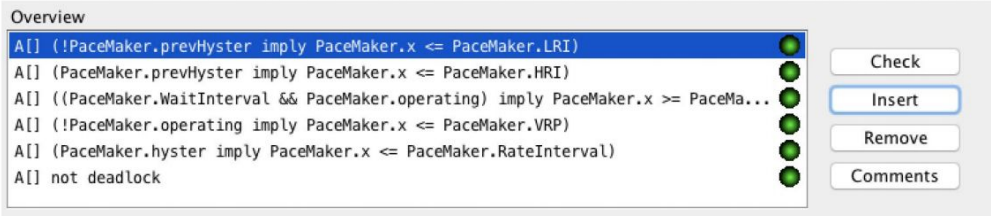
maximumWaitTime(URL). After an event is sent or detected the Heart reenters the Waiting state after resetting it's internal clock, y, for timing.

Ventrical Controller Model



The entire goal of the VentricleController model is to replicate a Pacemaker in VVI mode. The Model has two states, WaitInterval and WaitVRP. WaitInterval state waits for a sense or pace event to be fired. Obviously, if a heart sense event is not detected within the time of the RateInterval, then the VentricleController sends a pacing signal to the heart and the new RateInterval is LRI. Observe that this transition means the Heart is not in hysteresis pacing. When the heart is in hysteresis pacing, then the High Rate Interval(HRI) must be applied. Whenever an event is detected in the VentricleController model, the internal clock, x, is reset for timing consistency. The next state is WaitVRP, where the Pacemaker shall not interrupt or detect signals. In WaitVRP, the VentricleController is waiting for time equivalent to VRP to begin operating again. Finally, observe that it is critical for the VentricleController to know if it was previously in hysteresis; therefore, the model 'remembers' via the use of a boolean variable prevHyster.

To make sure that our model works correctly and how we intended it to, we identify and check 6 verification queries. They are defined below along with what they were used for and how we verified them.



1. ***A[] (not deadlock)*** :This query ensures there is no deadlock in the model. This is especially important for the Pacemaker because deadlock could result in not properly regulating the heart as it would need to be for a patient ALWAYS.

2. ***A[] (!PaceMaker.prevHyster imply PaceMaker.x <= PaceMaker.LRI)*** :This query ensures that when the hysteresis is disabled i.e the heart is working properly, the pacemaker value of RateInterval = LRI and in the case when the heart does not function properly, it triggers a pacemaker response. This is done before the expiration of the RateInterval period. In this case LRI = RateInterval.
3. ***A[] (PaceMaker.prevHyster imply PaceMaker.x <= PaceMaker.HRI)*** :This query ensures that when the heart is in hysteresis, the pacemaker makes use of the value of HRI that we pre-define. This, in turn, causes the pacemaker to generate the pace before the RateInterval period expires. Thus helping the heart function normally.
4. ***A[] (!PaceMaker.operating imply PaceMaker.x <= PaceMaker.VRP)*** This query ensures that if the PaceMaker is not operating then the PaceMaker is in VRP period. This property is necessary to show that the PaceMaker is always active and ready for signal(operating) unless it is in VRP period, in which case the PaceMaker shall not interfere with the heart.
5. ***A[] ((PaceMaker.WaitInterval && PaceMaker.operating) imply PaceMaker.x >= PaceMaker.VRP)*** After an event, The VentricleController should not be sensing events i.e. in VRP period. This query ensures that the VentricleController satisfies the VRP period time requirement. This is done by testing if in the WaitInterval state that the internal clock has already waited for VRP period amount of time.
6. ***A[] (PaceMaker.hyster imply PaceMaker.x <= PaceMaker.RateInterval)*** This query ensures that the pacemaker does not fire when a sensing event occurs within the rate interval.

- **Code Generation and Model Implementation**

Following our UPPAAL model we proceeded towards the implementation of our models from our UPPAAL model after verifying queries. Making use of two Mbeds, one on which we implemented our heart model and the other on which we implement the pacemaker model and the heart monitor as two threads, we generate code for our models. Following this We publish the average heartbeats and alarms on the cloud using MQTT. The cloud application stores them and show them on a webpage.

We were able to make the two mbeds communicate using the *DigitalIn pace* and *DigitalOut beat* o n both Mbeds. This process is also defined along the way in the code explanations and the general overview of the different parts of the code above.

Pacemaker and Heart Monitor

In this file we generate the code for the pacemaker from our Uppaal model alongside the heart monitor. We start off by setting the default rate interval as the LRI value. This is followed by the declaration of guards for the VRP, pacemaker function to start and the LEDs that indicate the abnormal heart-beat scenarios. Setting all the values that we had in our Uppaal model, eg Sense, Pace count etc as 0 to start the working of the pacemaker. After this we declare the semaphores to prevent deadlocks and so that the resources are not being used simultaneously.

