Pacemaker

UPenn CIS-541 (Fall 2014) Project

Ariel Eizenberg

arieleiz@seas.upenn.edu

ariel.eizenberg@gmail.com

2014-12-17	Version 1.0	Final version
2014-12-14	Version 0.9 – Touchup	Ready for demo
2014-12-12	Version 0.3 – Assurance Case	Phase III
2014-12-03	Version 0.2 – Implementation	Phase II
2014-11-15	Version 0.1 – Modelling	Phase I

Contents

1. Introduction

1.1. Goals

The goal of this project is to implement a verified DDD pacemaker, on an embedded platform (MBED NXP LPC1768), in three stages:

1. Phase I – Modelling and verification using the UPPAAL system[1]
2. Phase II – Implementation and validation
3. Phase III – Assurance case

1.2. Project Highlights

The following features were modelled, implemented and tested as part of the project:

- **Pacemaker** - A DDD-R pacemaker using an **atrial-based lower rate timing (A-A interval)** modelled in UPPAAL, and a heart model used for correctness testing. The pacemaker includes several extra features not required by the project specification:
    a. PVAB timing cycle,
    b. PVC support,
    c. Dynamic AVI extension,
    d. Alarm buzzer,
    e. GUI Heart rate visualizer (over serial).

  The pacemaker was developed as an timed automata model in the UPPAAL language and verified for correctness using UPPAAL.
- **Code Generation** –A versatile UPPAAL to MBED C++ automatic code generator with direct support for binding UPPAAL channels to hardware. The generator accepts UPPAAL XML files directly, and supports most of the UPPAAL language. The generator was verified against several models and then used to generate code for the pacemaker and heart. The auto-generated code was used with no changes, as-is, for the final implementation of both pacemaker and heart.
- **Simulator** – A CMSIS OS[2] simulator for Windows was developed for rapid development and testing on a PC before deployment to the embedded environment. **The simulator does not require any code changes in the MBED program**, just compiling it using a PC compiler and linking with a stub library. This allows for comprehensive testing using tools available for PC and not on embedded platforms (builtin fuzzing mechanism, versatile automatic checking, code analysis tools, code coverage tools, etc'). The simulator is designed to be extensible by means of a plugin architecture, supports multiple MBED programs running concurrently, simulates multiple hardware devices/sensors, and supports interconnects between the simulated MBED platforms.
- **Runtime verification and visualization tool** – The pacemaker and heart programs output (through the serial port) trace logs of signals, senses and paces. A GUI tool was developed to record these

[1] http://www.uppaal.com/
[2] The operating system used by the MBED platform

traces, visualize them as graphs and perform verification on them. In addition, runtime verification was implemented by generating a version of the pacemaker with special *observer* templates that check for different runtime properties such as timings.

The project did not utilize any external resources except books focused on domain specific knowledge (pacemaker, heart). The project utilizes one single extra library for the embedded implementation to support the TextLCD, and a complementary TextLCD simulator for the PC MBED simulator. The code generation mechanism and simulation environment were written from scratch with no external references.

The initial implementation combined both pacemaker and heart implementations in a single file which detected the correct application by inspecting the hardware, but due to RAM restrictions was split into two separate binaries.

1.3. Terminology

(Adapted from Wikipedia and "The Nuts and Bolts of Cardiac Pacing").

Heart – The human heart is a four-chambered pump that circulates blood through a complex network of blood vessels. The heart has two upper chambers (atria) and two lower chambers (Ventricles), each on the right and left sides. The right side pumps oxygen depleted blood to the lungs, while the left side receives oxygenated blood from the lungs and pumps it through the body.

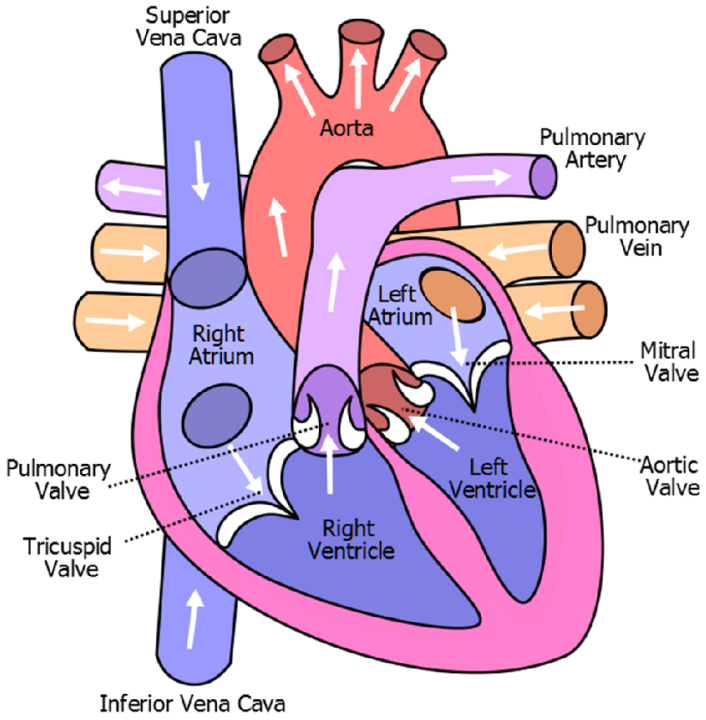


Figure 1 (Source - Wikipedia[3])

[3] http://en.wikipedia.org/wiki/File:Diagram_of_the_human_heart_(cropped).svg

Atria - The **atrium**, plural: **atria,** is one of the two blood collection chambers of the heart. The atrium receives blood as it returns to the heart to complete a circulating cycle.

Ventricle - a **ventricle** is one of two large chambers that collect and expel blood received from an atrium towards the peripheral beds within the body and lungs.

Heart operation – when the heart beats, it contracts due to changes at the cellular level called "depolarizations". The heart beat has two stages – an atrial depolarization followed by a ventricular depolarization. These depolarizations can be seen on an electorcardiogram, as depicted in Figure 2, with the P part indicating atrial depolarization, and the QRS complex representing the ventricular depolarization. The delay between the atrial depolarization and the ventricle depolarization (A-V delay) is achieved by the tissue conducting the impulse from the atria to the ventricles.

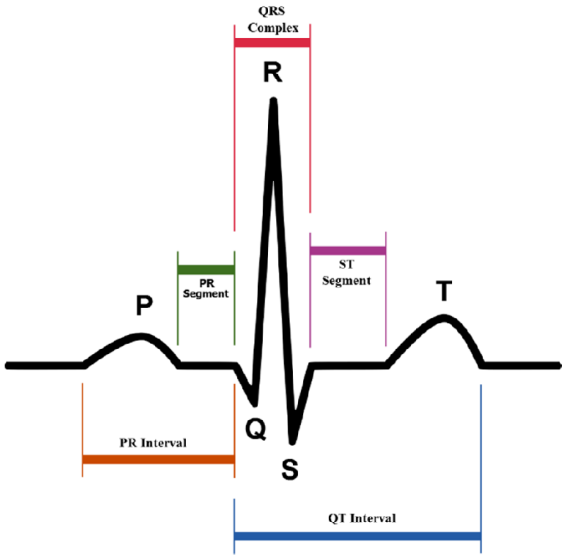


Figure 2 (Source – Wikipedia[4])

Heart failure – Heart failure is the gradual decline in the ability of the heart muscle to pump efficiently, or a conduction disorder that occurs when the heart's electrical system does not work properly.

Bradycardia – A heart condition where the heart rate is too low (a heart rate below 50 bpm in adults).

Pacemaker - A **pacemaker** is a medical device used to address conduction disorders of the heart. The pacemaker uses electrical impulses, delivered by electrodes contracting the heart muscles, to regulate the beating of the heart.

DDD mode – There are many different kinds of pacemaker operating modes. The modes are denotes using a three letter NBG[5] code, the first denoting the chamber(s) paced, the second denotes which chamber(s) are sensed, and the third denotes the mode(s) of response. DDD stands for Dual Paced, Dual Sensed, Dual Mode (Inhibited and Triggered). DDD-R is DDD with an additional feature of rate modulation to better match the heart's natural behavior.

---

[4] http://commons.wikimedia.org/wiki/File:SinusRhythmLabels.svg
[5] http://www.ncbi.nlm.nih.gov/pubmed/11916002

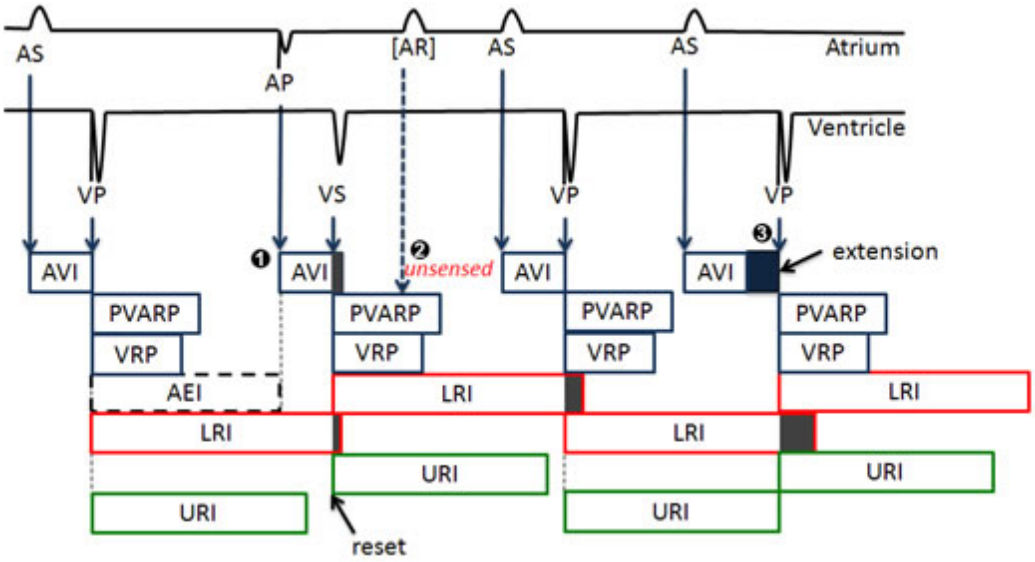


Figure 3 – DDD pacemaker timing cycles[6]

AV interval – Atrio-Ventricular Interval – the time the pacemaker waits for a ventricular event after an atrial event. If no atrial event occurs during this time, the pacemaker will generate a ventricular event. In DDD-R mode the AV interval can be adjust dynamically to try to detect the hearts natural A-V interval.

LRI – Lowest Rate Interval – The basic timing cycle for a pacemaker. The purpose of a pacemaker is to verify the actual heart rate never gets below the LRI.

URI – Upper Rate Interval – Upper bound for pacing – makes sure the pacemaker does not pace at a too high rate, and will not respond to atrial events occurring before URI.

VRP – Ventricular Refractory Period is the period of time the pacemaker ignores all signals after a ventricular pace.

PVARP – Post-Ventricular Atrial Refractory Period. A ventricular pace can cause an electric signal to be detected by the pacemaker's atrial sensor and be erroneously detected as an atrial event. To prevent this from interfering with the pacemaker's timing, the pacemaker blocks all atrial input for this length of time immediately after the generation of a ventricular pacing event.

PAVB – Post-Atrial Ventricular Blanking. In some crosstalk disorders the atrial event can be erroneously sensed as a ventricular event, and prevent correct AV interval wait. The PAVB period of time immediately after an atrial pace that the pacemaker does not sense ventricular events (this of course has to be considerably less than the AV interval).

[6] Figure taken from "Model-Based Closed-Loop Testing of Implantable Pacemakers", Proceeding ICCPS '11 Proceedings of the 2011 IEEE/ACM Second International Conference on Cyber-Physical Systems Pages 131-140.
https://alliance.seas.upenn.edu/~mlabweb/dynamic/wp-content/uploads/2012/02/pacemaker.jpg

2. Overview

As stated in the introduction, the project implemented a software DDD-R pacemaker on the MBED NXP LPC1768 platform.

While the project specification called for using ventricular (V-V) timing for the pacemaker, I chose to implement the pacemaker with an atrial-based LRT (an A-A interval), rather than a ventricular (V-V interval) pacemaker.

The A-A mode is specified on pages 160 - 163 of "Cardiac Pacemakers and Resynchronization Step-By-Step", 2nd Edition by *Barold, et al*. My choice is based on a section in the excellent book "The Nuts and Bolts of Cardiac Pacing", 2nd Edition, by *Tom Kenny*, in Chapter 10 (page 86):

> **Ventricular versus atrial timing**
>
> In the early days of dual-chamber pacing, all devices used what is commonly called ventricular-based timing. …. . In the mid-1980s, pacemaker manufacturers decided to fix this timing quirk by changing to what is now called atrial-bases timing. In atrial-based timing, it is an atrial event (senses or paced) that starts the interval knows as the AA interval."

This means my pacemaker model's LRI implies the longest timespan between two atrial events (A-A), rather than two ventricular events (V-V).

The use of A-A timing implies support for a special behavior, premature ventricular contractions (PVCs). In the presence of a PVC (extra ventricular event sensed before the next atrial event is sensed or paced), the pacemaker switches to an V-A timing mode, e.g. it waits for an *LRI – AVI* timespan before then next atrial pace (*Barold* page 161, *Kenny* page 86).

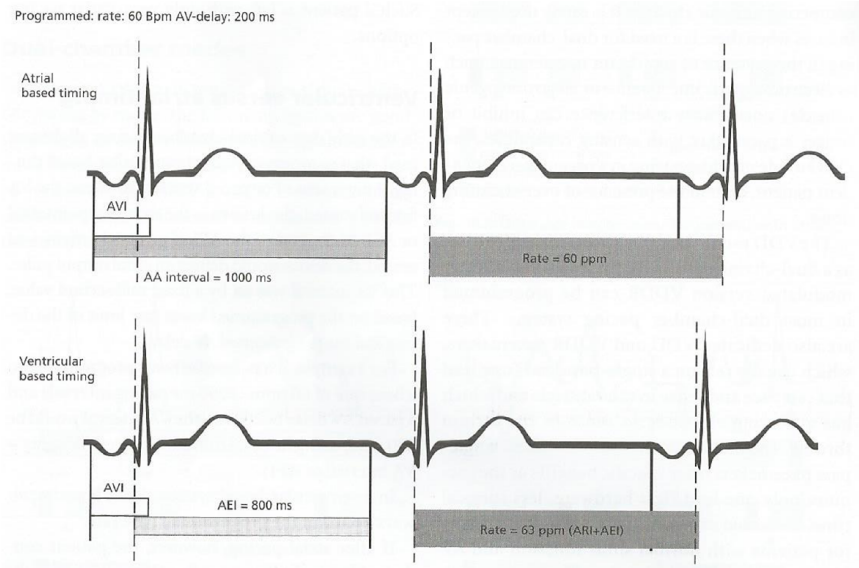


Fig. 10.12 Atrial-based timing compared to ventricular-based timing in a pacemaker set to a base rate of 60 ppm (1000 ms) with an AV delay of 200 ms. Ventricular-based systems could sometimes pace faster than the base rate, but atrial-based systems do not.

Figure 4[7]

[7] Reproduced from the book The Nuts and Bolts of Cardiac Pacing", 2nd Edition, by *Tom Kenny*, in Chapter 10 (page 86)

3. Modeling and Verification

3.1. General

UPPAAL is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata.

The pacemaker and heart were modelled using UPPAAL version 4.0.14, using several templates for each and additional templates for verification. Both pacemaker and heart templates have input and processing split into different templates as per the project requirements.

3.2. Pacemaker

The pacemaker is built from three core templates and six utility templates. All pacemaker templates are prefixed with PM_.

3.2.1. Sensing

Heart sensing is performed using two templates, PM_SensorA for atrial sensing and PM_SensorV for ventricular sensing. These state machines process incoming heart A and V signals, respectively, and mask or forward them based on the different timing cycles of DDDR.

Atrial sensing is performed using the following EFSM:

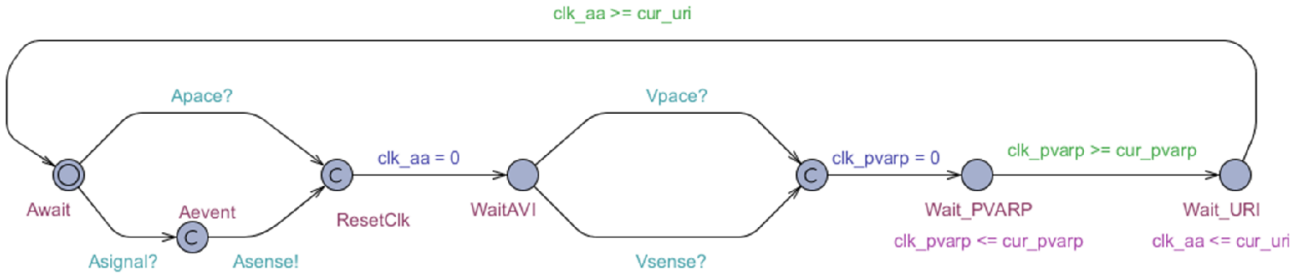


Figure 5

It is built from four major stages:

1. Once an atrial event occurs (Apace? or Asense!), the LRI A-A clock is reset, and the model starts ignores all atrial events – the AVI, although the AVI cycle length itself is controlled from the pacing template, described later in this document,
2. The model waits the next ventricular event (sensed or paced), which signals the end of the AVI, and the template starts waiting a PVARP using a dedicated clock.
3. Once the PVARP clock expires, the template waits for the URI cycle to complete (if URI ≥ AVI + PVARP).

4. The main atrial stage - At this point the EFSM is ready restart the cycle, waiting for an atrial event (signaled by the heart or paced). If a heart event is detected, the model signals the pacer using the Asense signal.

The second template, PM_SensorV, is used for ventricular sensing:

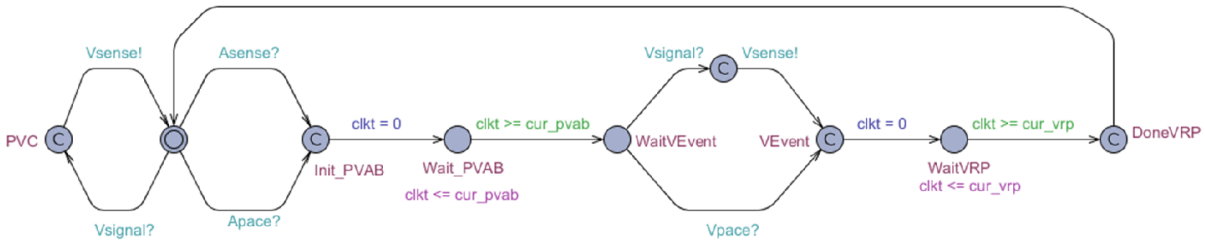


Figure 6

This template is built from four main stages, plus support for the PVC condition:

1. Once an atrial event occurs (Apace? or Asense!), the model resets a private clock used to time the PVAB cycle (during which ventricular events are blocked).
2. The main ventricular stage - once the PVAB cycle is complete, the model begins waiting for the AVI period to end by waiting for a heart Vsignal or the pacer Vpace. If a heart event is detected, the model signals the pacer using the Vsense signal.
3. The VRP cycle, blocking ventricular events, starts immediately after the ventricular event happens, by resetting the private clock and waiting the VRP period to timeout.
4. At this time the loop resets, waiting again for an atrial event.
5. During this wait, a extra ventricular event (PVC) requires extra processing by the pacer model, and thus ventricular events sensed in this period (Vsignals) are translated into Vsense events.