This is followed by the declaration of RTOS timers for the waiting of VRP, intervals, the heart rate monitor and resetting our statistics. After this we initialize our polling timer to check if the heart had sent

a ‘sense’ signal. This is followed by the method *poll_input* to check if the heart mbed had sent out a sense signal. This is followed by declaration of different methods to calculate the heart rate for the given window. This is called every PERIOD seconds which we in our code are specifying as 60 seconds. This method for the monitor sends the data to the MQTT server and gives us a signal on the mbed. Within tis we check for conditions such as if the pacemaker is being called two often we send an alarm and set the red Led as on and if the heart is beating too fast, ie rate is greater than the URL (window/1.8). Here 1.8 is our maximum pulse width. If this condition holds true we again Alert and send a signal to the Led 2, which is the transparent LED. Following this we define a method, *pace_timer,* which is called once our timer expires (when maximum wait time defined above is exceeded) and we need to pace the heart. In this we check if the if we are waiting for the VRP or not. If we are not waiting we send a signal to the heart, start the VRP timer and clear the pace signal and update our statistics that we defined in the start (Look at the code). This is followed by another method *wait_VRP_timer(),* which is called when the VRP expires and we have to start a new heart cycle. It sets *waitVRP* and *paceStarted* to false and by using the semaphore, set *stopVRP* to true and increment the heart beats

The while loop has two purposes, and it executes only when waitVRP time has been exceeded, that is, the pacemaker is now ready to wait for a heartbeat or if enough time has elapsed, send a pace signal to the heart. The other role is two flash the alarms if an irregularity has occurred, such as the heart beating too quickly or a high frequency of pace signals were sent.

Heart Model

We start off by defining a boolean value - *beatAgain* to make the heart beat again and set the RTOS timers again to track the polling pace and our heartbeat. This is followed by the declaration of a method *getHeartBeatTime()* that gives us a randomly generated heartbeat interval between the minimum 100ms and the max 1900ms values defined. After this we are defining a polling method *poll_pace* that checks to see if the pacemaker has sent a signal every *POLLING_INTERVAL*ms. After this the *heartBeat* method sends a signal to the pacemaker telling it that the heart is beating normally. We set the beat vale to 1 which we send out to the pacemaker via DigitalOut and then sets the guards setting the *beatAgain* as true and then releases it.

The heart’s main loop continuously checks whether beatAgain is true. If true, then the heart is ready to beat. A function returns a random time between 100ms - 1900ms and starts the heartbeat timer with that interval. beatAgain is set to false, ensuring that there is only one beat per interval. Finally, after a quick wait, the beat voltage is set back to 0.

- **Cloud Application and MQTT**

Data Collector: Mbed Heart Rate Monitor
Average Heart Rate data and pacemaker mode are sent from the heart rate monitor simulator running on mbed. The data is sent via MQTT publishing to the data ingestor.

Data Ingestor: Google Cloud Pub/Sub for MQTT

Google Cloud Pub/Sub for MQTT was used to create a MQTT broker. From there, we create a topic to publish and subscribe to monitor messages. The topic is *cis-441-final-project.* I n this model, the data gets published to the topic via MQTT. Then, a subscriber(Google Cloud Function) receives an MQTT message.

Data Pipeline: Google Cloud Pub/Sub → Google Cloud Function → Google Cloud BigQuery → Python Flask Server → FrontEnd.

Google Cloud Pub/Sub receives an MQTT message and triggers a Google Cloud Function. The Google Cloud function parses the message and uploads relevant data to storage, BigQuery. The structure of data for the table in BigQuery is also below. Once the data is updated from the MQTT message, the data is presented on the Frontend via Chart.js. The graph of avg_hr data can be found on the frontend at https://cis-441-final-project.appspot.com/.