3.2.2. Pacing

Pacing is implemented using the PM_Pacer_AA template:

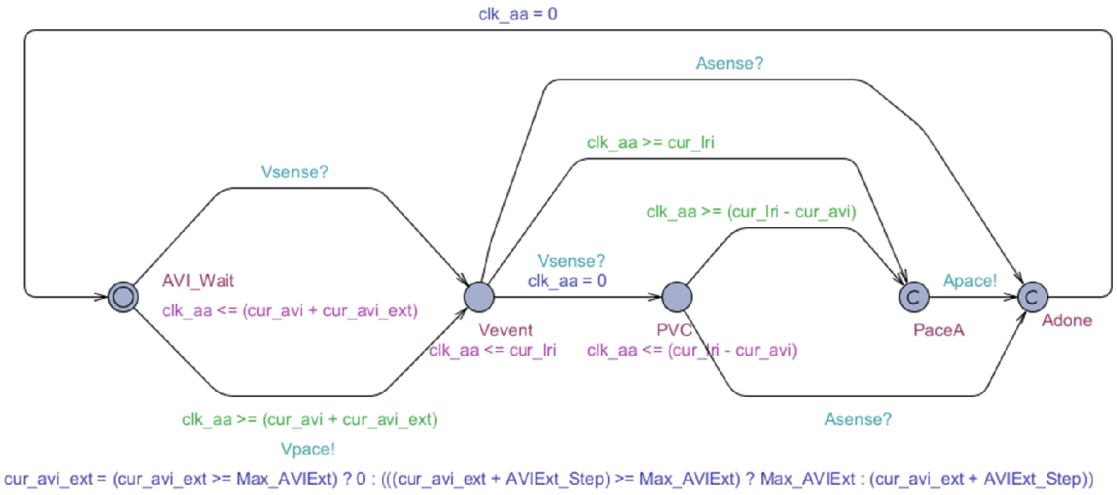


Figure 7

The pacing loop is built from two top-level steps (for clarity, the diagram above does not contain the mode-switch manual mode, although it's included in the UPPAAL model):

1. Once the A-A clock restarts (following any atrial event), the AVI period begins. The pacer template implements support for a dynamic AVI period. In a dynamic AVI period, the AVI set in the pacemaker is lower than usual, and the pacemaker tries to adjust it dynamically to the native AV node delay by extending it slowly until heart ventricular events are sensed within the AVI. If the AVI reaches the maximal AV delay, without any sensed heart ventricular event, the AVI is reset to its lowest value. This algorithm is described in *Barold* page 393.
2. Once the AVI period is complete, the model supports three behaviors:
    a. An atrial event is sensed (e.g. it is signaled by the heart and the PM_Sensor_AA template does not block it due to PVARP or URI, rather translating it into an Asense!). In this case the loop restarts.
    b. LRI detection – if no heart atrial event is sensed within the LRI timing cycle, the pacer generates an atrial pace (Apace!), and then restarts the loop.
    c. If an extra ventricular event is sensed (PVC), the pacemaker reverts to V-A timing, waiting (LRI – AVI). If at any time during this V-A wait the pacemaker senses an atrial event it restarts the loop. If the V-A timing expires, an atrial pace is generated.

3.2.3. Mode-switch

The mode-switch is implemented using the PM_ModeSwitch template:

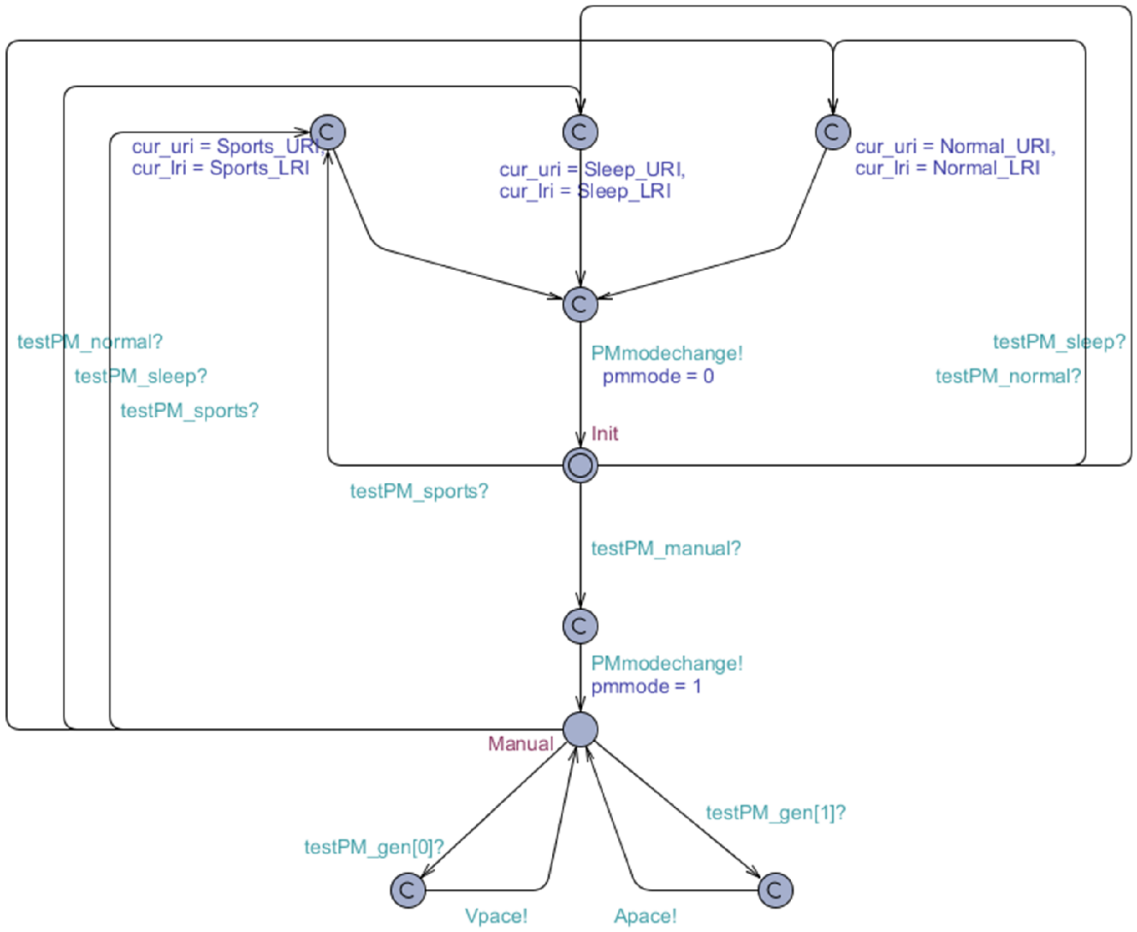


Figure 8

The mode-switch is responsible for supporting the pacemaker's four operating modes, Normal, Sleeping, Sports and Manual.

The model receives four possible events from a testing template four setting the mode - testPM_sleep, testPM_normal, testPM_sports, testPM_manual. If any of the first three is detected, the LRI and URI are changed accordingly. If a manual mode is detected, the modeswitch begins waiting for the manual A and V signals, testPM_gen[0] and testPM_gen[1] respectively. If during manual mode any of the other three settings is detected, the model exits manual mode and switches to the desired mode.

3.2.4. Display and Alarms

The pacemaker supports functionality for displaying the current heart rate and generating audio alarms for overly high are low rates.

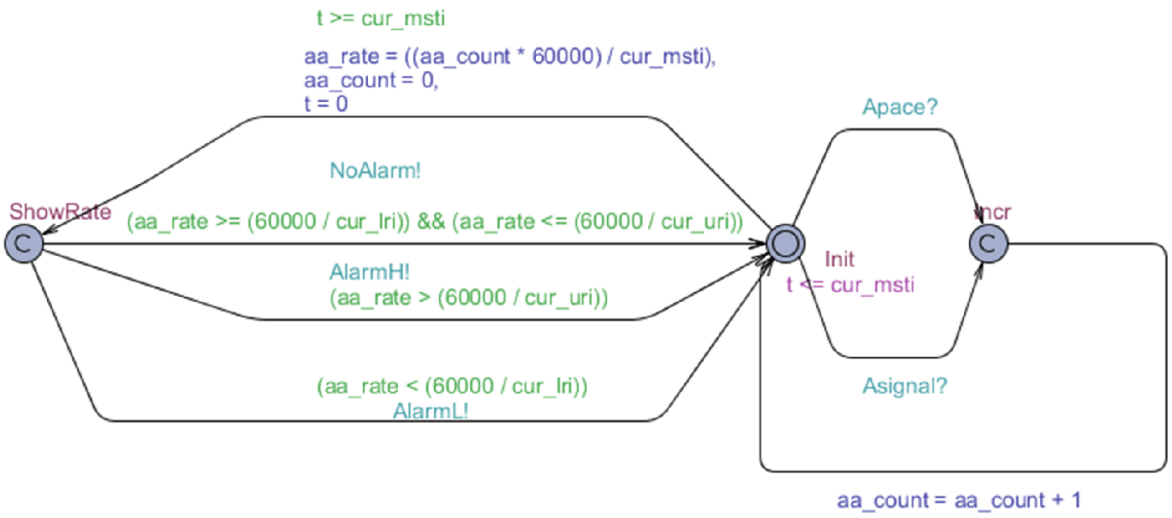


Figure 9

The EFSM counts the clock ticks in a “Mode Switch Time Interval” (MSTI), and calculates the rate from the accumulated amount. Once the rate is calculated it is checked for alarm values and the appropriate signal is generated.

3.2.5. LEDs

The pacemaker supports four LEDs for describing atrial sense, atrial pace, ventricular sense and ventricular pace. The sense LEDs light up only when the event is sensed by the sensing template, and ignored when the signaled events are blocked.

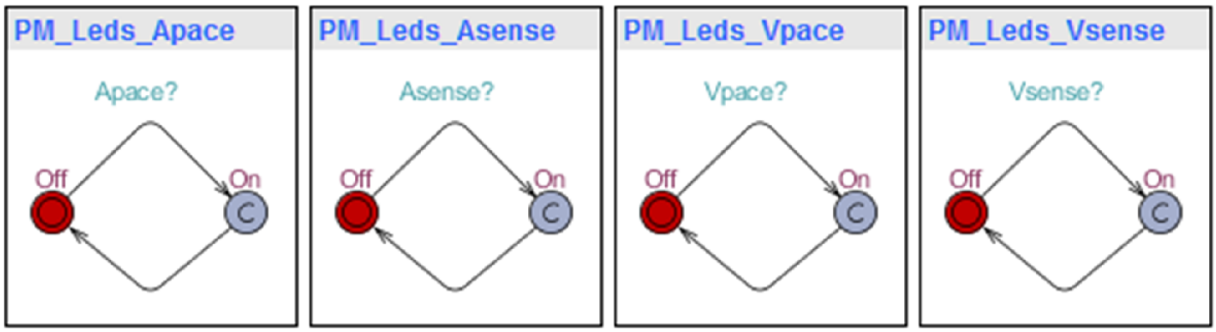


Figure 10

3.3. Heart

The heart model is implemented using four main templates and five utility templates. All heart templates are prefixed with HM_.

3.3.1. Signaling

Normal heart behavior, signaling, is implemented using two templates:

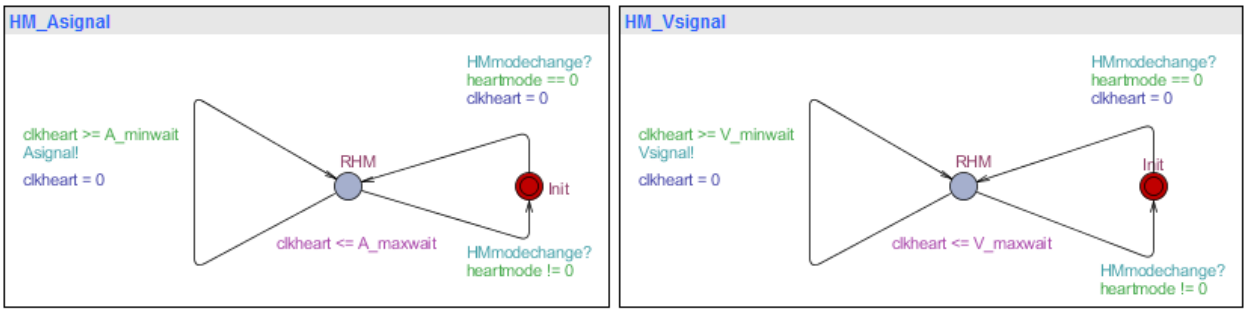


Figure 11

Once *heartmode* is set to RHM, signified by the value zero, the models start generating Asignal And Vsignal events in at least a A/V_minwait and at-least A/V_maxwait. Both templates share the same clock as required by the project specification (I would have rather implemented them using separate clocks).

If a mode change to a mode other than RHM is detected, the template reverts to the Init state.

3.3.2. Pace response

The heart in RHM mode responds to pacing events by resetting the heart clock, again using two templates, one for atrial pacing and one for ventricular pacing:

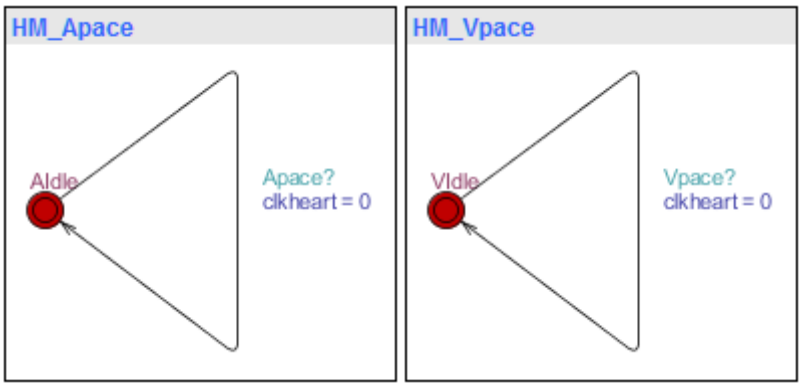


Figure 12

(These could have been merged into a single template at no cost.)

3.3.3. Mode-switch

The heart mode switch supports three operating modes: a random heart model (heartmode = 0), testing mode which generates several predefined testing scenarios (heartmode = 1), and manual mode (heartmode = 2). The mode switch is very similar to the pacemaker mode switch, so it is not explained in detail (figure on next page).

3.3.4. LEDs

The heart model uses four LEDs to display the heart state – HM_Apace, HM_Asignal, HM_Vpace, HM_Vsignal. The Apace and Vpace are exactly the same as the pacemaker's similarly named LEDs, but the signal LEDs show the heart generated events and not the more limited pacemaker sensed events (figures on next page).

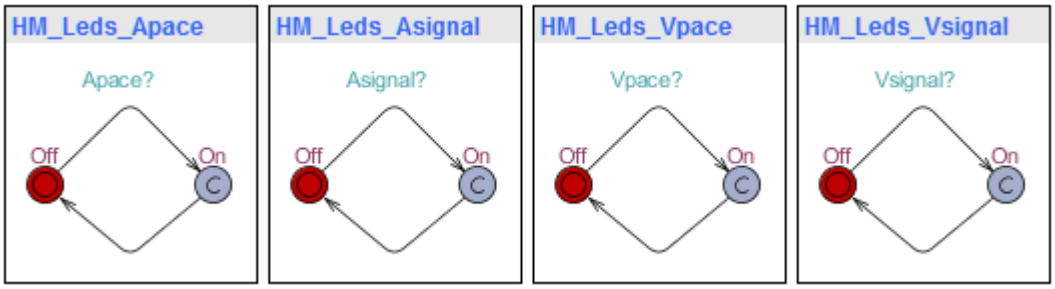


Figure 13

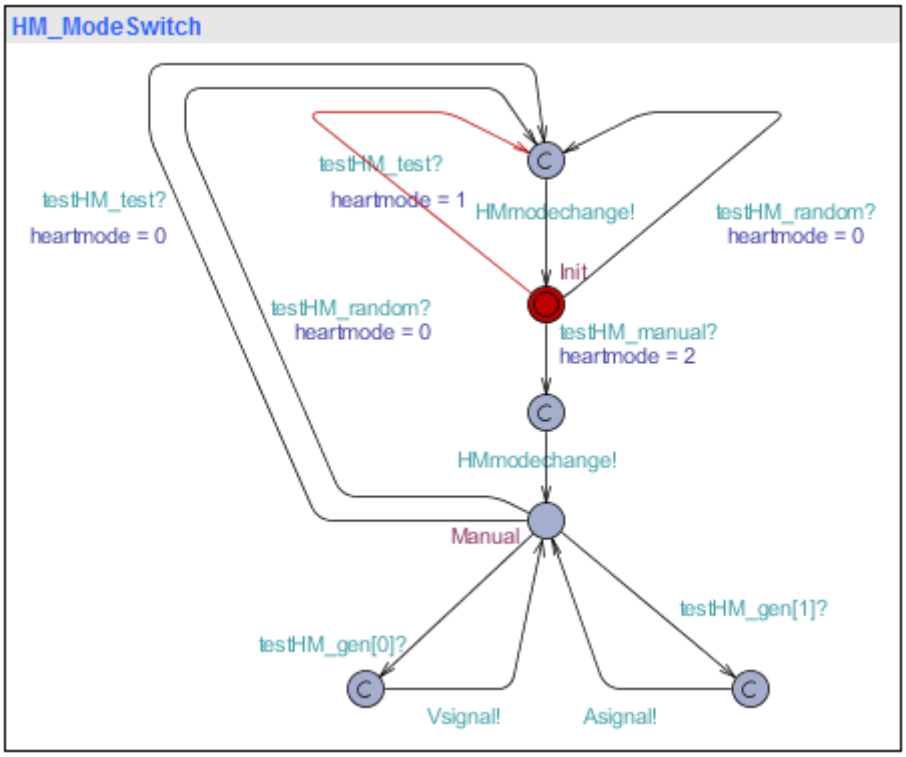


Figure 14

3.4. Testing templates

3.4.1. The Observer templates

These template exists to help with testing, by providing events and sinks suitable for querying the required validation properties in UPPAAL.

3.4.2. Testing templates

These are a multitude of templates, all prefixed with Tester_, and used to test different scenarios. These templates are not always instantiated, and are activated by uncommenting them in the systems declaration block.

3.5. Verification

The model was verified using UPPAAL's querying capabilities.

3.5.1. Deadlock

The system was verified successfully against deadlock in several testing configurations using

```
A[] !deadlock
```

3.5.2. Timing Cycles

- LRI (the LRI observer checks the maximal time between A-A events):

```
A[] Observer_LRI.S3 imply Observer_LRI.t <= cur_lri
```

- URI (the LRI observer checks the minimal time between A-A events):

```
A[] Observer_URI.S3 imply Observer_URI.t >= cur_uri
```

- AVI + PVARP:

```
A[] Observer_URI.S3 imply Observer_URI.t >= (cur_avi + cur_avi_ext + cur_pvarp)
```

- VRP (the VRP observes checks the minimal time between V-V events).

```
A[] Observer_VRP.S3 imply Observer_VRP.t >= cur_vrp
```

- PVAB (the AV observer checks the minimal time between V-A, A-V events).

```
A[] Observer_AV.S5 imply Observer_AV.t >= cur_pvab
```

- PVC

```
A[] Observer_AV.S7 imply Observer_AV.t >= (cur_lri - cur_avi)
```

- The pacemaker will never generate two ventricular paces without an atrial event in-between:

```
A[] not Observer_AV.BAD1
```

- The pacemaker will never generate two atrial paces without an ventricular event in-between:

```
A[] not Observer_AV.BAD2
```

3.5.3. LEDs

These safety and liveness of these components was verified using queries of the following form:

- Pacemaker A pace implies A led is on:

```
A<> PM_Pacer_AA.PaceA imply PM_Leds_Apace.On
```

- Pacemaker V pace implies V led is on:

```
A<> PM_Pacer_AA.PaceV imply PM_Leds_Vpace.On
```

And so on.

3.5.4. Heart Liveness

- Signaling:

```
A<> Observer_PS.Asignalled

A<> Observer_PS.Vsignalled
```

4. Implementation and Validation

When considering the implementation of the pacemaker, two options come to mind:

- Hand code UPPAAL model into code,
- Automatically generate code from the model

Hand coding is easier then automatically generating code, but can create subtle differences between the model and implementation. Furthermore, changing the model (due to discovered errors) will sometimes require major code changes. On the other hand, UPPAAL "ideal model" semantics are not easily translatable to a real-time embedded platform. In the interest of facing this challenge I chose the latter option, which is discussed in section 4.1.

Development for an embedded platform is slower than development for PC, as it involves a much more limited system, cross-compilers, remote debugging (if any), can be affected by hardware before those are completely ironed out. Embedded platforms mostly do not support the large plethora of software analysis tools available in regular development, such as rich code analysis tools, coverage testing, memory checking and the like. During debugging, limits of embedded hardware bandwidth can substantially limit the ability to generate large trace files to aid in development and testing. To enable rapid prototyping and development of the project code, I developed a "cross-compilation" environment that simulates the MBED's CMSIS operating system on a regular PC. This will be explained in section 0.

The MBED online IDE is very limited, and does not provide tools for debugging and analyzing code. To ease development, the project was developed using the GNU toolchain for ARM and uses PyOCD for debugging, as I detail in section 4.2.1.

The actual hardware developed for this project is described in section 0, while the actual software implementation of the pacemaker and heart is shown in section 0. Finally, testing and validation is detailed in section 0.

4.1. Code Generation Tool

4.1.1. Overview

The code generation tool can be surmised in one sentence – It takes an UPPAAL system as input and generates C++ code for the CMSIS operating system used by the MBED platform. Each process (template) in UPPAAL is mapped to its own thread, and an additional engine thread is responsible for synchronizing these threads.

The dependency on CMSIS is quite minimal, and requires three basic APIs – thread *creation+signaling*, *mutex* locks and a timer. Additional dependencies are required to connect channels to hardware pins – the *gpio_xxx* functions.

To ensure consistent and low resource performance, the engine does not allocate any memory dynamically, with all required memory allocated in advance as data variables. The single exception is the stack space for the different threads which is allocated just once at startup. The engine and the autogenerated code are designed to ensure maximal performance, preserving thread locality as possible

and minimizing data structure traversals and sizes. The code generator also attempts to optimize guards, invariants and updates as much as possible during code generation time.

A key difference between classic UPPAAL semantics and the semantics I chose to implement is in channel signal delivery. In classic UPPAAL, and channel action (*x*!) will not occur if there is no one listening for the channel (*x*?). I chose to implement the code generation tool to generate channel actions even if no code is actively listening.

The runtime system is comprised of two parts: a static engine and code+data generated from the model. These form two C++ files – the static ModelImp.cpp for the engine, and Model.cpp which is autogenerated from the UPPAAL model.

If the reader has not time to read the engine description, I suggest he gloss over section 4.1.10, which gives a concise demonstration of the code generator's power.

4.1.2. Engine Execution Flow

The most important concept in UPPAAL models is the "clock tick". Two modifiers on nodes (location) affect their behavior – urgent nodes do not allow time to pass, and committed nodes require that the next transition, in all processes, be from a committed node. Furthermore, synchronization events that are "signaled" during one tick are not stored for next ticks, and only have effect on other processes whose current state that have transitions synchronized by these channels.

The real world cannot fully support UPPAAL notation of no time progress, but it is still important to maintain urgent and committed semantics, as well as to synchronize the steps between the different threads so they all follow the same clock tick (otherwise assumptions in the model can be broken in the implementation).

The component responsible for maintaining the flow of time is the main engine thread. The main thread runs an infinite loop:

```
forever: {
   run_steps();
}
```

Each step is responsible for a "tick" in time, although it can sometimes take more than a tick. Each tick is made up of three blocks – initialization, normal state progress and urgent state progress (committed states are treated separately as will be detailed later). The run_steps() function does not return until the active state in all process is a regular state (pseudo-code, the actual implementation is more complex):

```
clear_signals(); // reset all synchronizations fired and not "taken"
clear_normal_step(); // clear step markers from previous iteration
run_tasks(Commited);
forever: {
   if(tick_has_passed())
         break;
   run_tasks(Normal | Urgent); // normal state will only
                               // be executed once
   signal_wait(change, time_left_to_end_of_tick());
```

```
}
run_tasks(Urgent);
```

The function first makes sure all committed states are taken care of, and then, time allowing, tries to progress process whose current state is regular. Finally, the function does not return while some processes are still in urgent states. Once all process who could have progressed have done so, the function waits for changes such as processes changing states (and possibly entering urgent or committed states), or channel synchronization events (e.g. if a channel was signaled, we can unfreeze any receivers what have transitions synchronized by this channel). Committed states require special treatment in all stages of run_steps(), not just when explicitly specified, and the run_tasks() function is responsible for that:

```
forever: {
  bool anyfound = false;
  for (all processes p) {
        state s = p->current_state;
        if ((looking_for != Commited) && s->mode == Commited){
              // found commited step in middle of urgent/normal loop
              restart loop with looking_for = Commited;
        }
        if(p->mutex is locked) {
              anyfound = true; // a task is still running
              continue;
        }
        lock p;
        if(p->mode in looking_for) {
              if(p->mode != Normal
                    || (p has not stepped in this tick)) {
              anyfound = true; // found a task to run!
              signal(p, authorization to run a step);
        }
        unlock p;
  }
  if (!anyfound) {
        if (loop was switched to "committed mode") {
              // no more commited nodes, so we can:
              switch back to the original looking_for;
        }
        return; // nothing more to do but wait for events
  }

  // we have active tasks running, wait for them to complete
  signal_wait(changed, forever);
  should_stop = true;
  for (all processes p) {
        if(can lock p) { /* not running */
              if(p is in Commited state) should_stop = false;
              unlock p;
        } else { /* running */
              should_stop = false;
```

```
        }
    }
    if(should_stop)
        return;
}
```

Each task is implemented using a thread, waiting for signals from the main thread and running steps as required and possible. When awakened, the thread will take a single transition at a time, unless the thread enters a Committed state, and then it will continue to try and transition until it leaves the committed state without waiting for instructions by the main thread:

```
Let p be thread's process;
lock();
enter initial state;
unlock();
signal(main thread, p ready);
forever: {
   signal_wait(run tasks, forever);
   lock();
   do {
        transition = find_valid_transition();
        if (found transition) {
            take_transition(transition);
        } else {
            p->stepped = false;
            break;
        }
   } while (p's current state is Commited);
   unlock();
   signal(main thread, p ready);
}
```

Finding a transition is simple enough:

```
let c be p->current_state.
for(all possible transitions t from c) {
   if(can take transition t) {
        return t;
   }
}
return nothing found;
```

Testing if a transition can be taken is simple as well:

```
if(t requires synchronization by "x?") {
   if(x is not signaled) return false;
}
if(t has guard function g) {
```

```
   lock all states variables();
   result = g();
   unlock all states variables();
   if(!result) return false;
}
return true;
```

Taking a transition involves leaving the current state of the process, signaling synchronizations of the form “x!” (if they exists) , and running the variable update function for the transition, followed by entering the new state.
As a side note, the engine does not use the regular MBED rtos mbed::Thread class, as this starts the thread from the constructor and not when required by the engine. Instead, the engine uses the CMSIS thread API directly (`osThreadCreate(), …`).

Warning: the actual code is more complex than the pseudo-code shown above. Care should be taken when changing any of it as not to break UPPAAL compatability.

4.1.3. Code generation

The code generation tool, written in C#, parses the UPPAAL XML file and extracts templates and global declarations. Each template is analyzed to extract locations, edges and other parameters. Once the XML file is loaded it is analyzed to ensure it is valid and consistent. Each element gets a unique name, generated from the UPPAAL name. The extracted data is stored as a tree:

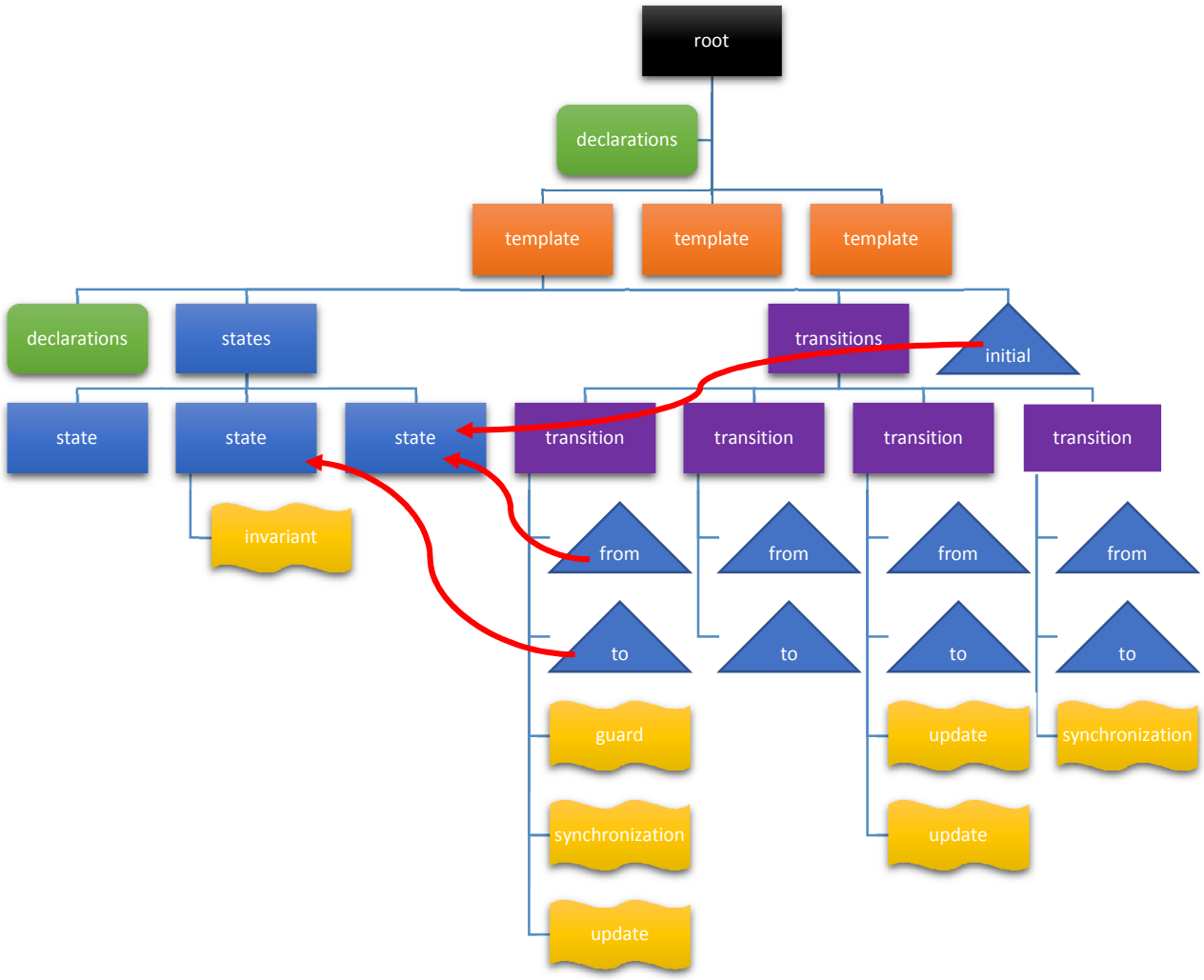


Figure 15

Declarations, invariants, guards, synchronizations and updates are parsed using a BNF parser implemented using the open source Coco/R package. The parser tracks state of type definitions (such as typedefs), as well as scoping, e.g. each template can have its own declarations, but all "see" global declarations.

Once the model and code are completely parsed, the code generator performs an optimize sweep to optimize ("fold") expressions.

Finally data and code is generated for the model. Only invariants, guards and updates generate actual code – all the rest of the model is generated into data definitions used by the engine. All objects (channel, transitions, nodes, templates) are split into several data structures:

- A const structure describing the static state of the object,
- A user visible structure used for registering callbacks on the object,
- An optional dynamic state structure

4.1.4. Channels and Hardware Support

Synchronization channels represent *chan* entities in UPPAAL declarations. Channels are fired (signaled) when a transition has a synchronization rule with '!' at the end (for example "lights_on!"), or are fired by external events. External events can include user code or hardware interrupts. When channels are fired they can trigger hardware events, and can call user callback functions.

Synchronization channels are generated into three structures.

The first is the constant static descriptor:

```
struct SYNCHRONIZATION_CHANNEL_DATA {
        const char* name; // used only in debug
        bool urgent;
        bool broadcast;
        SYNCHRONIZATION_CHANNEL_STATE* state;
};
```

A private dynamic state descriptor:

```
struct SYNCHRONIZATION_CHANNEL_STATE {
  osMutexId mutexId;
  osMutexDef_t mutexDef;
  SendData send[MAX_SEND_DATA];
  RecvData recv;
  bool fired, prev_fired;
};
struct SendData {
  PinName pin;
  SendChannelMode mode;
  int pulse_length_us;
  gpio_t gpio;
  int last_value;
};
struct RecvData {
  RecvChannelMode mode;
  gpio_irq_t gpio_irq;
  gpio_t gpio;
};
```

And finally the user visibile structure:

```
struct SYNCHRONIZATION_CHANNEL {
   const SYNCHRONIZATION_CHANNEL_DATA* data;
   void(*fired)(SYNCHRONIZATION_CHANNEL* channel);
   void* context;
}
```

The code generator generates all of these for all channels appearing in the UPPAAL model. The channel variable name is derived from the template name (or "GLOBAL" for global declarations) and the variable name. For example, a broadcast channel named *Apace* defined globally will generate the following code in Model.cpp:

```
// brodcast chan Apace

SYNCHRONIZATION_CHANNEL_STATE GLOBAL_CHANNEL_VAR_Apace_STATE;
const SYNCHRONIZATION_CHANNEL_DATA GLOBAL_CHANNEL_VAR_Apace_DATA =
{
   DBGSTR("brodcast chan Apace[]"),    // name
   false,                              // urgent
   true,                               // broadcast
   &GLOBAL_CHANNEL_VAR_Apace_STATE,    // state
};
SYNCHRONIZATION_CHANNEL GLOBAL_CHANNEL_VAR_Apace =
{
   &GLOBAL_CHANNEL_VAR_Apace_DATA,          // data
   NULL,                                    // fired_callback
   NULL,                                    // context
};
```

The "`SYNCHRONIZATION_CHANNEL GLOBAL_CHANNEL_VAR_Apace`" declaration is visible to the user through Model.hpp. `DBGSTR` is a macro the defines the string in debug mode but returns NULL in release mode.

Channel input is set by calling the Dispatcher::`set_receive_input()` function:

```
void set_receive_input(SYNCHRONIZATION_CHANNEL* channel,
                       PinName pin,
                       PinMode mode, // pull up, pull down, …
                       ReceiveChannelMode rcmode)
```

ReceiveChannelMode is one of:

- ReceiveChannelModeNone – do nothing,
- ReceiveChannelInterruptRise – fire channel on pin rise ,
- ReceiveChannelInterruptFall – fire channel on pin fall.

A channel can also be fired manually using the Dispatcher::fire_channel() function:

```
void fire_channel(SYNCHRONIZATION_CHANNEL* channel);
```

When a channel is fired, apart from triggering state changes, the user can specify additional actions to occur.

First, the user can specify a generic callback using Dispatcher::set_channel_fire_cb():

```
void set_channel_fire_cb(SYNCHRONIZATION_CHANNEL* channel,
              void (*func)(SYNCHRONIZATION_CHANNEL* chan),
              void *context);
```

Only a single callback function can be specified this way.

Channel firing can also be configured to directly invoke hardware output using Dispatcher::set_channel_acount():

```
void set_channel_action(SYNCHRONIZATION_CHANNEL* channel,
                    PinName pin,
                    SendChannelMode mode,
                    int pulse_length_us = 0);
```

SendChannelMode is one of:

- SendChannelModeNone – do nothing,
- SendChannelModeSet – set the pin to 1,
- SendChannelModeToggle – toggle the value of the pin,
- SendChannelModeReset – set the pin to 0,
- SendChannelModePulseUp – pulse up the pin to 1 for the time specified by pulse_length_us (in microseconds), then resets to 0,
- SendChannelModePulseDown – pulse down the pin to 0 for the time specified by pulse_length_us (in microseconds), then set back to 1.

A channel can trigger several hardware events (limited by a #define, currently set to two), and can be specified independently of a callback function.