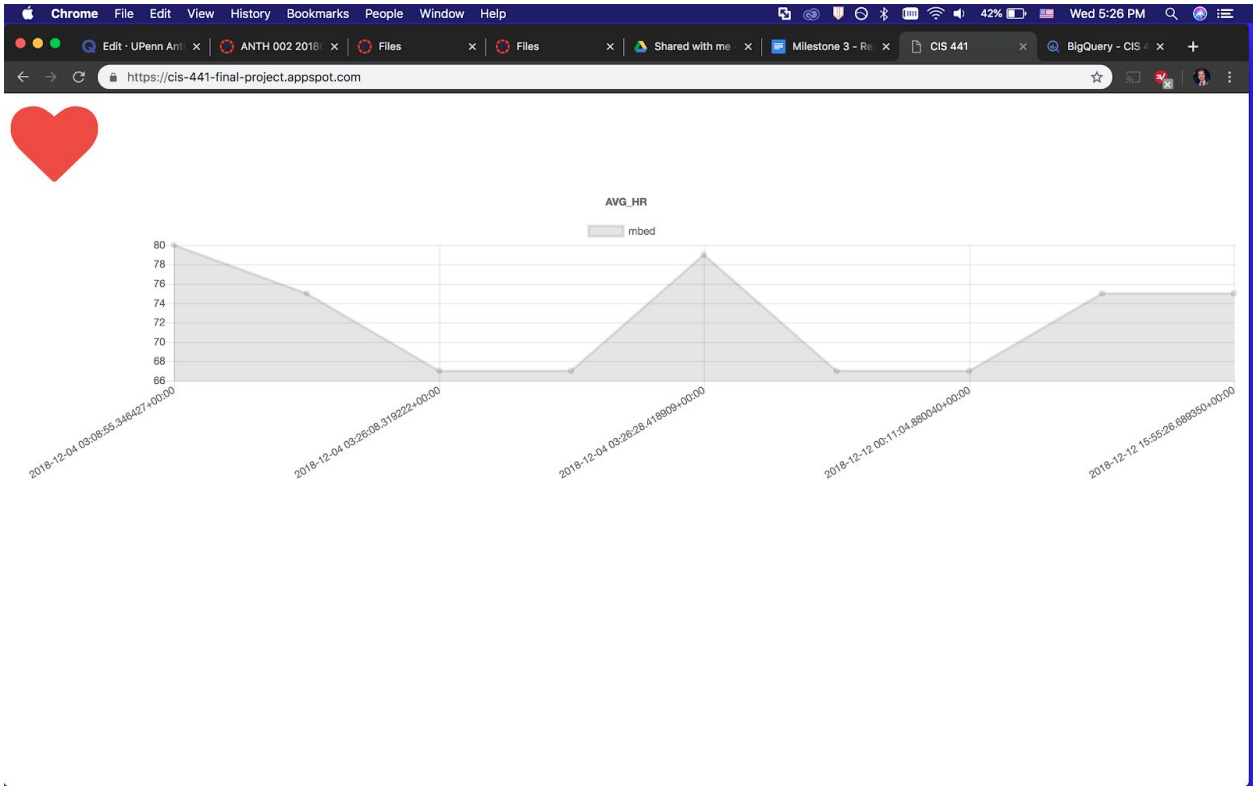
Frontend: Chart.js -> https://cis-441-final-project.appspot.com/

Chart.js allows easy graphing of BigQuery table data. Using Python Flask Server, an embedded graph is created that can be displayed on any webpage. The frontend web page is served using Python 3.7 and Flask framework.

Python Flask Server:

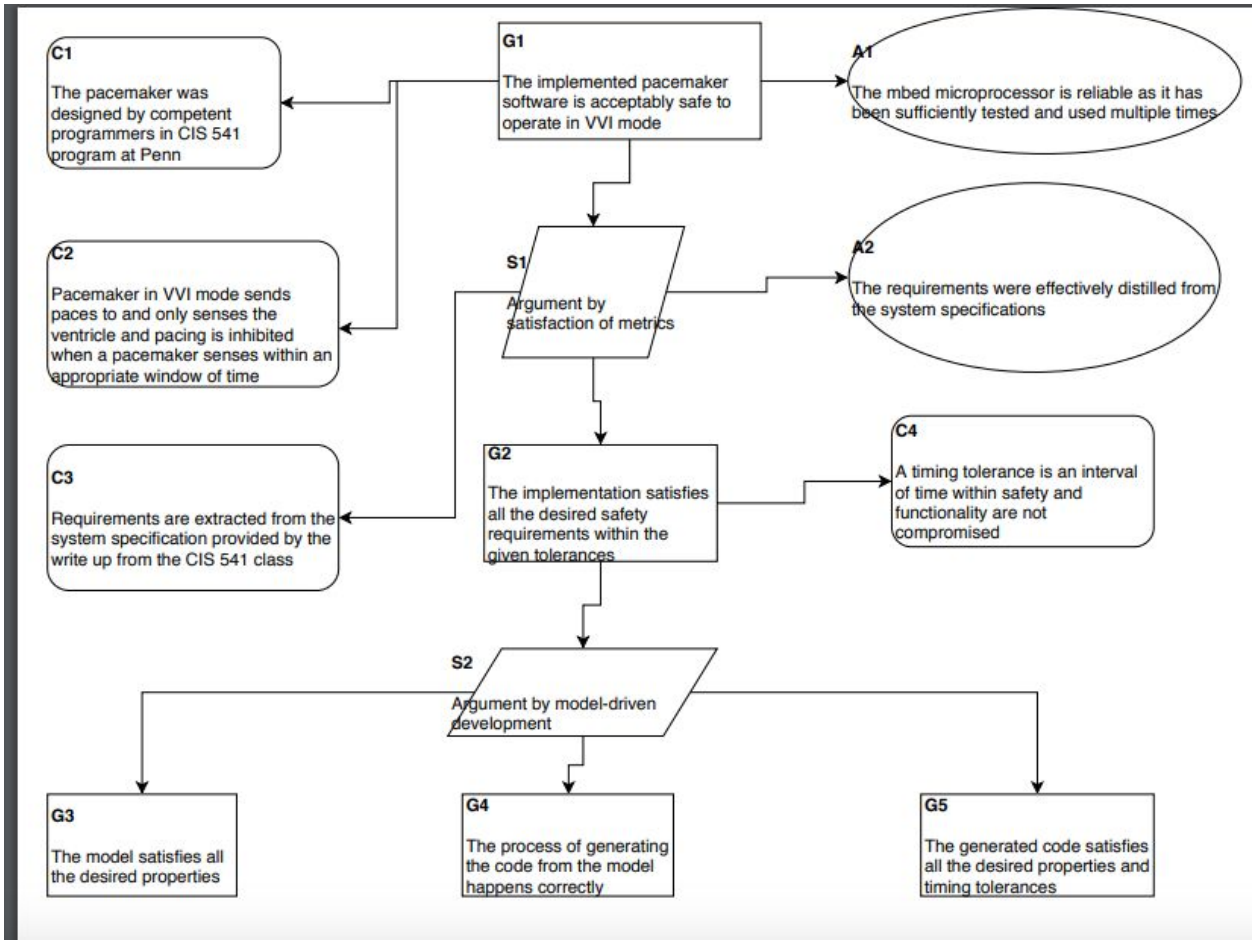
```
@mqtt.on_connect()
def handle_connect(client, userdata, flags, rc):
    print("MQTT Connect!")


@mqtt.on_message()
def handle_mqtt_message(client, userdata, message):
    data = dict(topic=message.topic,payload=message.payload.decode())
    emit('hrData', {'data': data})
    print(data)

def callback(message):
    print('Received message: {}'.format(message))
    message.ack()

def query_hr():