Channels can be defined as arrays. Consider the following example of two channels used to denote the ‘A’ and ‘V’ key presses of a pacemaker (this syntax allows for random transition channel selections):

```
typedef int[0,1] tstkeyAV;
tstkeyAV k;
urgent broadcast chan testHM_gen[tstkeyAV];
```

The code generator automatically generates the following structures:

```
SYNCHRONIZATION_CHANNEL_STATE GLOBAL_CHANNEL_VAR_testHM_gen_0_STATE;
const SYNCHRONIZATION_CHANNEL_DATA GLOBAL_CHANNEL_VAR_testHM_gen_0_DATA =
{ … }
SYNCHRONIZATION_CHANNEL GLOBAL_CHANNEL_VAR_testHM_gen_0 =
{ … }

SYNCHRONIZATION_CHANNEL_STATE GLOBAL_CHANNEL_VAR_testHM_gen_1_STATE;
const SYNCHRONIZATION_CHANNEL_DATA GLOBAL_CHANNEL_VAR_testHM_gen_1_DATA =
{ … }
SYNCHRONIZATION_CHANNEL GLOBAL_CHANNEL_VAR_testHM_gen_1 =
{ … }

SYNCHRONIZATION_CHANNEL* const GLOBAL_CHANNEL_VAR_testHM_gen_ARRAY[] =
{
        &GLOBAL_CHANNEL_VAR_testHM_gen_0,
        &GLOBAL_CHANNEL_VAR_testHM_gen_1,
        NULL
};
```

If, for example, a transition is synchronized by `testHM_gen[k]`, the engine automatically will access `GLOBAL_CHANNEL_VAR_testHM_gen_ARRAY[(int)SystemVariables.int_GLOBAL_k]` to use the correct appropriate synchronization channel.

The code generator also creates an array of all system channels. This list is used by the engine to initialize and clear channels at the beginning of each step:

```
SYNCHRONIZATION_CHANNEL* ALL_CHANNELS[] = {
   &GLOBAL_CHANNEL_VAR_...,
   &GLOBAL_CHANNEL_VAR_...,
   …,
   &TEMPLATE_A_CHANNEL_VA_...,
   …,
   NULL
};
```

4.1.5. Transitions

The code generator generates three structures for all edges.

A static description:

```
struct TRANSITION_ENTRY_DATA {
        const char* name; // debug only
        STATE_ENTRY* target; // target state
        SYNCHRONIZATION_CHANNEL *receive;   // x? synchronization
        SYNCHRONIZATION_CHANNEL *send;      // x! syncrhonization
        func_guard_t      guard;
        func_update_t     update;

};
```

And a dynamic user visible structure:

```
struct TRANSITION_ENTRY {
   const TRANSITION_ENTRY_DATA* data;
   void(*transition_pre_execute)(TRANSITION_ENTRY* me);
   void(*transition_post_execute)(TRANSITION_ENTRY* me);
   void* context;
};
```

Transitions (edges) names are generated using the following formula:

```
[TEMPLATE NAME]_[SOURCE STATE NAME]_[TARGET STATE NAME]_[Name or ID]
```

Since UPPAAL does not provide means to label edges, the code generator sequentially numbers all transitions between two given states. This can be cumbersome if callbacks are required for the transition, as the numbering may change with model edits. To alleviate this issue, the code generator scans the edge

comments and looks for a pattern “{name=MYNAME}”, and sets the transitions name to be MYNAME instead of the sequential ID.

Callbacks on transitions can be set using the Dispatcher::set_state_pre_execute_cb() and Dispatcher::set_state_pre_execute_cb() functions.

Guard functions are generated automatically based on guard conditions specified in the edge’s declaration. For example, the edge

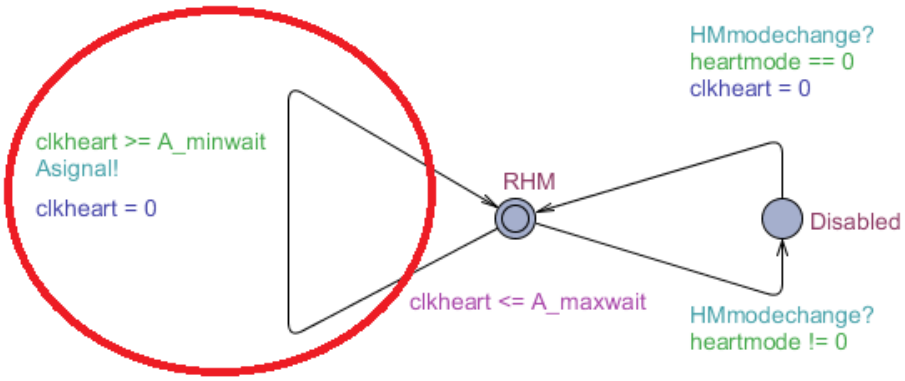


Figure 16

will generate the following code:

```
static bool guardfunc_HM_Asignal_TRANS_RHM__RHM__0(){
    return ((((int)SystemVariables.clk_GLOBAL_clkheart) >=
                    ((int)SystemVariables.int_GLOBAL_A_minwait)));
};
static void updatefunc_HM_Asignal_TRANS_RHM__RHM__0() {
    SystemVariables.clk_GLOBAL_clkheart = (0);
};
const TRANSITION_ENTRY_DATA HM_Asignal_TRANS_RHM__RHM__0_DATA = {
        DBGSTR("RHM__RHM__0"),   // name
        &HM_Asignal_STATE_RHM,              // target
        NULL,                               // receive
        &GLOBAL_CHANNEL_VAR_Asignal,        // send
        guardfunc_HM_Asignal_TRANS_RHM__RHM__0,    // guard
        updatefunc_HM_Asignal_TRANS_RHM__RHM__0,   // update
};
TRANSITION_ENTRY HM_Asignal_TRANS_RHM__RHM__0 = {
        &HM_Asignal_TRANS_RHM__RHM__0_DATA,         // data
        NULL,               // transition_pre_execute
        NULL,               // transition_post_execute
        NULL,               // context
};
```

4.1.6. Nodes

State nodes are similarly defined using two structures:

```
struct STATE_ENTRY_DATA {
        const char* name; // debug mode only
        TRANSITION_ENTRY* const* transitions;
        StateMode mode;
        func_guard_t guard;     // state invariant
};
struct STATE_ENTRY {
        const STATE_ENTRY_DATA* data;
        void(*state_enter)(STATE_ENTRY* me);
        void(*state_leave)(STATE_ENTRY* me);
        void* context;
};
```

The transition list *transitions* contains only transitions coming out of this specific state node, for example for a node *Manual* connecting to three unnamed nodes, with two separate transitions to the node *Unnamed1*:

```
const TRANSITION_ENTRY* HM_ModeSwitch_STATE_Manual_OUT[] =
{
   &HM_ModeSwitch_TRANS_Manual__Unnamed2__0,
   &HM_ModeSwitch_TRANS_Manual__Unnamed3__0,
   &HM_ModeSwitch_TRANS_Manual__Unnamed1__0,
   &HM_ModeSwitch_TRANS_Manual__Unnamed1__1,
   NULL
};
```

Callbacks on states can be specified using the Dispatcher::set_state_enter_cb() and Dispatcher::set_state_leave_cb() functions.

4.1.7. Processes

Process are defined using four different structures.

The user visible callback structure:

```
struct PROCESS_ENTRY {
        const PROCESS_ENTRY_DATA* data;
        void(*process_start)(PROCESS_ENTRY* me);
        void* context;
};
```

The static const process data:

```
struct PROCESS_ENTRY_DATA {
        const char* name; // debug only
        STATE_ENTRY* init_state;
```

```
        PROCESS_ENTRY_THREAD* threadptr;
};
```

`PROCESS_ENTRY_THREAD` is a small class wrapping the process's thread and mutex.

And finally the process dynamic state:

```
struct PROCESS_ENTRY_STATE {
        PROCESS_ENTRY* process;
        PROCESS_ENTRY_THREAD* thread;
        const PROCESS_ENTRY_DATA* data;
        STATE_ENTRY* current; // current state
        bool stepped;         // state has changed in current run
        bool normal_stepped;  // state was normal in current run
        ProcessState state;   // running, stopped, etc'
};
```

Process start callbacks can be set using the Dispatcher::set_process_start_cb() function.

The code generator creates an empty placeholder for all processes in the current system. The user can decide which processes to start by adding them to the dispatcher's list using the Dispatcher::add_process() function.

4.1.8. Variables

The engine supports two types of variables: *int*s and *clock*s. Bools are treated as regular *int*s, with non-zero values signifying true, and zero signifying false.

All variables are stored in a global structure

```
struct SYSTEM_VARIABLES
{
   clk clk_GLOBAL_...;
   …
   const int int_GLOBAL_...:
   …
   int int_GLOBAL_...:
   …
   clk clk_PROCESS_A_...;
   …
   …
};
extern SYSTEM_VARIABLES SystemVariables;
```

*clk*s are classed that keep time difference, e.g. setting them stores the current tick at the time of the set (if the value set is zero, otherwise the current tick minus the value set), and calculating the difference to the current clock when accessed.

All accesses to the variables are protected with a global Dispatcher mutex. To ensure value consistency across an entire expression calculation, the mutex is locked at the start of an expression calculation rather than for each specific variable access.

4.1.9. Running the engine

Once the code is generated the user can write a main.cpp file to setup the engine and run it.
The most important parameter to configure is the clock multiplier. This parameter defines the ratio between engine ticks and microseconds. For example, calling

```
Dispatcher d;
d.set_clock_multiplier(1000);
```

Will generate a tick every millisecond.

The next step is to connect all the different callbacks and hardware events.

Finally the user specifies which processes to execute by calling the Dispatcher::add_process() function for each process, and starts the engine using Dispatcher::run() (this function never returns).

4.1.10. Case Study: Two Mode Light

To complete the discussion of the code generator we will examine a simple case end-to-end: the case discussed in class of a light with two modes – low and bright. The first button press turns the light on in low mode. A second button press within less than 5 seconds from the first will trigger the bright stage, otherwise it will turn the light off. A press when the light is bright will also turn the lite of. This EFSM can be described in UPPAAL using the following system:

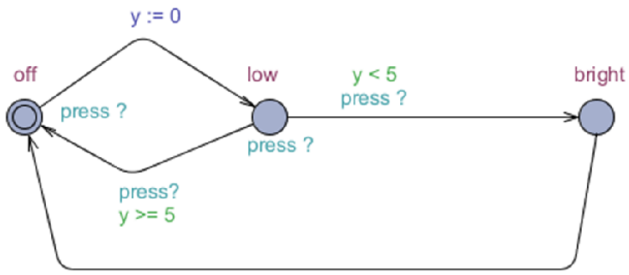
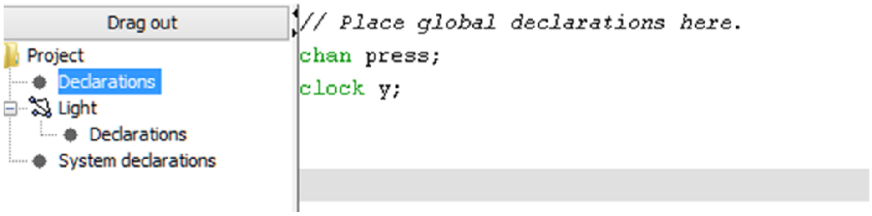


Figure 17

with the following declarations:



Running the code generator on the UPPAAL XML file generates the following **Model.h** file:

```
// THIS FILE WAS AUTO GENERATED BY model2mbed.exe DO NOT MODIFY!!!
#include "ModelImpl.h"
namespace Model {
```

```
// Channels
// chan press
extern ModelImpl::SYNCHRONIZATION_CHANNEL GLOBAL_CHANNEL_VAR_press;

struct SYSTEM_VARIABLES
{
   clk clk_GLOBAL_y;
};

extern SYSTEM_VARIABLES SystemVariables;
// Process - Light
extern ModelImpl::TRANSITION_ENTRY Light_TRANS_bright__off__0;
extern ModelImpl::TRANSITION_ENTRY Light_TRANS_low__bright__0;
extern ModelImpl::TRANSITION_ENTRY Light_TRANS_low__off__0;
extern ModelImpl::TRANSITION_ENTRY Light_TRANS_off__low__0;
extern ModelImpl::STATE_ENTRY Light_STATE_bright;
extern ModelImpl::STATE_ENTRY Light_STATE_low;
extern ModelImpl::STATE_ENTRY Light_STATE_off;
extern ModelImpl::PROCESS_ENTRY Light_PROCESS;
extern ModelImpl::PROCESS_ENTRY_STATE Light_PROCESS_STATE;
} // namespace
```

The generator also generates Model.cpp with the appropriate definitions (and is too long to include here).

We can then proceed to register callbacks and start the engine, in **main.cpp** (simulating the variable light using two LEDs):

```
#include "Model.h"

Serial pc(USBTX, USBRX); // tx, rx
DigitalOut led1(LED1), led2(LED2);

void fun_off(ModelImpl::STATE_ENTRY* ){
   led1 = 0; led2 = 0;
}
void fun_low(ModelImpl::STATE_ENTRY*) {
   led1 = 1;
}
void fun_high(ModelImpl::STATE_ENTRY*){
   led2 = 1;
}
int main (void) {
   ModelImpl::Dispatcher d;
   d.set_clock_multiplier(1000000 /*us*/); // a tick each second


   d.set_state_enter_cb(&Model::Light_STATE_off, fun_off, NULL);
   d.set_state_enter_cb(&Model::Light_STATE_low, fun_low, NULL);
   d.set_state_enter_cb(&Model::Light_STATE_bright, fun_high, NULL);
```

```
  disp.add_process(&Model::Light_PROCESS_STATE);
  disp.run();
}
```

All the rest of the code is generated automagically.

Instead of lighting the LEDs manually using callback functions, we could have used the engine's built-in support for hardware events:

```
// off to low:
d.set_channel_action(&Model::Light_TRANS_off__low__0,
                     LED1, ModelImpl::SendChannelModeSet);
// low to off:
d.set_channel_action(&Model::Light_TRANS_low__off__0,
                     LED1, ModelImpl::SendChannelModeReset);
// low to bright:
d.set_channel_action(&Model::Light_TRANS_low__bright__0,
                     LED2, ModelImpl::SendChannelModeSet);
// bright to off:
d.set_channel_action(&Model::Light_TRANS_bright__off__0,
                     LED1, ModelImpl::SendChannelModeReset);
d.set_channel_action(&Model::Light_TRANS_bright__off__0,
                     LED2, ModelImpl::SendChannelModeReset);
```

That's it – no more code is needed. The generated code with the engine and main.cpp were compiled and tested successfully both in the simulator environment and on an actual MBED.

Cool!

4.1.11. Appendix 1: Using the tool

The tool is named model2mbed.exe, and was developed using C#.

To execute the tool, use the following syntax:

```
model2mbed <uppaal xml file> <outdir> @templates.txt
```

or

```
model2mbed <uppaal xml file> <outdir> template1 [template2 […]]
```

The first version will read the list of templates for code generate from the file templates.txt, the second version will generate code for the templates listed on the output file.

The model2mbed tool will generate Model.cpp and Model.h that contain the model data and code definitions. The tool will also write the engine files, ModelImpl.cpp, ModelImpl.h and ModelPriv.h to the output directory if they do not already exists there.

4.1.12. Appendix2: UPPAAL support

As stated before, most of the UPPAAL semantics have been implemented. Unfortunately, due to time restrictions, several UPPAAL features were not implemented. Most of these are quite easy to add to the existing framework with minimal changes. The implementation is based on the features in UPPAAL 4.0.14 as described in the language reference found on www.uppaal.com (http://www.uppaal.com/index.php?sida=217&rubrik=101).

The code generator assumes the UPPAAL XML file was syntax checked by UPPAAL, although most errors will be detected by it as well (although error descriptions are somewhat vague).

4.1.12.1. Declarations

The code generator fully supports *int*, *bool*, *chan, clock*. Constants are supported and marked as such by the generator. Only one dimensional arrays are supported, but these cannot be pre-initialized to array values (e.g. replace `int a[2] = {1,2}` with `int a[2]; a[0] = 1; a[1] = 2` for the code generator to work). Variables initialization is optimized at code generation time. The *meta* prefix has no effect on the generator. The generator fully supports *typedef*s and ranged variables. s*tructs* are not currently supported.

Channels can be normal, broadcast and/or urgent. Channel priorities are fully supported.

4.1.12.2. Operators and functions

The code generator supports all operators specified by the UPPAAL specification, including: addition, subtraction, unary plus, unary minus, ++ prefix and postfix, -- prefix and postfix, multiplication, division, remainder, bit shift left, bit shift right, bitwise and, bitwise or, bitwise xor, >, >=, <=, <, equal, not equal, min, max, logical not, logical and, logical or, conditional (x ? y : z).

4.1.12.3. Custom Functions

Custom functions are not yet supported, as well as any code statements other than assignment.

4.1.12.4. Templates

Templates can only be single instantiated with no support yet for template parameters. Nodes (locations) can be either regular, urgent or committed. Nodes can have guards (invariants), and transitions (edges) can have guards, synchronizations and updates, although selections are not yet supported.

4.2. Simulation Environment

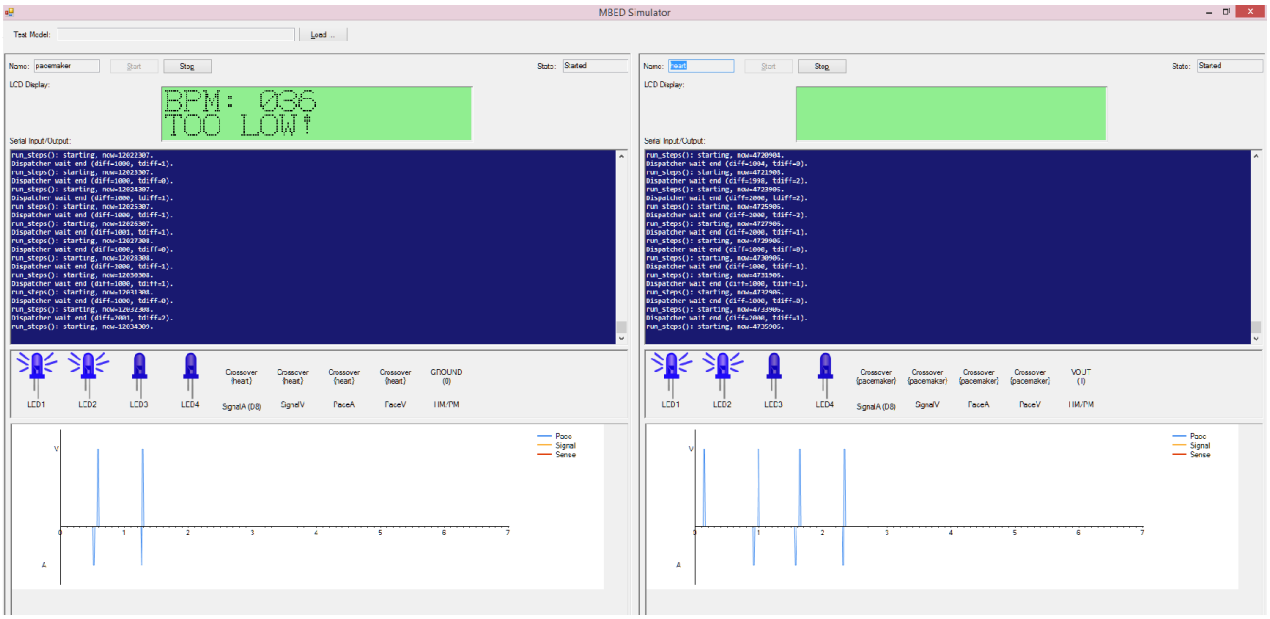


Figure 18

4.2.1. Overview

I wrote the MBED simulator for Windows to ease development for the MBED platform, allowing for more rapid programming and more complete testing.

The simulator does not require any code changes to the MBED code, but simply compiles it with the Visual C++ compiler and links it against a library simulating MBEDs OS behavior. The simulator is very versatile, supporting the simulation of many hardware devices, including the interconnecting of two MBEDs. The simulator also supports extending its GUI with custom visualizers that parse the serial output of the programs. All output is logged to a file for later processing. The devices supported are fully interactive (pressing on a button will raise the respective pin, an out pin will light a LED, etc', the serial port supports both input and output).

For example, below is a screenshot of two Homework 5 timers, running side by side in the simulator with **no code changes** except the addition of an additional LED, button and switch to test the system.

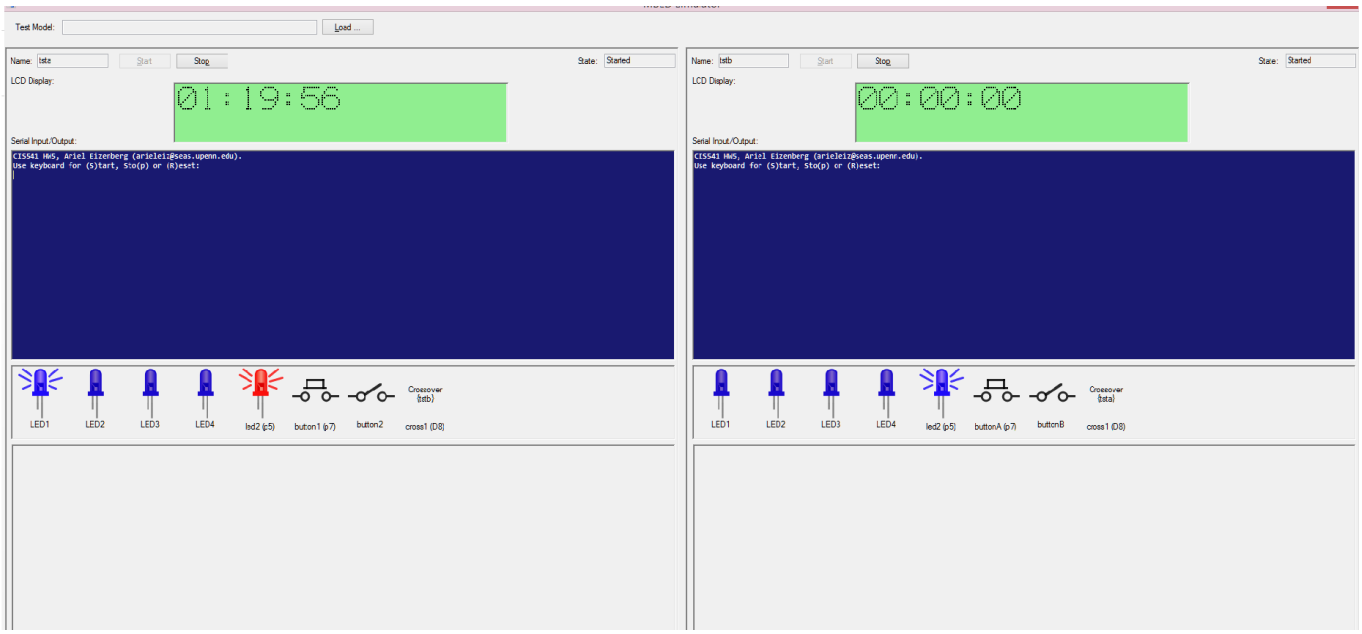


Figure 19

4.2.2. Implementation

Since this is an embedded development course I won't go into detail regarding the UI implementation, but rather focus on the MBED OS wrapper.

The MBED's OS implement the CMSIS OS interface, so creating a CMSIS compatible interface proved almost enough (there are some complex issues with shutdown races etc').

The following features are currently implemented:

- Threads, including signaling (signaling is not natively available on Windows and was implemented from scratchs),
- Timers, Tickers, RTOSTimer, etc' –reading and interrupts,
- Mutexes,
- GPIO read, write, interrupts,
- TextLCD,
- Serial Port – reading, writing, interrupts, break,
- Sleep,
- NVIC_SystemReset.

Additional features can be implemented quite simply as required.

An interesting quirk was actually working around an issue global constructors. The issue here is that the model is implemented as a DLL, and global objects (like global DigitalIn, Thread, Mutex, etc') are initialized before DllMain or exported funcations had a chance to run, and thus the CMSIS stub was not yet connected to the simulator. The solution was to store the connection data on the stack before a call to LoadLibrary, and from the CMSIS stub search the entire stack upwords for this data, locating it by looking for specific magic values.

4.2.3. Configuration

The simulator is configured using an XML file describing the system to test.

Consider an example configuration file for two MBEDs:

```
<?xml version="1.0" encoding="UTF-8"?>
<mbedsimulator>
	<mbed name="pacemaker" dll="mbedsimulatortst4.dll" viz="HeartRateVirualizer.dll">
		<textlcd/>
		<serial/>
		<crossover name="SignalA" pin="p9" extra="heart"  />
		<crossover name="SignalV" pin="p10" extra="heart"  />
		<crossover name="PaceA" pin="p11" extra="heart"  />
		<crossover name="PaceV" pin="p12" extra="heart"  />
		<ground name="HM/PM selector" pin="p13" />
	</mbed>
	<mbed name="heart" dll="mbedsimulatortst4.dll" viz="HeartRateVirualizer.dll">
		<textlcd/>
		<serial/>
		<crossover name="SignalA" pin="p9" extra="pacemaker"  />
		<crossover name="SignalV" pin="p10" extra="pacemaker" />
		<crossover name="PaceA" pin="p11" extra="pacemaker" />
		<crossover name="PaceV" pin="p12" extra="pacemaker"  />
		<vout name="HM/PM selector" pin="p13" />
	</mbed>
</mbedsimulator>
```

The file defines two MBEDs:

- The first is named “pacemaker”, and is implemented by `mbedsimulatortst4.dll.` It is connected to a TextLCD, a serial port, and pin 13 is connect to the ground.
- The second is named “heart”, it is also implemented by `mbedsimulatortst4.dll.` It is connected to a TextLCD, a serial port, and pin 13 is connected to Vout.
- The two MBED’s are connected to each other using four lines, mapped in both MBEDs to the same PINs (9, 10, 11, 12).

Each MBED comes with four default LEDs connect to the pins LED1, LED2, LED3 and LED4.

The configuration file supports additional elements as can be seen in the next XML configuration file:

```
<mbedsimulator>
	<mbed name="tsta" dll="mbedsimulatortst1.dll">
		<textlcd/>
		<serial/>
		<led name="led2" pin="p5" extra="red" />
		<pushbutton name="button1" pin="p7" />
		<switch name="button2" pin="p8" />
	</mbed>
</mbedsimulator>
```

This file defined just one MBED, with a TextLCD, a serial, an extra LED with red color connected to pin 5, a push button connect to pin 7 and a toggle switch on pin 8. The system currently supports LEDs with red, blue, green and yellow colors.

4.2.4. External Testing Tools

4.2.4.1. Analysis

Static analysis on the pacemaker project was performed using the code analysis feature of Visual Studio.

The pacemaker project does not use any dynamic memory allocations, but if necessary the project can be verified using the CRTs builtin leak detection mechanism.

4.2.4.2. Code Coverage

The Visual Studio Team CodeCoverage.exe tool was used to verify complete code coverage during testing.

4.3. Embedded Development Environment

As stated in the introduction, the online MBED platform is very limited and lacks many useful IDE features and most importantly a debugger. For example, during the development of the thread wrapper in the execution engine, an incorrect stack size parameter was passed to osThreadCreate in osThreadDef (size of pointer and not size of pointed data). The MBED simply froze up, and there was no way to understand what the problem was. The solution was to completely abandon the online MBED IDE and use a traditional GCC cross compiler to the ARM CONVEX M3.

To setup the environment, the following packages were used:

- GCC-ARM-EMBEDDED 4.8 downloaded from https://launchpad.net/gcc-arm-embedded/4.8
- PyOCD, the on chip debugger from https://launchpad.net/gcc-arm-embedded-misc/pyocd-binary/
- The MBED platforms' firmware was upgraded to 141212 to enable CMSIS-DAP support (http://developer.mbed.org/media/uploads/samux/mbedmicrontroller_141212.if)

The initial CORTEX M3 MBED environment was generated by exporting the MBED project from the online IDE to the GCC-ARM-EMBEDDED platform, and then the Makefile was manually edited to adjust it to the project. Another change was improving optimizations as the original resulting binary was significantly larger than the online environment's.

The GCC version does not calculate the generated binary's vectored exception handler checksum, so I wrote a tool, mbedcsum, to fix the checksum in the binaries so they could be used directly on the MBED without requiring GDB to fix the value at load time.

Actual debugging is done by running PyOCD, starting a gdb session in parallel and running the following commands:

target extended-remote localhost:3333	Connect to PyOCD
set remote hardware-breakpoint-limit 6	Configure M3's hardware breakpoints
set remote hardware-watchpoint-limit 4	Configure M3's hardware watchpoints
define hook-step mon cortex_m maskisr on end	} Mask ISRs before a single step
define hookpost-step mon cortex_m maskisr off end	} Unmask ISRs after a single step
continue	Start program

One has to be careful though, the GCC generates bad code when optimizing with –Os.

4.4. Hardware Implementation

4.4.1. Overview

The project was implemented using two NXP LPC1768 MBED platforms on a single breadboard, with a 16x2 HD44780 compatible LCD connected to each (through a potentiometer). The pacemaker is also connected to a buzzer for generating alarms. **The connection between the two MBEDs are pulled up when idle, and pulsed down to signal pace/signal.**

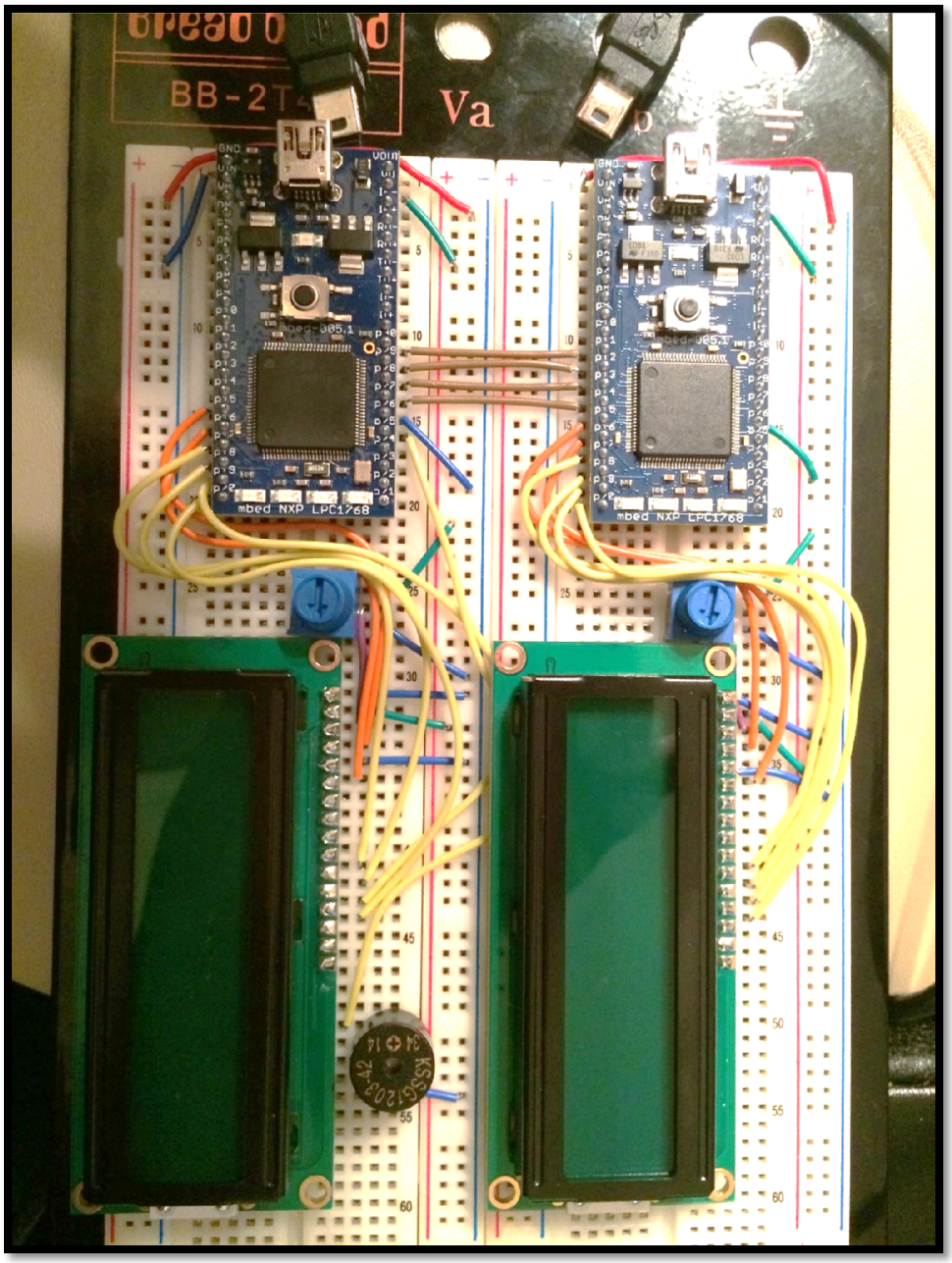


Figure 20

4.4.2. Wiring Diagram

The software uses p26 to detect if it is running on the correct hardware.

The pacemaker hardware has pin 26 connected to the ground, and the heart has it connect to vout.

The two MBEDs are connected using four wires. These connections are pulled-up, and lowered when signaled.

Heart to pacemaker
p30-p11 – A Signal
p29-p12 – V Signal

Pacemaker to heart
p28-p10 – A Pace
p27-p8 – V Pace

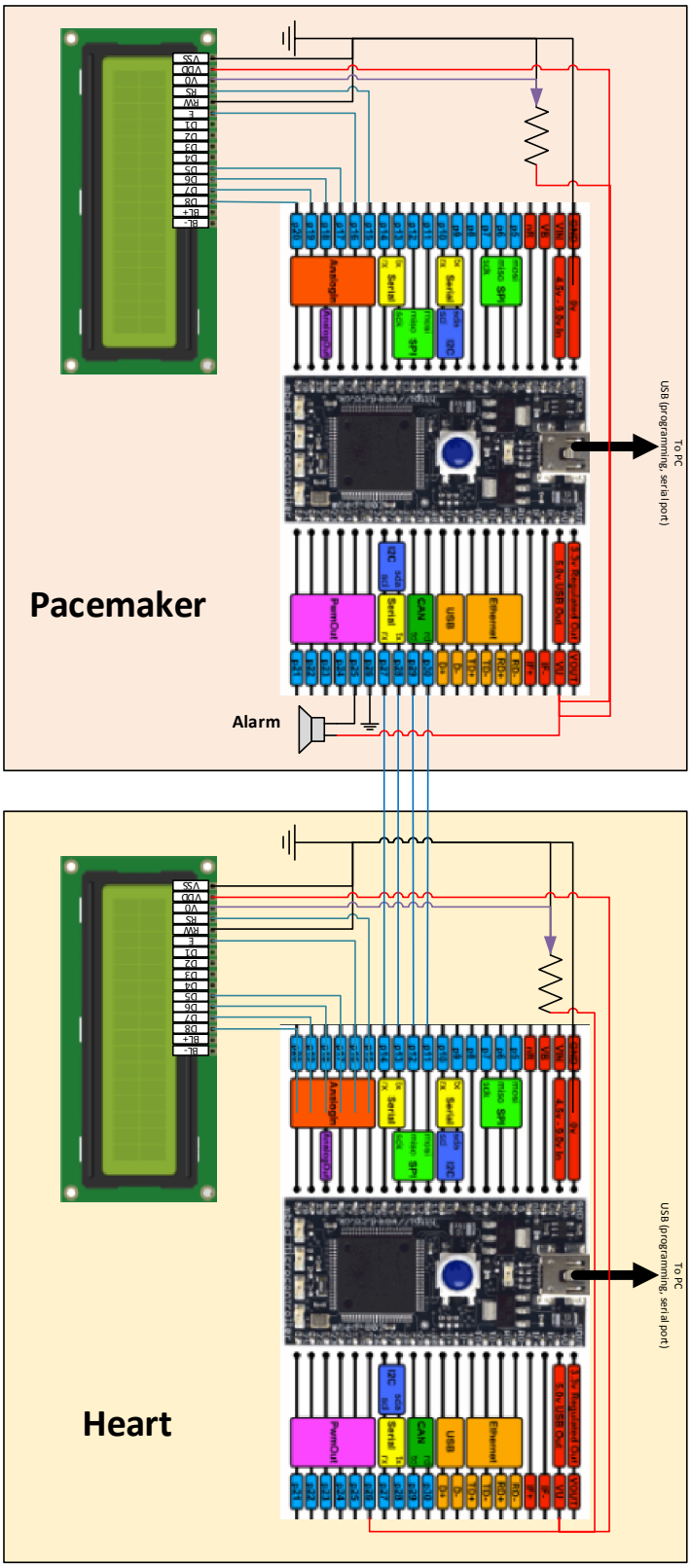


Figure 21

4.5. Software Implementation

Using the code generation tool described above, the actual pacemaker implementation is very simple and can be seen in main_pacemaker.cpp and main_heart.cpp.

Setting up the serial port traces:

```
d.set_channel_fire_cb(&Model::GLOBAL_CHANNEL_VAR_Apace,
                          print_event, (void*)"APACE");
d.set_channel_fire_cb(&Model::GLOBAL_CHANNEL_VAR_Vpace,
                          print_event, (void*)"VPACE");
d.set_channel_fire_cb(&Model::GLOBAL_CHANNEL_VAR_Asense,
                          print_event, (void*)"ASENSE");
d.set_channel_fire_cb(&Model::GLOBAL_CHANNEL_VAR_Vsense,
                          print_event, (void*)"VSENSE");
d.set_channel_fire_cb(&Model::GLOBAL_CHANNEL_VAR_Asignal,
                          print_event, (void*)"ASIGNAL");
d.set_channel_fire_cb(&Model::GLOBAL_CHANNEL_VAR_Vsignal,
                          print_event, (void*)"VSIGNAL");
```

The print_event function simply writes the context to the serial port.

Setting up alarms:

```
d.set_channel_fire_cb(&Model::GLOBAL_CHANNEL_VAR_AlarmL,
                          show_alarm, (void*)"TOO LOW! ");
d.set_channel_fire_cb(&Model::GLOBAL_CHANNEL_VAR_AlarmH,
                          show_alarm, (void*)"TOO HIGH!");
d.set_channel_fire_cb(&Model::GLOBAL_CHANNEL_VAR_NoAlarm,
                          show_alarm, (void*)"NORMAL   ");
```

The show_alarm function simply writes the context to the second line of the TextLCD.