    client = bigquery.Client()
    query = "SELECT timestamp, avg_hr, mode FROM heart.raw_data_hr ORDER BY timestamp"
    query_job = client.query(query)

    return query_job.result()


@app.route('/')
def hello():
    mqtt.subscribe('cis541/hw-mqtt/26013f37-10000015100000161000001710000018/echo')
    results = query_hr()
    return render_template("heart.html", results=results)
```

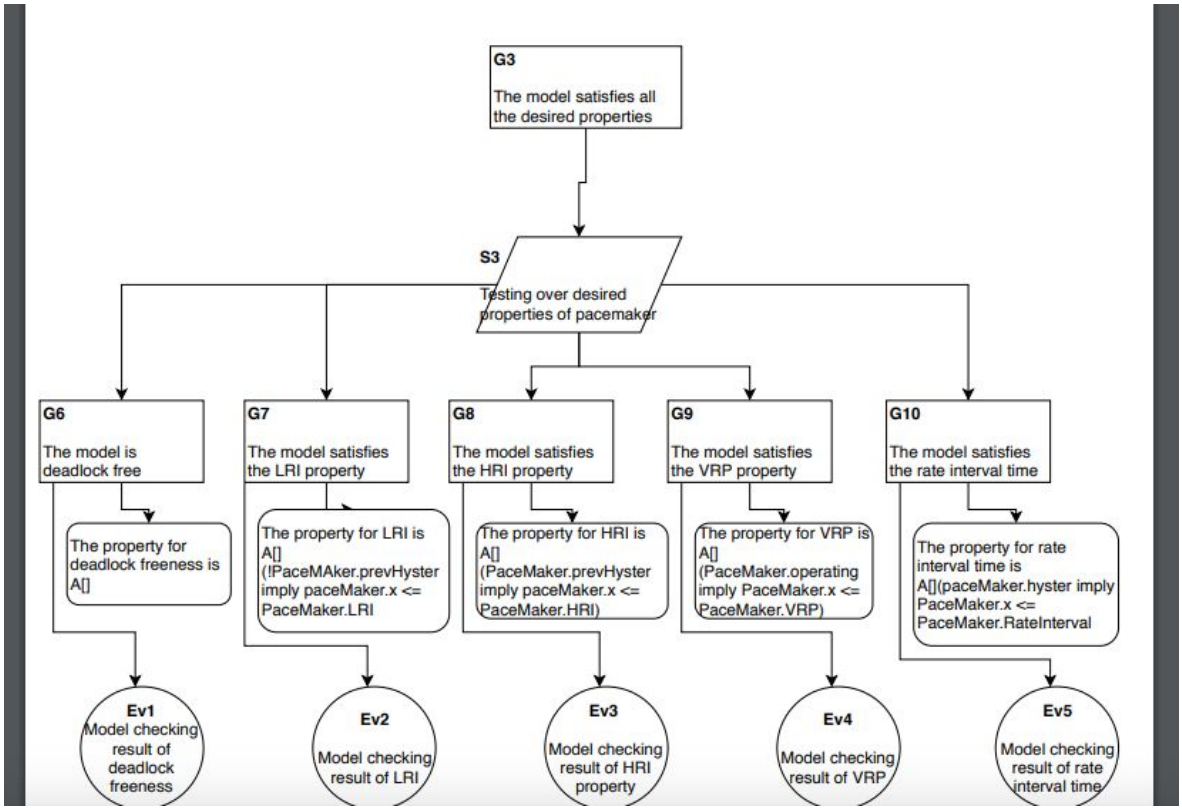
Frontend:

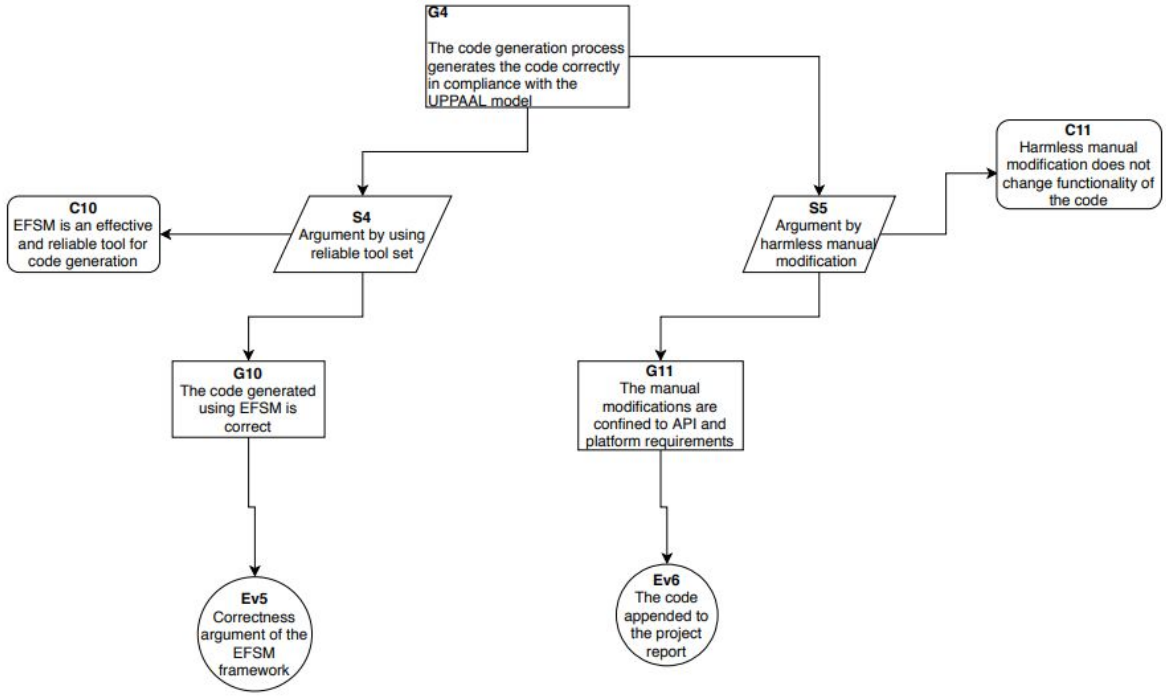


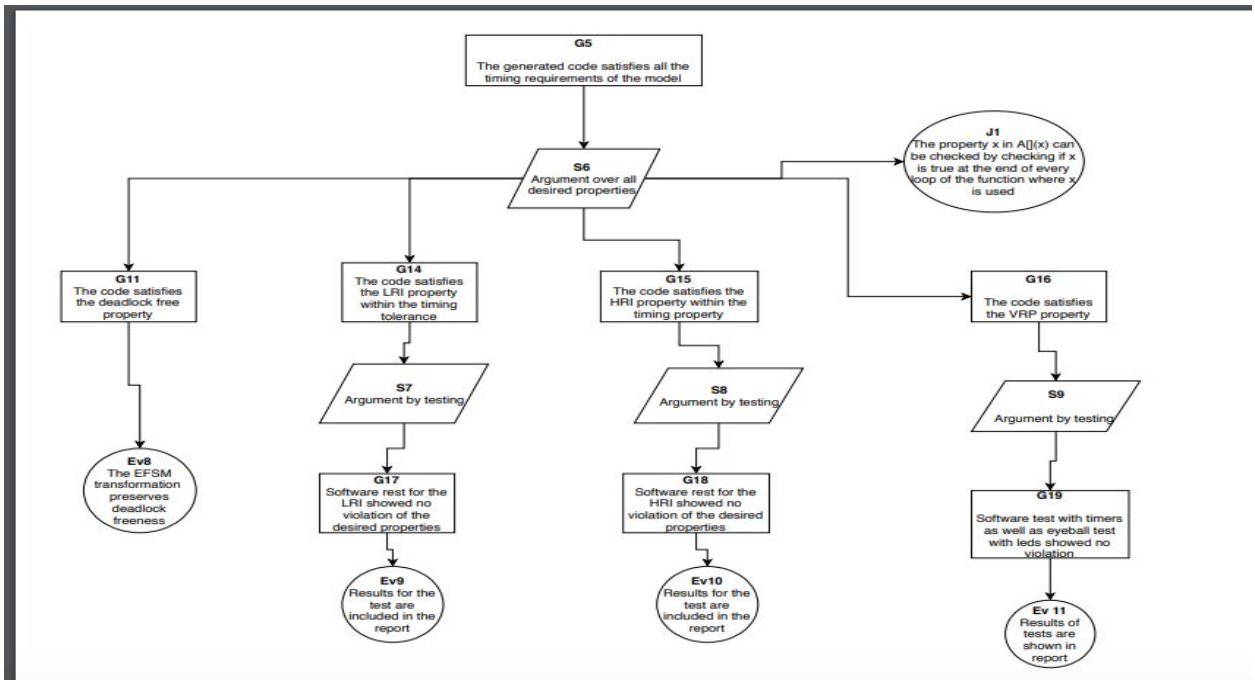
Finally we tested our model and It can be observed that when the heart is beating too fast we get an alert. This happens 2 times. There are 6 alerts that tell us that the heart beats are 72.6bpm. And finally 2 alerts showing the times when the pacemaker is required frequently.