The modeswitch display is setup in the same way on the four channels used in that process.

LED activation is tied to the signals by pulsing them for 25ms when the appropriate channels are triggered:

```
d.set_channel_action(&Model::GLOBAL_CHANNEL_VAR_Apace,
        LED1, ModelImpl::SendChannelModePulseUp, 250000);
d.set_channel_action(&Model::GLOBAL_CHANNEL_VAR_Vpace,
        LED2, ModelImpl::SendChannelModePulseUp, 250000);
d.set_channel_action(&Model::GLOBAL_CHANNEL_VAR_Asense,
        LED3, ModelImpl::SendChannelModePulseUp, 250000);
d.set_channel_action(&Model::GLOBAL_CHANNEL_VAR_Vsense,
        LED4, ModelImpl::SendChannelModePulseUp, 250000);
```

The connection between to the heart MBED is defined similarly:

```
// output, pace the heart - short pulse (40ms)
```

```
d.set_channel_action(&Model::GLOBAL_CHANNEL_VAR_Apace,
        PIN_PACE_A, ModelImpl::SendChannelModePulseDown, 40000);
d.set_channel_action(&Model::GLOBAL_CHANNEL_VAR_Vpace,
        PIN_PACE_V, ModelImpl::SendChannelModePulseDown, 40000);

// input - interrupt on rise
d.set_receive_input(&Model::GLOBAL_CHANNEL_VAR_Asignal,
    PIN_SIGNAL_A, PullUp, ModelImpl::ReceiveChannelInterruptRise);
disp.set_receive_input(&Model::GLOBAL_CHANNEL_VAR_Vsignal,
    PIN_SIGNAL_V, PullUp, ModelImpl::ReceiveChannelInterruptRise);
```

The heart rate is shown by specifying an entry callback to the ShowRate node of the PM_DisplayAndAlarm template.

The only manual code written was for the keyboard input loop (this could have been implemented using a UPPAAL model, but it is just stretching this too far in my opinion). To implement this without busy waiting, a separate thread was dedicated to the loop, and the serial interrupt is configured to wake the thread to accept the input.

That's it – no other code in the model except for the code automatically generated by model2mbed.

4.6. Operational Parameters

4.6.1. Pacemaker

LRI & URI limits:

Mode	LRI	URI
Normal	40 bpm	100 bpm
Sleep	30 bpm	60 bpm
Sports	100 bpm	175 bpm
Manual	30 bpm	175 bpm

Timing parameters and limits:

Property	Default	Low	High
A-V interval	30ms	30ms	100ms
A-V interval extension step	5ms		
A-V interval extension maximum	50ms		
VRP	150ms	500ms	500ms
PVARP	150ms	500ms	500ms
PVAB	10ms	60ms	60ms

Keyboard interface:

Key	Action
'N'	Enter normal mode
'S'	Enter sleep mode
'E'	Enter sports (exercise) mode
'M'	Enter manual mode
'A'	In manual mode only - initiate atrial pace
'V'	In manual mode only – initiate ventricular pace
'O' → Number → Enter	Change heart rate observation time to a new value (in milliseconds)

4.6.2. Heart model

Random heart time limits:

Signal	Min	Max
Atrial	500ms	1000ms
Ventricular	500ms	1000ms

Keyboard interface:

Key	Action
'R'	Enter random mode
'T'	Enter test mode (bracyhadra)
'D'	Enter "dead" mode
'M'	Enter manual mode
'A'	In manual mode only – signal atrial event
'V'	In manual mode only – signal ventricular event
'O' → Number → Enter	Change heart rate observation time to a new value (in milliseconds)

4.7. Testing

4.7.1. Tools

4.7.1.1. Data visualizer

The data visualizer tool was developed to collect test logs from the pacemaker and heart, while visually graphing the rate of signals, senses and paces. The tool adds a timestamp to each line of the log so it can be used to automatically test different properties of the implementation as well as correlate to external events.

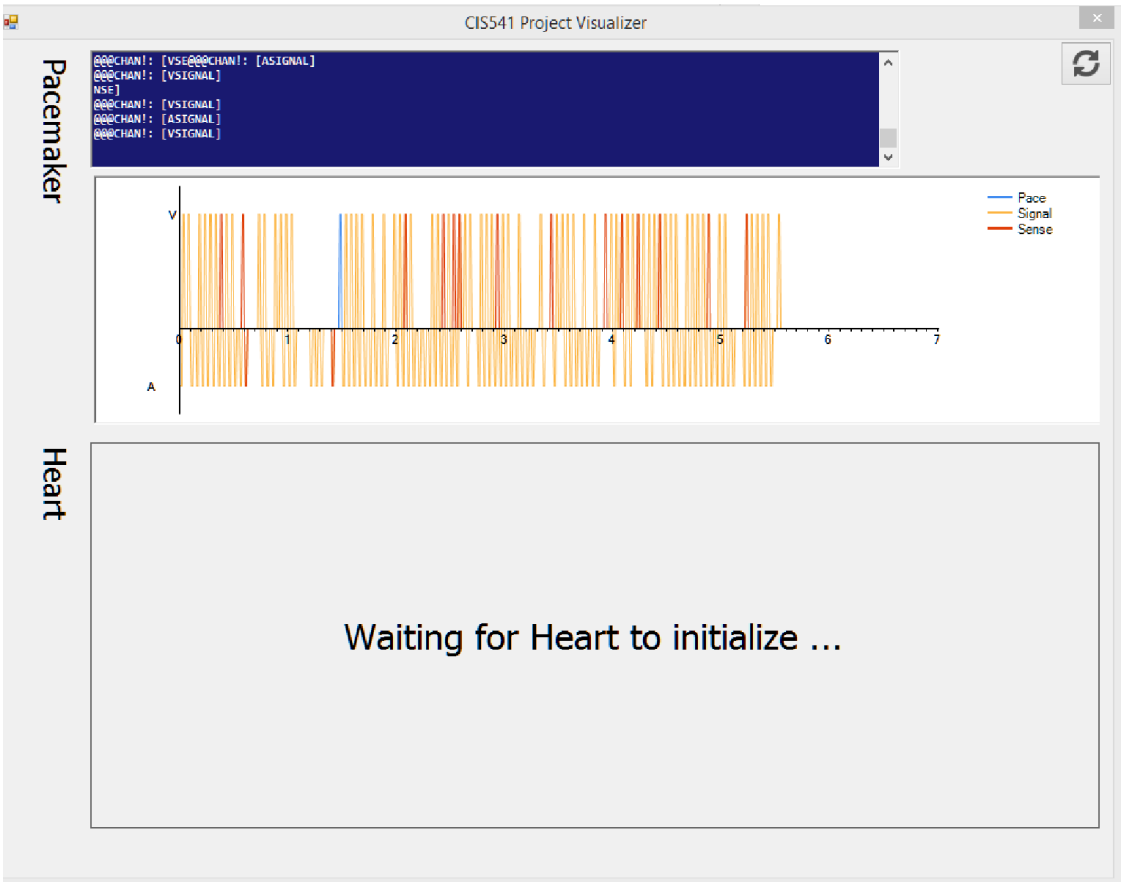


Figure 22

4.7.1.2. Log verification

Logs generated by the heart visualizer are processed automatically using a C# tool to extract the different timing properties.

4.7.1.3. Coverage Testing using Visual Studio's CodeCoverage.exe

Code coverage was tested using Visual Studio's CodeCoverag.exe tool in conjunction with the MBED simulator. This allowed validation that all possible branches have been tested. An example test run can be seen in Figure 23, with actual coverage rates in the bottom pane and source code coloring in the top pane reflecting code usage:

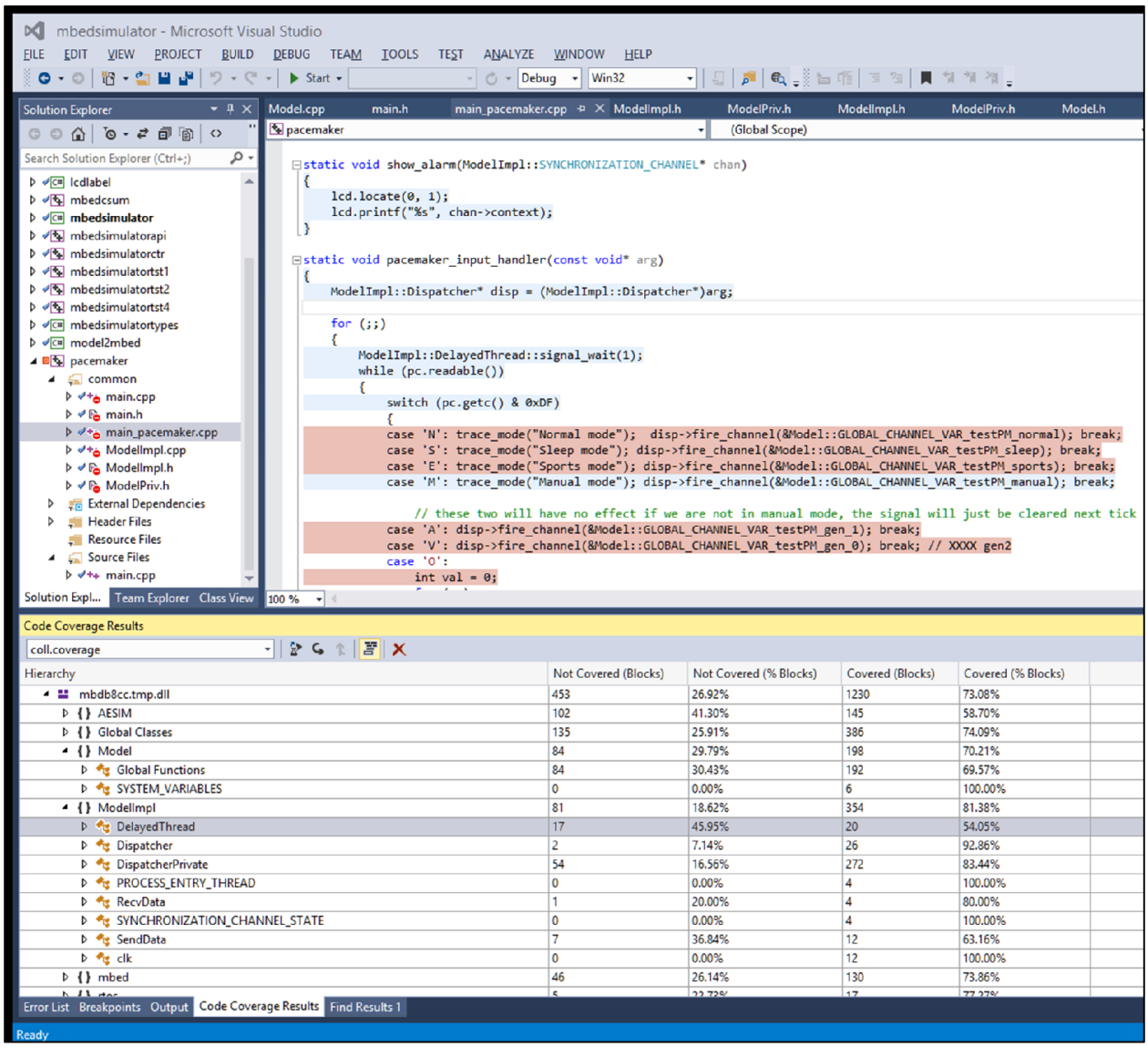


Figure 23 – Code coverage analysis

4.7.2. Runtime Verification

The code was tested for correct runtime operation by generating a version of the software that includes all the observer templates developed as part of the UPPAAL models. Unfortunately, due to the limited size of RAM on the actual hardware, this version of the software could not be tested on the actual hardware. Instead, it was executed on the software simulator for a long period of time. The "bad" were configured in the simulation to generate logs that could later be checked automatically (using grep). UPPAAL implication queries (A[] s implies c) were tested by adding callbacks to state entry.

4.7.3. Functional Testing

Not all tests were completed in time for project submission.

	Mode switch	
4.7.3.1.	Sleep → normal	Set the mode switch to sleep ('S') and then to normal ('N'), while pacemaker is connected to an inoperative heart ("He is dead Jim" test sequence). Expected – pacemaker beats at the normal LRI.
4.7.3.2.	Sports → normal	Set the mode switch to sports ('E') and then to normal ('N'), while pacemaker is connected to an inoperative heart ("He is dead Jim" test sequence). Expected – pacemaker beats at the normal LRI.
4.7.3.3.	Manual → normal	Set the mode switch to manual ('E') and then to normal ('N'), while pacemaker is connected to an inoperative heart ("He is dead Jim" test sequence). Expected – pacemaker beats at the normal LRI.
4.7.3.4.	Manual events	Set the mode switch to manual ('E') and then use 'A' and 'V' to generate paces. Expected – paces can be observed on heart model
	LEDs	
4.7.3.5.	A & V Pace LEDs	Operate pacemaker in normal mode and heart in random mode. Video record led activity and correlate time with heart logs. Expected - Pace LED light activity is correlated with timing in logs
4.7.3.6.	A & V Sense LEDs	Operated pacemaker instrumented with runtime verification observers with heart in random mode. Video record led activity and correlate time observer logs. Expected - Pace LED light activity is correlated with timing in logs
	Pacing	
4.7.3.7.	Normal – inactive	In normal mode, operate heart inside the normal LRI to URI limit. Expected – pacemaker does not pace
4.7.3.8.	Normal - bradycardia	In normal mode, operate heart below normal LRI limit. Expected – pacemaker paces at LRI.
4.7.3.9.	Normal – URI	In normal mode, operate heart in test mode generating AV events in a long – short – long – short pattern. Expected – pacemaker will not pace at over URI.
4.7.3.10	Sleep – inactive	In sleep mode, operate heart inside the sleep LRI to URI limit. Expected – pacemaker does not pace

4.7.3.11	Sleep - bradycardia	In sleep mode, operate heart below sleep LRI limit. Expected – pacemaker paces at LRI.
4.7.3.12	Sleep – URI	In sleep mode, operate heart in test mode generating AV events in a long – short – long – short pattern. Expected – pacemaker will not pace at over URI.
4.7.3.13	Sports – inactive	In sports mode, operate heart inside the sleep LRI to URI limit. Expected – pacemaker does not pace
4.7.3.14	Sports - bradycardia	In sports mode, operate heart below sleep LRI limit. Expected – pacemaker paces at LRI.
4.7.3.15	Sports – URI	In sports mode, operate heart in test mode generating AV events in a long – short – long – short pattern. Expected – pacemaker will not pace at over URI.
	Alarm and Rate	
4.7.3.16	Alarm – High	Configure heart to generate signals at rate higher then URI. Expected – rate alarm “TOO HIGH!” is visible on LCD, and the alarm buzzer starts with the “high” tone.
4.7.3.17	Alarm – Low	Configure pacemaker to manual mode, and heart as well, do not generate any events. Expected – rate alarm “TOO LOW!” is visible on LCD, and the alarm buzzer starts with the “low” tone.
4.7.3.18	Alarm stops (High)	Configure heart to generate signals at rate higher then URI. Wait for alarm to start. Configure heart to generate signals at LRI rate. Expected – alarm stops, LCD text disappears.
4.7.3.19	Alarm stop (Low)	Configure heart and pacemaker to manual mode. Configure heart to generate signals at LRI rate. Expected – alarm stops, LCD text disappears.
4.7.3.20	LCD rate	Configure pacemaker in manual mode, heart at rate between LRI and URI. Expected – LCD display correct rate

4.7.4. Stess Testing

4.7.4.1.	Long duration - normal	Test for 24 hours connected to heart simulator operating in normal mode. Expected - pacemaker does not intervene.
4.7.4.2.	Long duration - bradycardia	Test for 24 hours connected to heart simulator operating in a bradycardia mode. Expected – pacemaker beats at a consistent rate – LRI.
4.7.4.3.	Long duration - random	Test for 24 hours connected to heart simulator operating in a random mode. Expected – LRI, URI, PVARP, VRP, VPC properties are all maintained.
4.7.4.4.	High rate	Test with heart generating beats at 1000bpm. Expected - pacemaker does not intervene and does not crash.

5. Assurance Case

The assurance case was developed according to the goal structured notation (GSN) standard[8] , using the Astah GSN editor (evaluation version, http://astah.net/download).

The assurance case is very big, and thus we will dissect it part by part in the following sections.