Assurance/Confidence Case

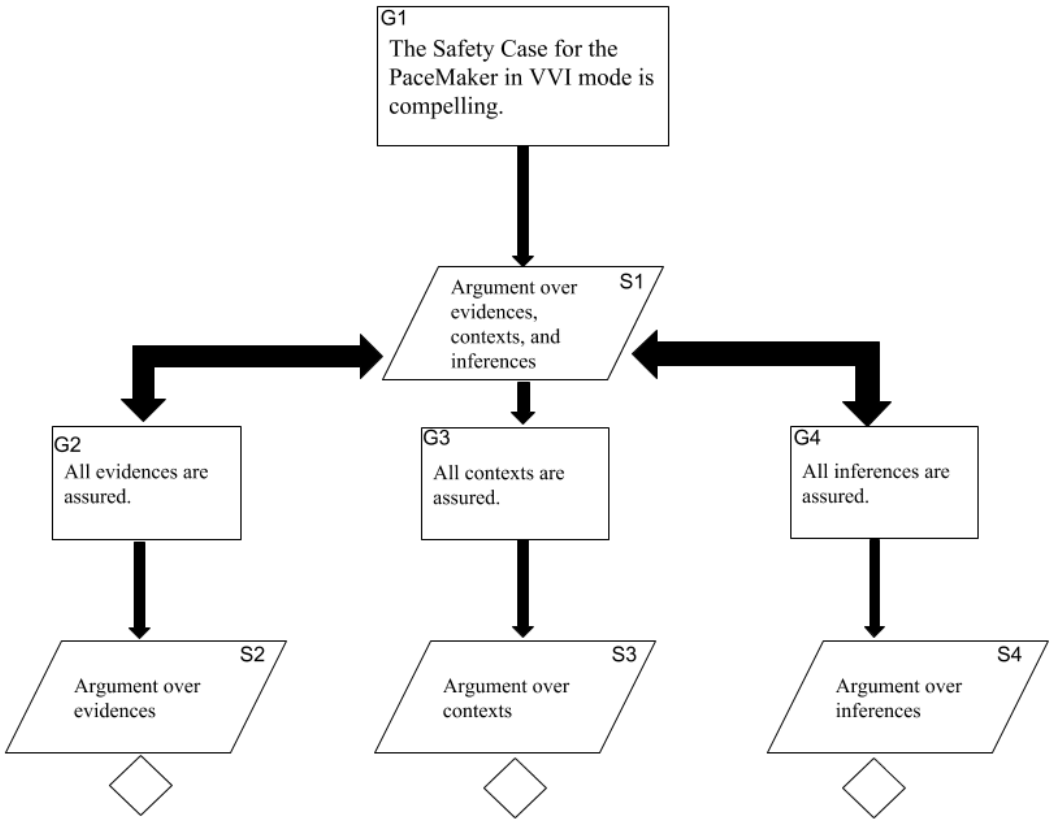

C1
The pacemaker was designed by competent programmers in CIS 541 program at Penn
G1
The implemented pacemaker software is acceptably safe to operate in VVI mode
A1
The mbed microprocessor is reliable as it has been sufficiently tested and used multiple times
C2
Pacemaker in VVI mode sends paces to and only senses the ventricle and pacing is inhibited when a pacemaker senses within an appropriate window of time
S1
Argument by satisfaction of metrics
A2
The requirements were effectively distilled from the system specifications
C3
Requirements are extracted from the system specification provided by the write up from the CIS 541 class
G2
The implementation satisfies all the desired safety requirements within the given tolerances
C4
A timing tolerance is an interval of time within safety and functionality are not compromised
S2
Argument by model-driven development
G3
The model satisfies all the desired properties
G4
The process of generating the code from the model happens correctly
G5
The generated code satisfies all the desired properties and timing tolerances

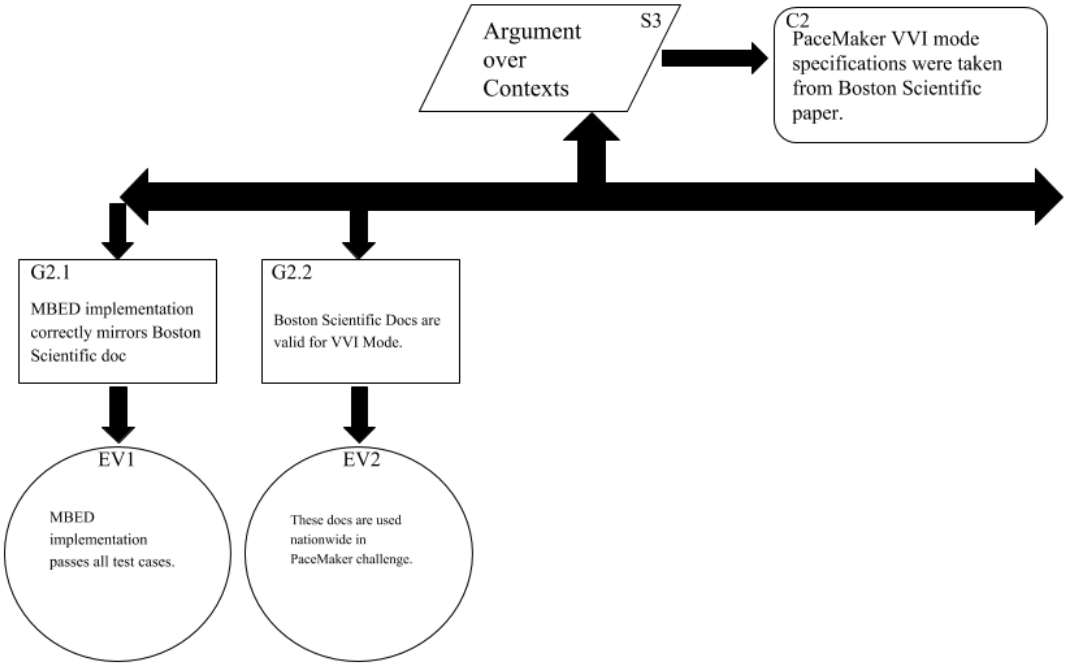

G3
The model satisfies all the desired properties
S3
Testing over desired properties of pacemaker
G6
The model is deadlock free
G7
The model satisfies the LRI property
G8
The model satisfies the HRI property
G9
The model satisfies the VRP property
G10
The model satisfies the rate interval time
The property for deadlock freeness is A[]
The property for LRI is A[] (!PaceMAker.prevHyster imply paceMaker.x <= PaceMaker.LRI
The property for HRI is A[] (PaceMaker.prevHyster imply paceMaker.x <= PaceMaker.HRI)
The property for VRP is A[] (PaceMaker.operating imply PaceMaker.x <= PaceMaker.VRP)
The property for rate interval time is A[](paceMaker.hyster imply PaceMaker.x <= PaceMaker.RateInterval
Ev1
Model checking result of deadlock freeness
Ev2
Model checking result of LRI
Ev3
Model checking result of HRI property
Ev4
Model checking result of VRP
Ev5
Model checking result of rate interval time

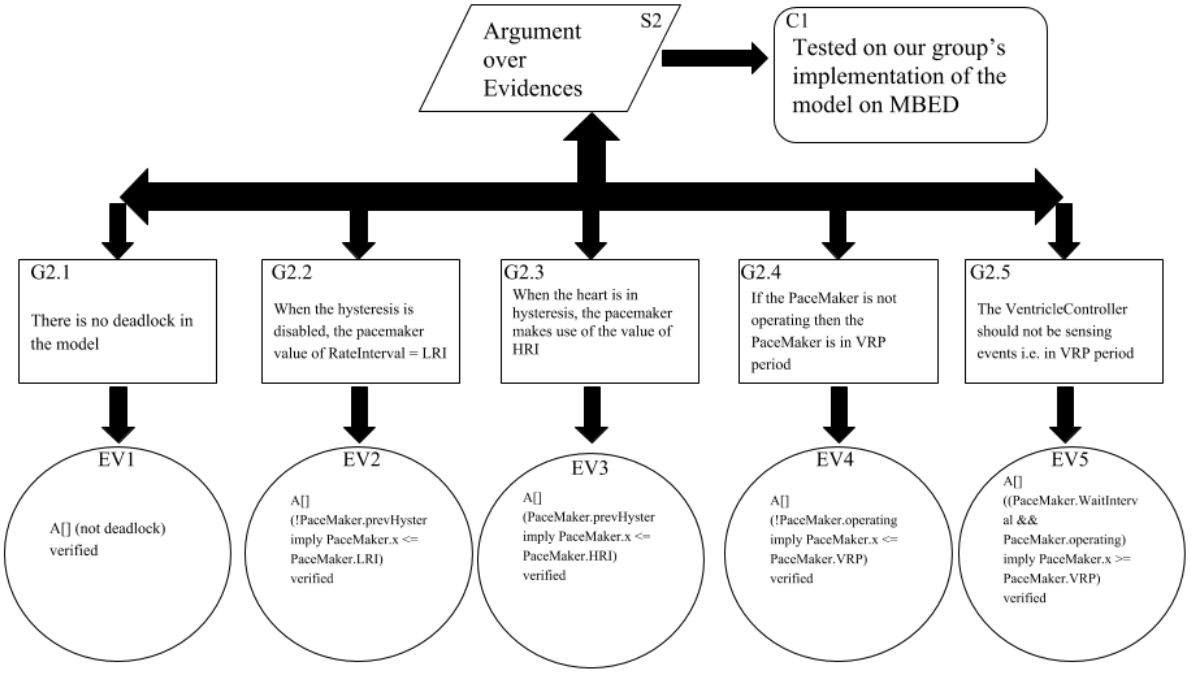

G4
The code generation process generates the code correctly in compliance with the UPPAAL model
C10
EFSM is an effective and reliable tool for code generation
S4
Argument by using reliable tool set
S5
Argument by harmless manual modification
C11
Harmless manual modification does not change functionality of the code
G10
The code generated using EFSM is correct
G11
The manual modifications are confined to API and platform requirements
Ev5
Correctness argument of the EFSM framework
Ev6
The code appended to the project report



Confidence Case:




Argument over Contexts
S3
C2
PaceMaker VVI mode specifications were taken from Boston Scientific paper.
G2.1
MBED implementation correctly mirrors Boston Scientific doc
G2.2
Boston Scientific Docs are valid for VVI Mode.
EV1
MBED implementation passes all test cases.
EV2
These docs are used nationwide in PaceMaker challenge.



Member Contributions

Kwame Owusu: The heart UPPAAL model generation along with assurance cases. Code generation for the pacemaker model and the heart monitor thread.

Henry Garant: The heart UPPAAL model generation along with assurance cases. Development of the cloud application/MQTT, Frontend, and code generation for the pacemaker model.

Muhammad Fahad Asghar: The Pacemaker UPPAAL model generation along with assurance cases. Development of the heart model and the heart monitor thread.

Max Brown: The Pacemaker UPPAAL model generation along with assurance cases. Code generation for the heart model and implementation of MQTT.