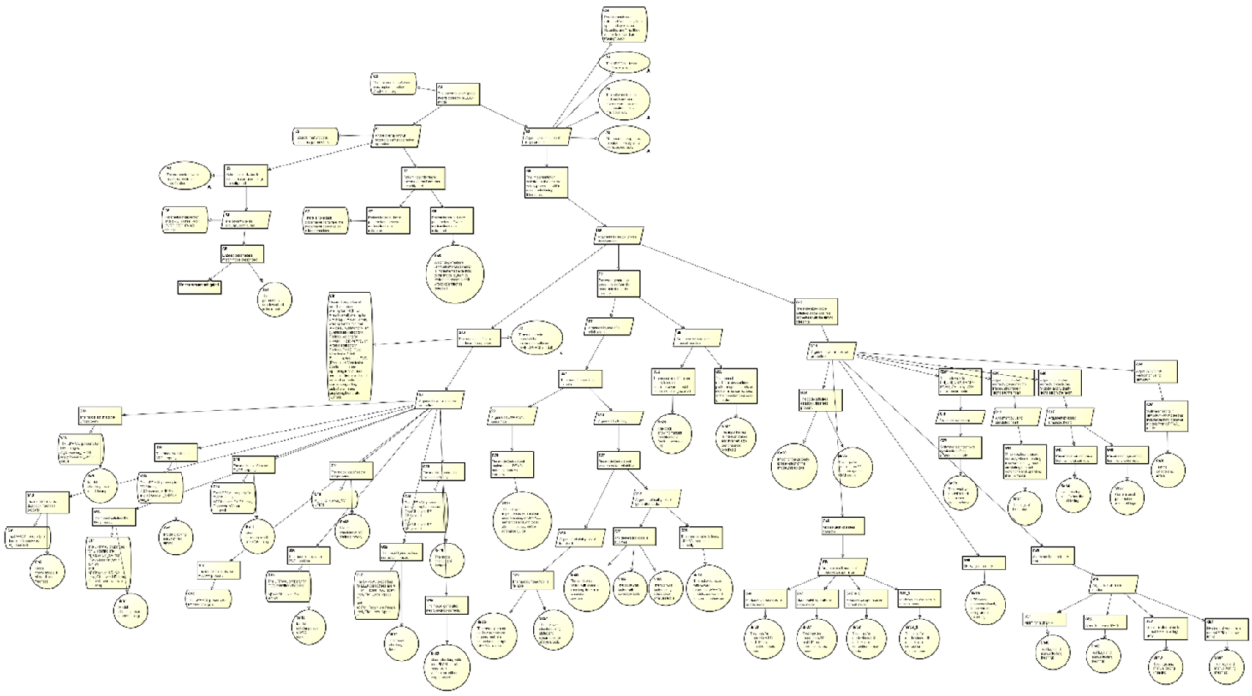
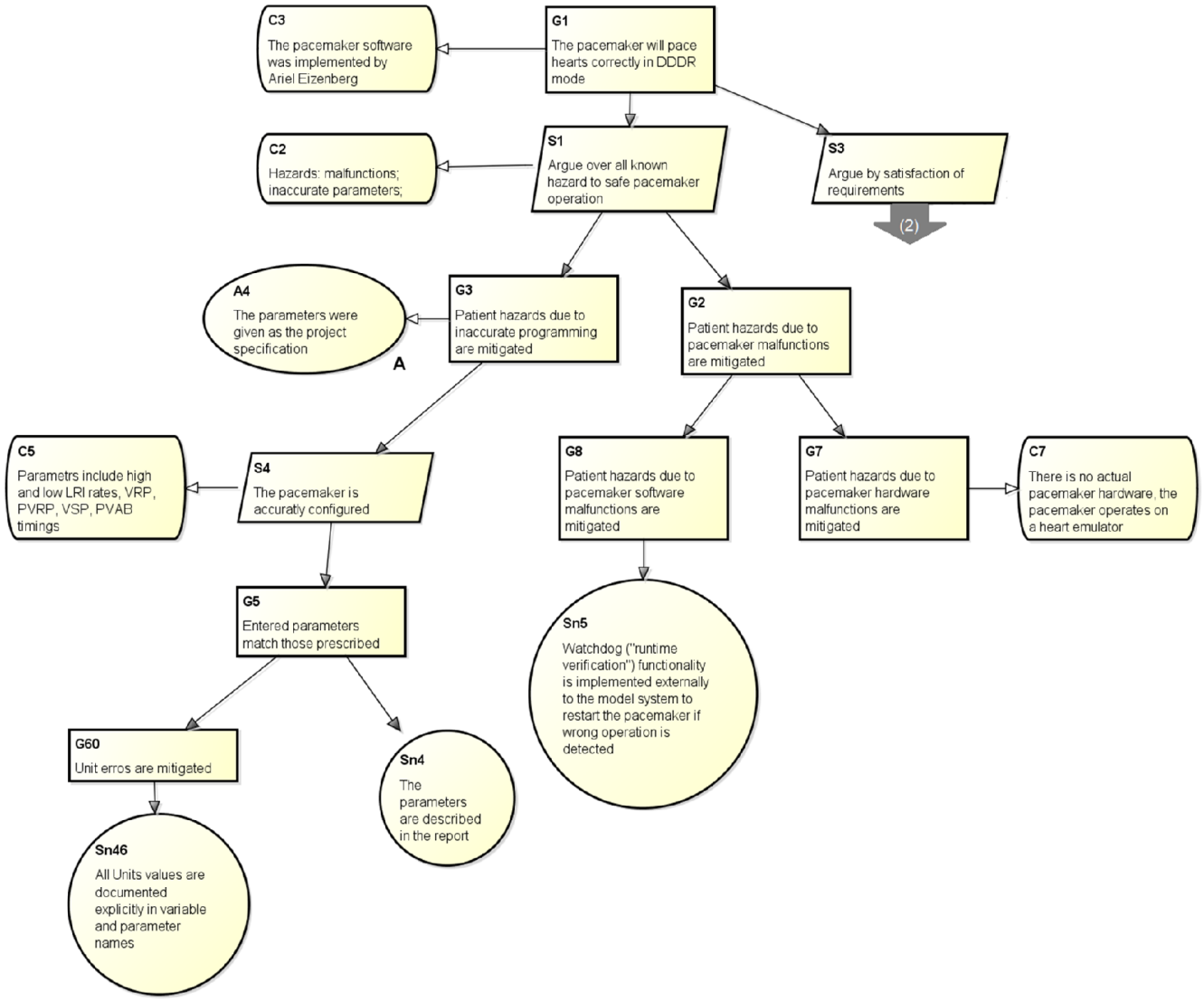


Figure 24

[8] http://www.goalstructuringnotation.info/

5.1. Top Level

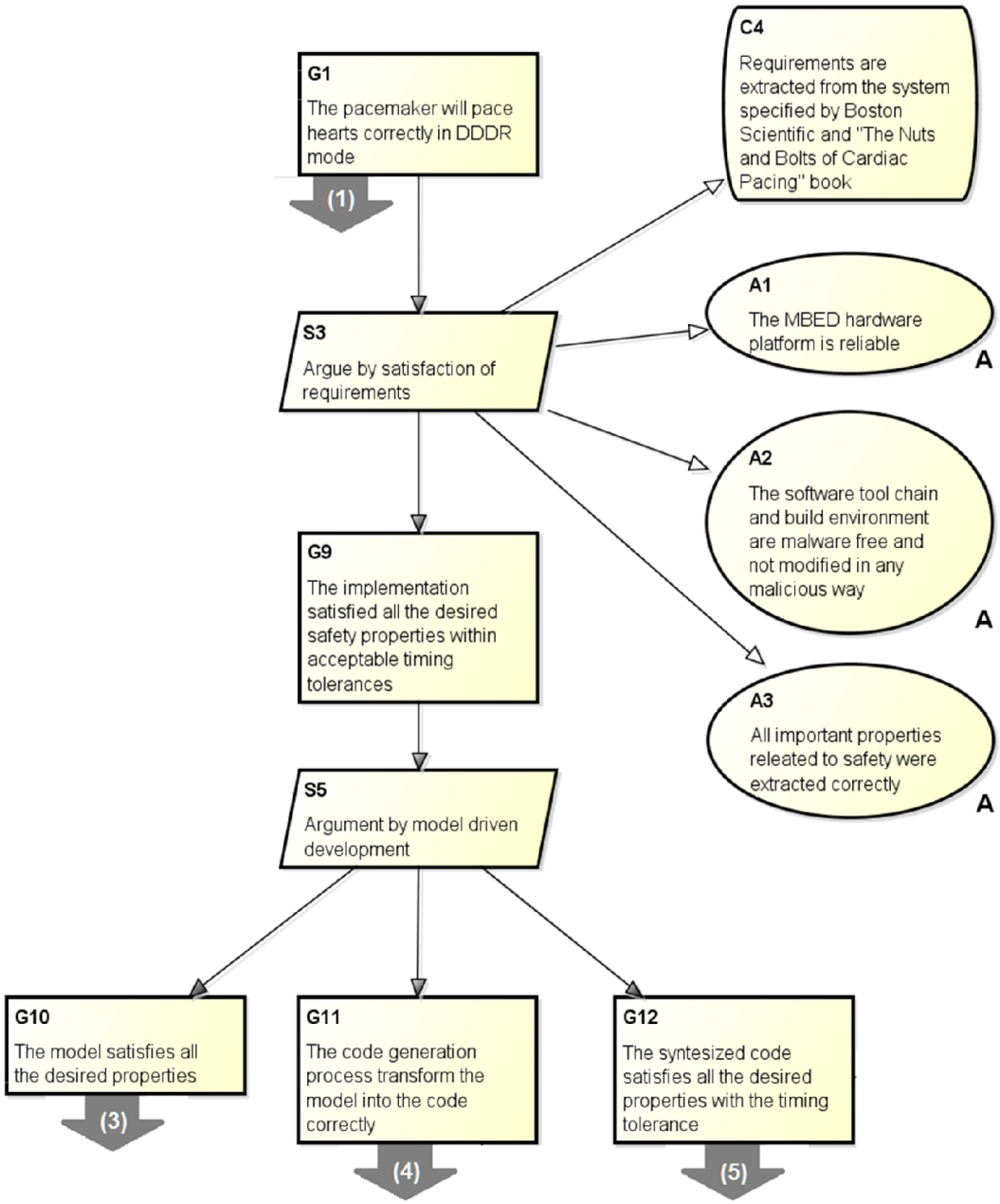
The top level of the assurance case split the goal (G1) of proving the pacemaker will pace hearts correctly into two strategies – (S3) The pacemaker implementation satisfies the requirements, and (S1) The difference hazards during operation are handles in a satisfactory manner. The figure below details the latter.



Assurance Case Figure 1

5.2. Model and Code-Synthesis

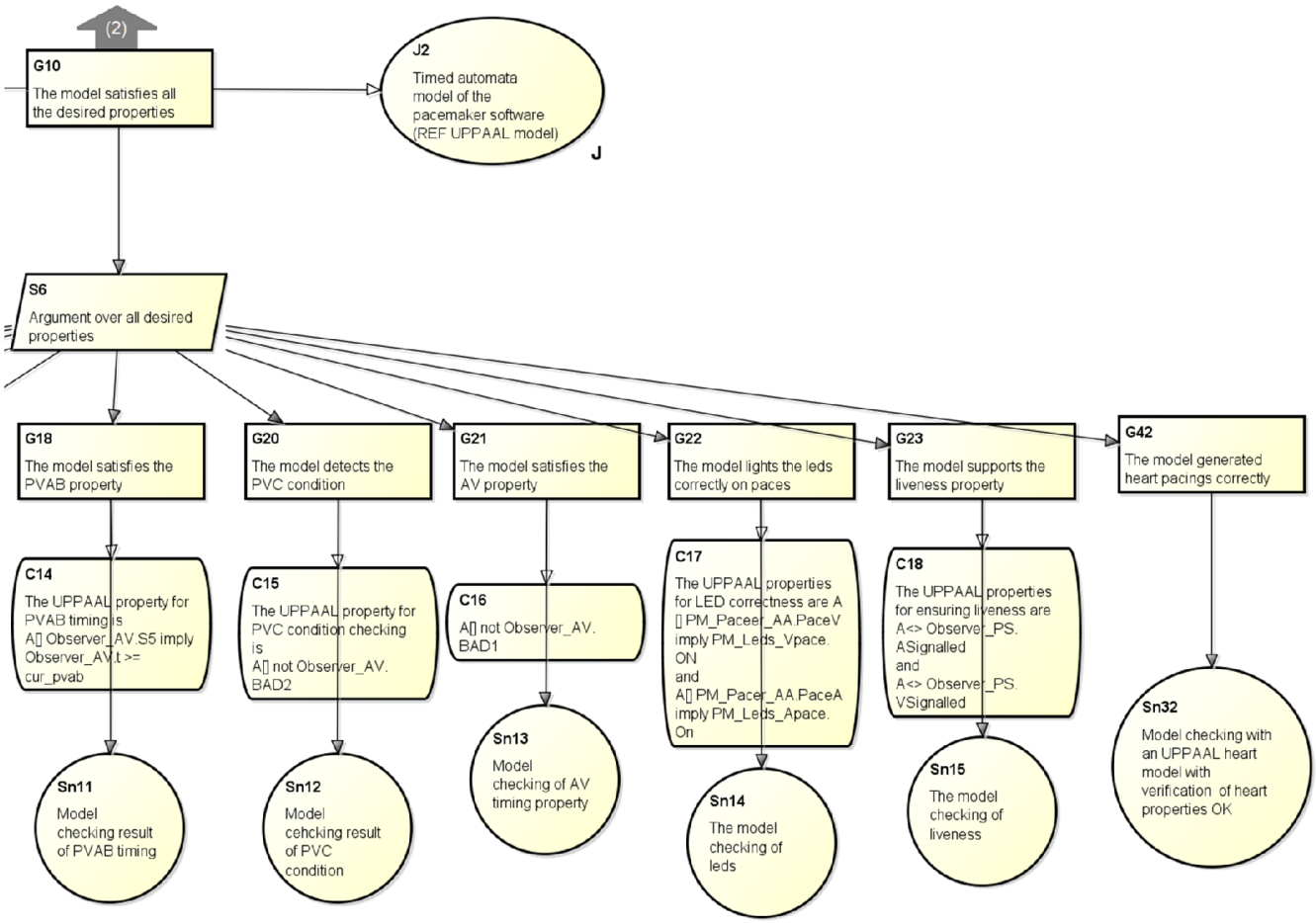
The second half of the assurance case attempts to prove the pacemaker correctly satisfies it requirements. Assuming the hardware is reliable (A1) and that the software tool chain is so as well (A2), and since the pacemaker project was developed using model based design, and transformed into code using a code generation tool developed for this project, we need to prove three goals: the model satisfies all desired properties (G10), the code generation tools operates correctly (G11), and the code generated itself satisfies all relevant requirements (G12).



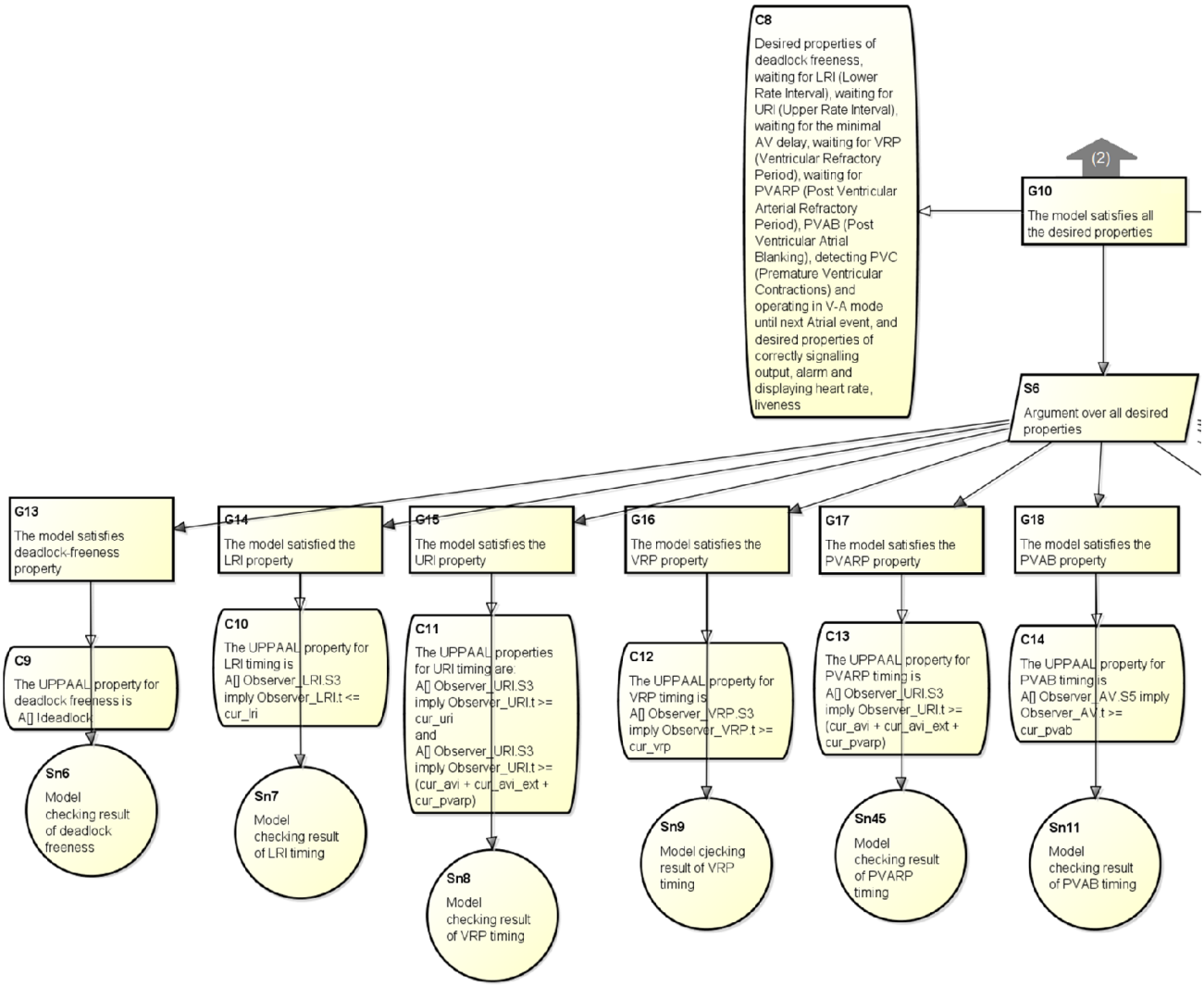
Assurance Case Figure 2

5.3. Model base design

This part of the assurance case deals with all requirements for the model, and the UPPAAL verification method for these properties. The figure was cut into two to improve readability.



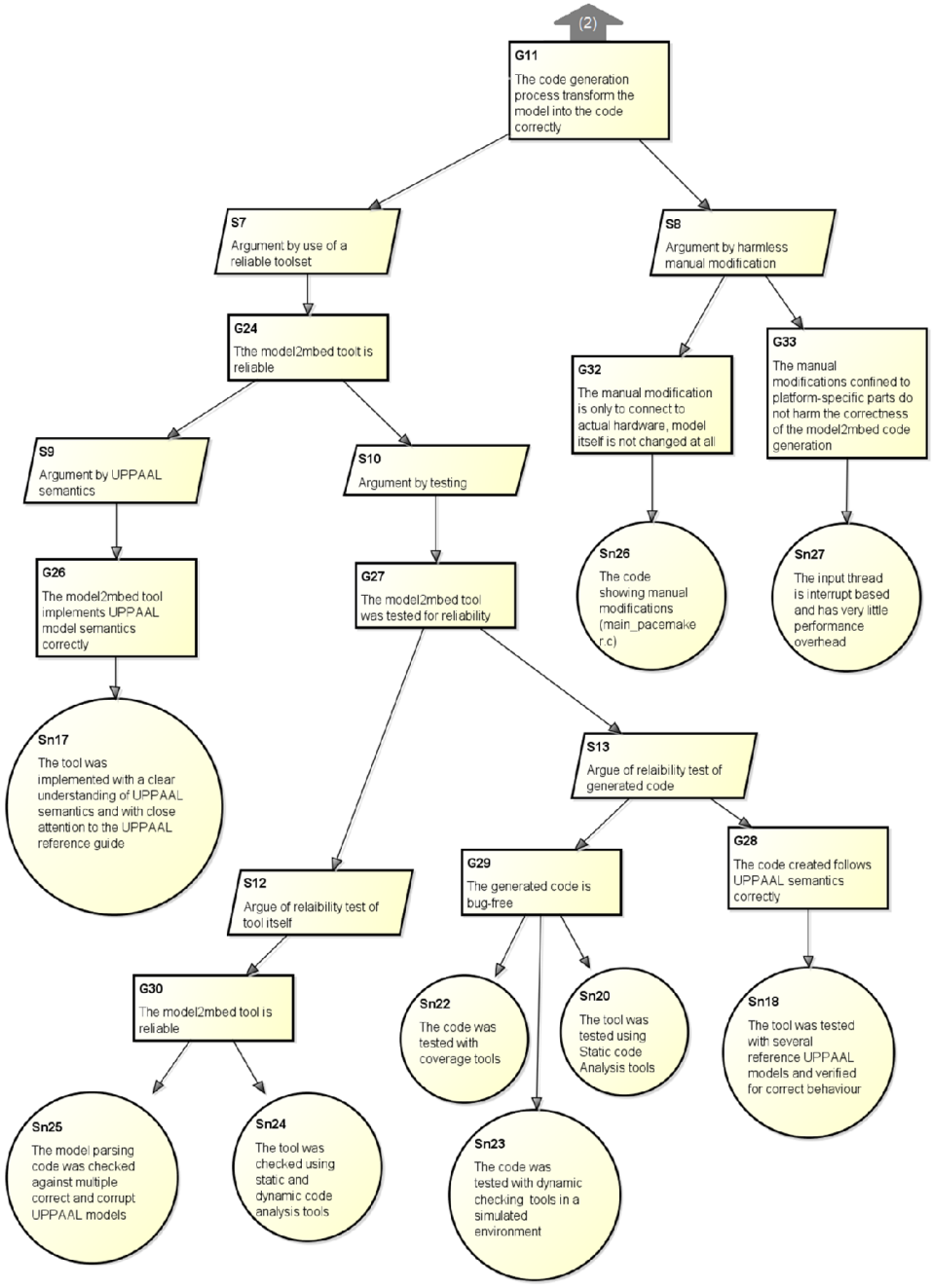
Assurance Case Figure 3(a)



Assurance Case Figure 3(b)

5.4. Code generation tool

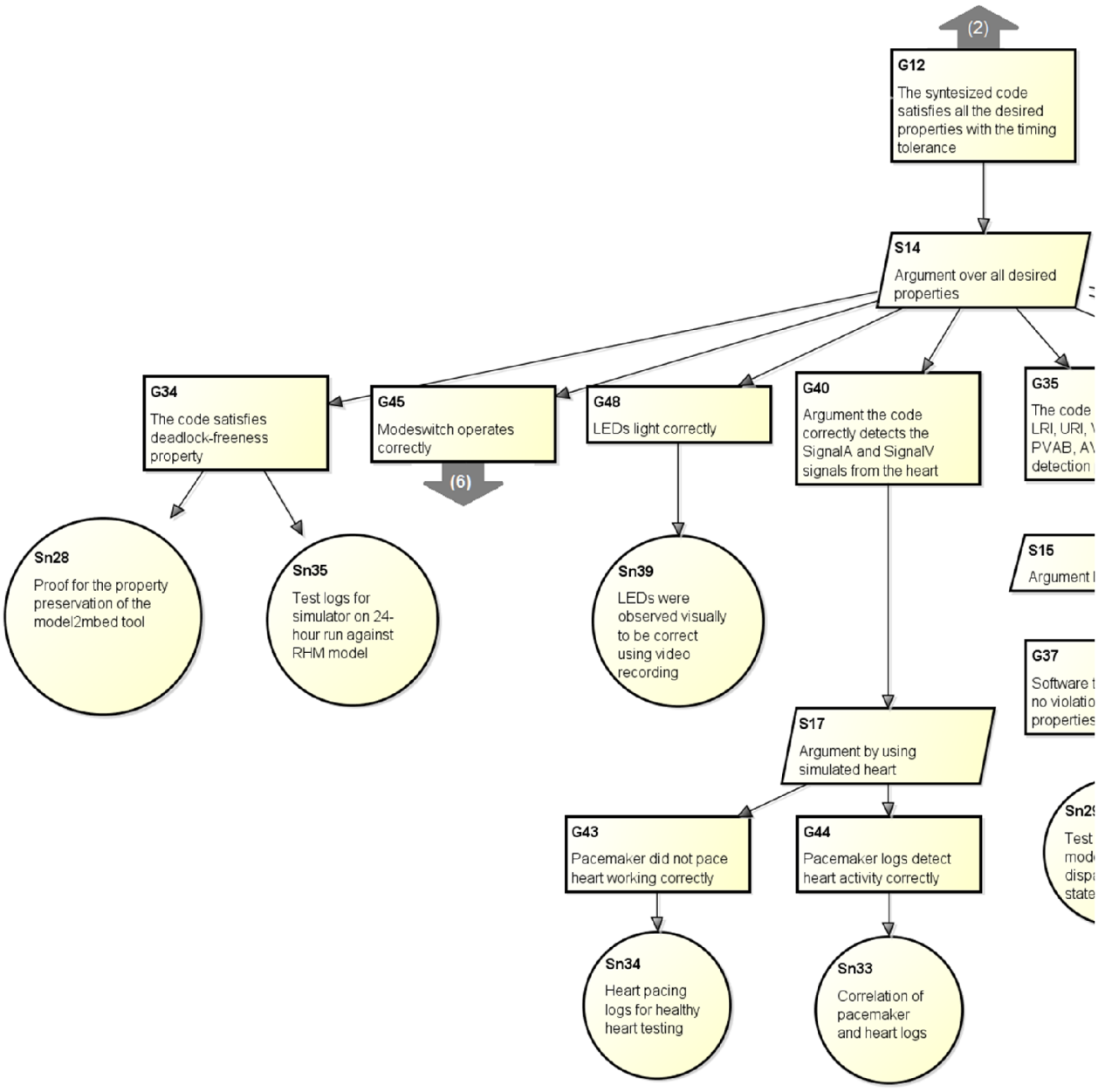
Goal G11 argues the code generation process generates code correctly by arguing the tool itself is reliable (S7), and then arguing the manual modifications do not modify this correctness (S8).



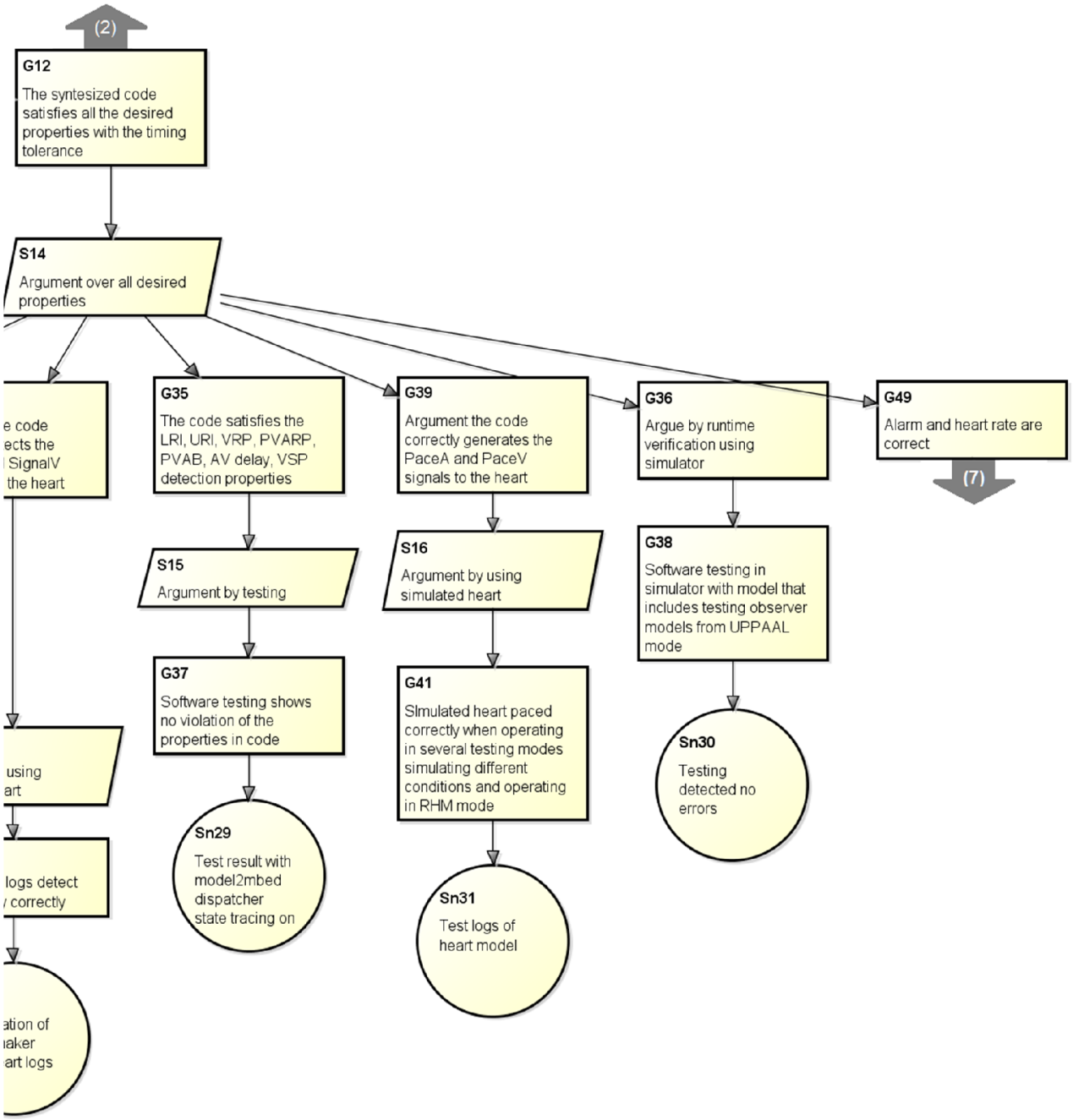
Assurance Case Figure 4

5.5. Generated code correctness

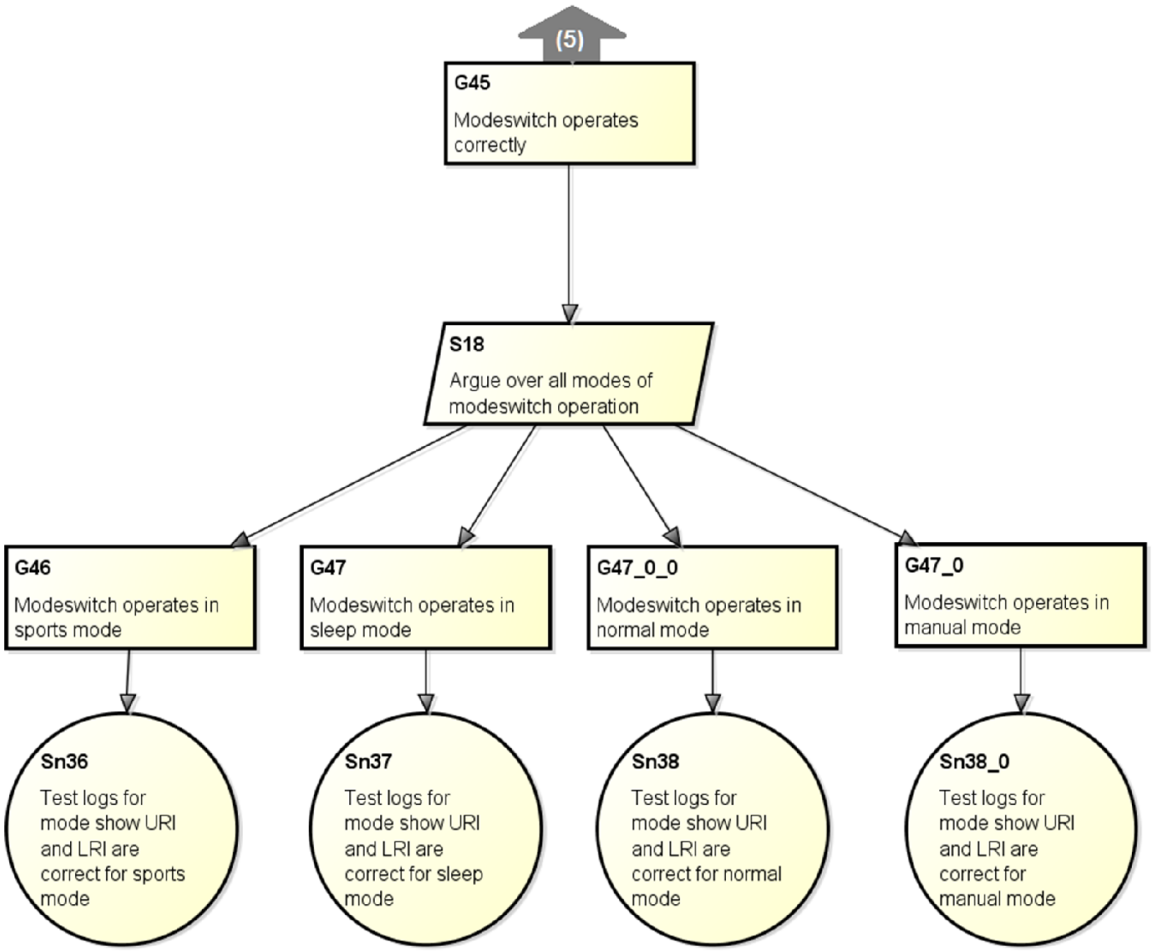
Assurance case figures 5(a) and 5(b) deal with proving the generated goals satisfies the timing requirements by using testing logs as evidence. Figure 6 expands on the modeswitch and figure 7 on the alarm and rate monitor



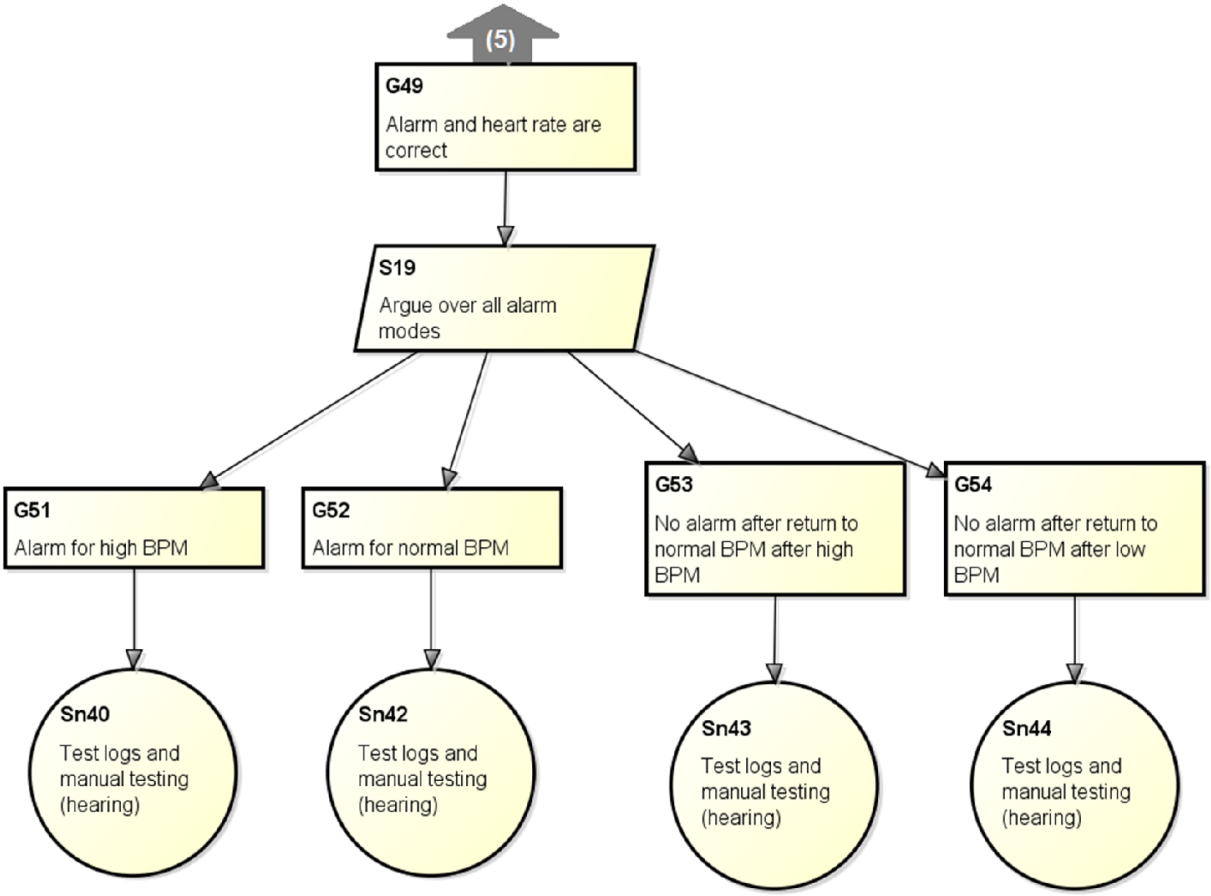
Assurance Case Figure 5(a)



Assurance Case Figure 5(b)



Assurance Case Figure 6



Assurance Case Figure 7

6. Attached Files

The following programs and tools can found in the attached zip file:

- `project.xml & project.q` – UPPAAL model
- `pacemaker` – pacemaker implementation
- `pacemaker_with_rt` – pacemaker implementation with runtime verification using observers
- `heart` – heart implementation
- `model2mbed` – code generator
- `mbedsimulatortst4` – pacemaker & heart as single binary
- `mbedsimulatortst2` – lights example (generated from codegenlights.xml)
- `mbedsimulatortst1` – HW5 for simulator (+other tests)
- `mbedsimulator` – GUI of MBED simulator
- `mbedsimulatortypes` – helper library for simulator
- `mbedsimulatorapi` – CMSIS stub implementation linked to MBED tools for use with simulator
- `mbedsimulatorctr` – MBED tools loaded for simulator
- `mbedcsum` – tool for updating the checksum of the CORTEX M3 exception vector in bin files generated by ARM GCC.
- `HeartRateVisualizer` – component for visualization tool and simulation tool plugin

7. References

Barold, S., Stroobandt, R., & Sinnaeve, A. (2010). *Cardiac Pacemakers and Resynchronization Step by Step: An Illustrated Guide (2nd Edition).* Wiley-Blackwell.

Boston Scientific. (2007). *PACEMAKER System Specification.* Retrieved from http://sqrl.mcmaster.ca/_SQRLDocuments/PACEMAKER.pdf

Eunkyoung Jee, I. L. (2010). Assurance Cases in Model-Driven Development of the Pacemaker Software. *Proceedings of the 4th International Symposium On Leveraging Application of Formal Methods, Verification and Validation (ISoLA 2010), Part II* (pp. 343-356). Heraclion, Crete: Springer.

Ford, S. (n.d.). *ARM MBED Libraries - TextLCD.* Retrieved from http://developer.mbed.org/users/simon/code/TextLCD/

Kelly, T. (1998). *A six-step Method for Developing Arguments in the Goal Structuring Notation (GSN).* UK: York Software Engineering.

Kenney, T. (2008). *The Nuts and Bolts of Cardiac Pacing (2nd Edition).* Wiley-Blackwell.

Leveraging Applications of Formal Methods, Verification, and Validation, 4th International Symposium on Leveraging Applications, ISoLA 2010, Proceedings, Part II. (October 18-21, 2010,). Heraklion, Crete, Greece: Springer.

Ray, A., & Cleaveland, R. (2013). Constructing Safety Assurance Cases for Medical Devices. *Assurance Cases for Software-Intensive Systems (ASSURE), 2013 1st International Workshop on* (pp. 40 - 45). San Francisco, CA : IEEE.

UPPAAL Language Reference. (2014, November). Retrieved from http://www.uppaal.com/index.php?sida=217&rubrik=101

Weinstock, C. B., & Goodenough, J. B. (2009). *Towards an Assurance Case Practice for Medical Devices.* CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST.