Minho University

Maths and Computer Science Course
(LMCC) 5th Year
Internship Report

Validating and Understanding Boston Scientific PACEMAKER Requirements

by

Hugo Daniel dos Santos Macedo

Supervisors:
José Nuno Oliveira (Minho University)
Peter Gorm Larsen (Århus Engineering College)

Braga, 2007

Abstract

The need for rigorous and agile development methods for dependable distributed real-time systems motivated us to use the PACEMAKER system as a case-study to the assess the Vienna Development Method (VDM) capabilities on this field. With this work we intend to show a stepwise approach in the design of a distributed real-time system, that will be valuable in the understanding and validation of the requirements. The abstract VDM-SL model of the PACEMAKER and the less abstract VDM++ sequential, concurrent and distributed real-time models were designed to gain insight of the system and its particular issues and a good framework to validate the next steps towards implementation.

Contents

Acknowledgements

I would like to express my gratitude to Professor Peter Larsen who has given me the opportunity to do this research under his supervision at Århus Engineering College. His scientific and personal guidance and encouragements throughout this internship have been invaluable. He also spent much of his time in the preparation of my visit to IHA and during the stay.

Special thanks to Peter's family for being so supportive and having me with them in the first days I arrived at Århus. Also for the invitations for dinner and the reception tea that was so an essential step in my integration with the other students.

Many thanks to José Nuno Oliveira for being supervisor at my home institution and for providing the opportunity and inspiration during the preceding courses that lead to the realisation of this project.

I have to thank all the people that helped me spending those days away from home. Professor Paulo Azevedo for the company during his visit to the Engineering College. Rosa Albrecht and family for a fabulous Easter. The Danish class crew and teacher Sonja. Maria and friends. . .

I would like to thank Brian Larson for providing valuable domain input on drafts of this work. In addition to Shin Sahara, Marcel Verhoef and John Fitzgerald and all the Overture core members I had the pleasure to work with.

Thanks to Simon for his lecture and later introduction to the academic traditions of Århus University. To Kasper, Henrick and Sander the guys who were also working in the Pervasive Computing lab, specially for Kasper who baked homemade pizzas for us in a attempt to integrate the newcomers.

I would like to thank my entire family - father, mother and brother - for their patience, understanding and support without which I would not have been able to write this report.

Finally this report would not exist without the support, encouragement and love of Ana. She made the effort of being apart for 5 months and every day met virtually, I apologise for being away for far too long.

Chapter 1

Introduction

The verification grand challenge proposed by Hoare[8] set the stage for the program verification community to embark upon a collaborative effort to build verifiable programs. There seemed to be a consensus that a necessary stepping stone to such an effort would be the development of repositories for sharing specifications, models, implementations and benchmarks so that different tools could be combined and compared.

The pacemaker system specification [17] is one of the grand challenges [21]. This challenge is characterised by including system aspects like hardware requirements. In this report, we try to demonstrate how such interdisciplinary requirements can be appropriately modelled and incorporated gradually in a stepwise fashion. In this way a system architect will have guidance in taking informed decisions in the early stages of system development.

A lot of detail needs to be taken into account in the design process of complex and critical embedded systems when alternative architectural solutions are considered. Such complexity can be mastered using this stepwise development approach with VDM-SL and VDM++ models at different levels of abstraction and purpose.

The main advantage of this approach is that, instead of coping with all complex aspects at one time, one can gradually take more and more into account and get stepwise understanding of the system requirements.

1.1 Problem domain

Pumping about 280 litres in one hour the human heart beats over 2.5 billion times and pumps over 200 million litres of blood, in a lifetime [1]. All this work is done by a muscle that is coordinated autonomously.

Heart anatomy review The human heart anatomy is shown in figure 1.1: it is divided into 4 chambers, the upper ones being the atria and the lower ones the ventricles.

Our heart has a natural pacemaker. An electric signal is generated at the Sinus Node which, after some period hits the AV Node that amplifies it.

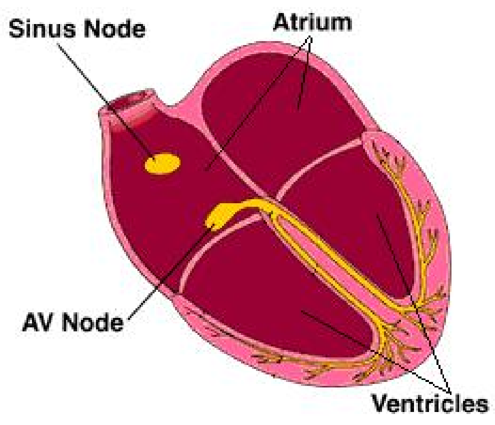


Figure 1.1: Heart anatomy[9]

Bradycardia Bradycardia[20] is a physical condition that normally causes the beats per minute (bpm) rate of the heart to be under the expected (60 bpm). This happens because the natural pacemaker is malfunctioning. In some cases the pulse is generated at the correct time at the Sinus Node, but it's so weak that isn't enough to stimulate the AV Node. In other cases there's no pulse at all. (These are only two of the different natural pacemaker malfunctions causing bradycardia.)

To aid or replace a natural pacemaker an artificial pacemaker is used, which physicians implant on the patient chest and configure according to the needs of each particular patient.

1.2 Artificial pacemaker

An artificial pacemaker (hereafter referred to pacemaker) is a system composed of the following components

- Device: The implanted batteries and controller aka pulse generator.
- Device Controller-Monitor: External unit.
- Leads: One or more wires, normally two, that both sense and discharge electric pulses.
- Accelerometer: Unit inside the device measuring body motion.

The pacemaker is designed to be used in different lead combinations, typically one or two. The chambers where these leads are attached to can differ, the typical usage consisting of one lead attached to the right atrium and another to the right ventricle of the patient, as shown in figure 1.2.

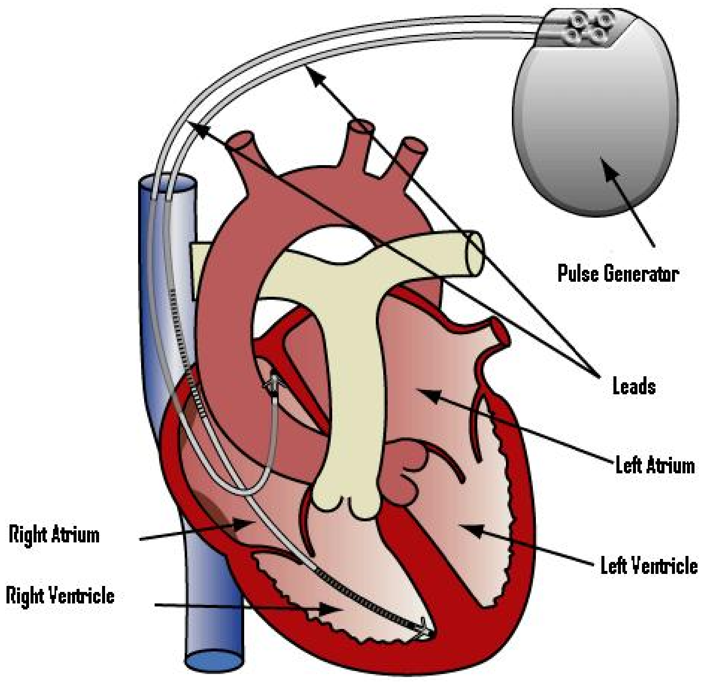


Figure 1.2: A dual lead pacemaker[14]

1.3 Operation mode

Pacemakers not only have different physical configurations but also different operation modes in order to cope with the different kinds of natural pacemaker malfunction.

As explained in [17][11][16][19] the different operation modes of this device are classified using a "Pacemaker Code" consisting in a sequence of letters divided into 4 categories as shown in table 1.1. Valid combinations of letters from each category (the fourth is optional) results in a operating mode.

For example, AOO is a valid mode where the pacemaker paces the Atria without sensing any chamber. AAI is another valid mode where the pacemaker paces and senses the and just discharges a pulse if there's no natural pulse sensed.

Operating codes can also be grouped together in classes using "wildcard notation" where "X" is used to denote any letter. For instance, AXX will denote all modes starting by an A.

I	II	III	IV
Chamber(s) paced	Chamber(s) sensed	Response to sensing	Rate modulation
O = None	O = None	O = None	R = Rate modulation
A = Atria	A = Atria	T = Triggered	
V = Ventricle	V = Ventricle	I = Inhibited	
D = Dual (A+V)	D = Dual (A+V)	D = Tracked	

Table 1.1: Pacemaker operating modes

1.4 Pacemaker vocabulary

The following is a glossary of terminology and acronyms used in the Pacemaker literature:

- **LRL** (**L**ower **R**ate **L**imit) - number of pace pulses delivered per minute in the absence of sensed activity in an interval starting at a paced event.
- **URL** (**U**pper **R**ate **L**imit) - maximum rate at which the paced ventricular rate will track sensed atrial events.
- **MSR** (**M**aximum **S**ensor **R**ate) - maximum pacing rate allowed as a result of accelerometer control.
- **Fixed AV Delay** - programmable period from an atrial event to a ventricular pace.
- **ARP** (**A**trial **R**efractory **P**eriod) - for single chamber atrial modes, this is the time interval following an atrial event during which time atrial events shall not inhibit or trigger pacing.
- **PVARP** (**P**ost **V**entricular **A**trial **R**efractory **P**eriod) - time interval following a ventricular event when an atrial cardiac event shall not

  1. inhibit an atrial pace
  2. trigger a ventricular pace

  available to modes with atrial sensing.
- **ActivityThreshold** - the value the accelerometer sensor output shall exceed before the pacemaker's rate is afected by activity data.
- **ReactionTime** - time required to drive the rate from LRL to MSR with maximum activity.
- **RecoveryTime** - time required to drive the rate from MSR to LRL upon cessation of activity.

Structure of this report

Chapter 2 will contain the background. Chapter 3 will focus on the requirements analysis. Chapter 4, 5 and 6 will describe the sequential, concurrent and distributed real-time models of the requirements. Chapter 7 will describe the model validation. Chapter 8 will provide a summary of the conclusions of this study.

Chapter 2

Background

2.1 Requirements and Model Engineering

Models are everywhere in science and engineering where they are a basis for comprehension and guidance. The same happens in computer systems development, where formal methods play the very important role of recording the understanding of a problem or situation common to all mental models of project participants.

Requirements Engineering (RE) studies the following core activities:

- eliciting requirements
- modelling and analysing requirements
- communicating requirements
- dealing with evolving requirements

In this study we will be mainly interested on requirements analysis and abstract modelling, because we advocate that the use of rigorous models is a good basis to address these activities. With such a model the communication of requirements is clean and not ambiguous.

2.2 Real-time dimension

Real-time systems are systems subject to real-time constraints, e.g. time deadlines from stimulus to system reaction. So the events (observable behaviour of the system) are time dependent. A system is said to be real-time if the total correctness of an operation depends not only on its logical correctness, but also upon the time in which it is performed.

Embedded system such as the PACEMAKER, are called hybrid systems because they are both discrete and continuous systems.

The opposite is a non real-time system where there are no such time constraints; so, triggering or observing a particular event produces always the same result independently of the time the observation takes place.

Traditionally, real-time systems are divided into two classes:

- Hard real-time - the completion of an event after its deadline is considered useless.
- Soft real-time - the completion of an event after its deadline is considered undesirable but not useless.

Most real time systems are regarded as mission critical, thus needing to be designed, validated and verified formally. Example of formal methods applicable to such systems are Timed Raise[13], the Duration Calculus[2] and the VDM VICE[4] extension. Below we address two of these.

2.3 Duration calculus

The Duration Calculus (DC) was born out of the failure of several real-time formalisms in the task of specifying the requirements and design decisions of a gas burner case-study[7]. From the successful application to this and a diverse range of other case studies a methodology and notation for designing real-time systems has emerged[22].

DC is a logical approach to real-time systems design that uses real numbers to model time and functions from time to Boolean values to model the system behaviour. These functions analogous to digital signals are called Boolean states or just states. Let $\mathbb{R}$ be the real numbers and $\mathbb{B} = \{0, 1\}$ in the following example of a state:

$$\begin{aligned} AP &: \mathbb{R} \rightarrow \mathbb{B} \\ AP(t) &= \begin{cases} 1, & \text{if atrium lead is sensing,} \\ 0, & \text{otherwise} \end{cases} \end{aligned}$$

Boolean logic operators can be used to compose atomic states. For instance, we can model the absence of a pulse in the atrium as the negation of the AtriumPulse state.

$$\begin{aligned} NAP &: \mathbb{R} \rightarrow \mathbb{B} \\ NAP(t) &= \neg AP(t) \end{aligned}$$

2.3.1 State Durations

The state duration is the accumulated time in which the state is present in the interval. So the following

$$\int_a^b NAP(t)\,dt$$

is the amount of time in the interval [a,b] where there is no Pulse sensed.

An interesting use of state durations is the expression of an occurrence of a state in an interval [a,b]. For instance to declare the occurrence of the absence of an atrium pulse one can write:

$$NAP[a,b] \equiv \int_a^b NAP(t)dt = (b\text{-}a) \qquad (b\text{-}a > 0)$$

This means that during the interval [a,b] the state NAP occurs.

2.4 Vienna Development Method

The Vienna Development Method and its tool support was recently improved for describe and validating distributed real-time systems, after some attempts to enable VDM with real-time features. An example is the NewThink[15] specification language that mixes a subset VDM-SL with Duration Calculus properties. Later an option was made to improve VDM by itself, extending the language and introducing the system and environment distinction methodology. The language was extended with:

- periodic threads
- duration and cycle statements

2.4.1 Periodic threads

Threads may be either procedural or periodic. A procedural thread is simply a statement, which is executed to completion subject to scheduling and descheduling. A periodic thread has the form

```
periodic (period,jitter,delay,offset) (operation)
```

A periodic thread is started in the same way as a procedural thread, using a `start` statement. The operation stated in the thread declaration is then executed repeatedly, with frequency determined by `period`. If `jitter` is allowed it means that the period is not perfectly periodic, but may vary with the amount of `jitter` before and after the ideal periodic time. If there is a minimal arrival time between two periodic event that is indicated by the `delay` parameter. Finally, in case the periodic occurrence should not start right when the periodic thread is started it is possible to use the `offset` parameter to describe that.

Being more precise we can consider that:

- **period** is a non-negative, non-zero value that describes the length of the time interval between two adjacent events in a strictly periodic event stream (where $jitter = 0$)
- **jitter** is a non-negative value $/j/$ that describes the amount of time variance that is allowed around a single event. We assume that the interval is balanced [$-j, j$]. Note that jitter is allowed to be longer than the period to characterise so-called event bursts.

- **delay** is a non-negative value smaller than the period which is used to denote the minimum inter arrival distance between two adjacent events.
- **offset** is a non-negative value which is used to denote the absolute time value at which the first period of the event stream starts. Note that the first event occurs in the interval $[\mathit{offset}, \mathit{offset} + \mathit{jitter}]$.

2.4.2 Duration and Cycle Statements

A duration statement is a VICE statement that allows a fixed estimate to be placed on the execution time of a particular portion of a model. A duration statement has the form

duration (*time*) *statement*

This means that *statement* is estimated to take *time* time units to execute. This information is used for accumulation of real-time behaviour. Details of this accumulation, and the manner in which time is to be interpreted are described in [6]. Otherwise the duration statement has the exact functional effect of its body *statement*.

A cycles statement is a VDM++ statement that allows a relative estimate to be placed on the execution of a particular portion of a model relative to the `CPU` it is allocated to (deployed). A cycles statement has the form

cycles (*instruction cycles*) *statement*

This means that *statement* is estimated to take *instruction cycles* to execute on any platform. Thus, if a processor with double processor speed is chosen it will take half the time. This information is used for accumulation of real-time behaviour. Details of this accumulation, and the manner in which time is to be interpreted are described in [6]. Otherwise the cycles statement has the exact functional effect of its body *statement*.

2.4.3 The System and the Environment

In order to be able to describe distributed systems, VDM++ includes a notion of a system that describes how different parts of the system modelled are deployed to different Core Processing Units (`CPU`'s) and communication `BUS`'es connecting the `CPU`'s together. Syntactically the system is described exactly like ordinary classes, except that the keyword **system** instead of the keyword **class** is employed.

What is special about a system is that it can make use of special implicitly defined classes called `CPU` and `BUS`. It is not possible to create instances of the system, but instances made of `CPU` and `BUS` will be created at initialisation time. Note that `CPU` and `BUS` cannot be used outside the system definition.

The instances of `CPU` and `BUS` must be made as instance variables and the definition must use constructors. The constructor for the `CPU` class takes two parameters: the first one indicates the primary scheduling policy used for the `CPU` whereas the second parameter provides the capacity of the CPU (indicated as Million Instructions Per Second or MIPS in case one wish to use miliseconds as the time unit). The constructor for the BUS class takes three parameters. The first one indicates the kind of bus, the second one the capacity of the bus (its band width) and finally the third parameter gives a set of CPU instances connected together by the given BUS instance.

The currently supported primary scheduling policies for the CPU are:

<FP>: Fixed Priority

<FCFS>: First Come First Served

The currently supported primary scheduling policies for the BUS are (however, in the first version all of them are treated as FCFS):

<TDMA>: Time Division Multiple Access

<FCFS>: First Come First Served

<CSMACD>: Carrier Sense Multiple Access with Collision Detection

Note that the principles used for describing the system to be developed using the VDM++ technology may be copied for the environment as well. This means that it is possible to use the `system` keyword also for the environment as a whole and thus be able to accurately describe the interaction between the system and its environment. In case this is not done virtual processors (CPU's) and a virtual communication channel (a BUS) are established for each of the environment instances created inside the special class "World" that describe the composition of systems.

2.4.4 Deployments

The CPU class has member operations called `deploy` and `setPriority`. The `deploy` operation takes one parameter which must be an object that is declared as a static instance variable inside the system. The semantics of the deploy operation is that execution of all functionality inside this object will take place on the CPU that it has been deployed to. The `setPriority` operation takes two parameters where the first must be the name of a public operation that has been deployed to the CPU and the second parameter is a natural number. The semantics of the `setPriority` operation is that the given operation is assigned the given priority (the second parameter). This will be used when fixed priority scheduling is used on the given CPU.

2.4.5 Validation conjectures

VDM++ dynamic semantics was extended with the notion of an execution. A trace being a sequence of records containing the set of events occurred and the values of the instance variables in the system model at that time. We can use traces to verify conjectures over traces. These extensions are not completely developed[4]. We define them on chapter 7 anyway.

Chapter 3

Requirements Analysis

The Pacemaker requirements can be divided in two classes: general and mode dependent requirements. General requirements are the ones that apply to the whole system, independently of the mode of operation. Mode dependent requirements only apply to a specific mode and are bounded to the programmable parameters (variables) that must be taken into account while operating in that mode.

In the first section below the general requirements are described and in the following sections some of the modes are analysed.

3.1 General requirements

As referred in the system requirements section on the PACEMAKER System specification[17] that "Each requirement is defined by a sentence containing the word **shall** or by each element in a list of items" thus the approach was to select all of these pieces of information relative to the "device" pacing functionality. The outcome of gathering these requirements is as follows:

- The Pacemaker shall support single and dual chamber rate adaptive pacing [17, p. 13 s. 3.1].
- The device shall output pulses with programmable voltages and widths (atrial and ventricular) which provide electrical stimulation to the heart for pacing[17, p. 16 s. 3.4].
- The atrial and ventricular pacing pulse amplitudes shall be independently programmable[17, p. 16 s. 3.4.1].
- The atrial and ventricular pacing pulse width shall be independently programmable[17, p. 16 s. 3.4.2].
- The following bradycardia operating modes shall be programmable: Off, AOO, AAT, . . . , DDDR [17, p. 16 s. 3.5].

- The device shall have the ability to adjust the cardiac cycle in response to metabolic need as measured from body motion using an accelerometer [17, p. 32].
- The accelerometer shall determine the rate of increse of the pacing rate[17, p. 33 s. 5.7.4].
- The accelerometer shall determine the rate decrease of the pacing rate [17, p. 33 s. 5.7.5].

Keep in mind that for each operating mode a set of mode dependent requirements will be added.

3.2 Programmable parameters requirements

For each programmable parameter the specification document[17] displays the ranging values that can be assigned to and a tolerance interval in a table like Table 3.1.

Parameter	Programmable values	Nominal	Tolerance
Fixed AV Delay	70-300 ms	150 ms	8 ms

Table 3.1: Programmable parameters

Because our purpose is to understand the operation modes we choose the nominal value as a requirement for our model. As an example for the Fixed AV Delay presented in table 3.1 we will derive a requirement stating that the Fixed AV Delay shall be 150 ms.

3.3 AOO mode requirements

The AOO code states

A	Pace the Atria
O	Without sensing the chambers
O	Without response to sensing

meaning that the pacemaker must pace the atria chamber discarding any sensed data from the chambers just regarding the programmable parameters.

Programmable parameters requirements

- LRL1.4 shall be 60 ppm
- URL1.4 shall be 120 ppm

Purpose and abstraction level The purpose is to model and validate the requirements of the Pacemaker AOO mode.

- Atrial Amplitude, Atrial Pulse With and Sensitivity are discarded because they are not relevant for the propose and don't add any understanding of the mode of pacing.
- In this mode the paced/sensed chamber is always the atria so the other chambers are discarded in the model.

VDM-SL module

The model of the requirements start by defining the value of the LRL.

```
module PacemakerAOO

        exports all

definitions
values

        LRL : ℕ = 60
```

The input is a sequence and each element of it corresponds to a time unit abstraction (in this case 1 millisecond).

```
types
        SenseTimeline = Sense*;
        Sense = NONE | PULSE;
```

The output will be a sequence of the reactions to the input that can be either do nothing or discharge a pulse on the Atria.

```
        ReactionTimeline = Reaction*;
        Reaction = NONE | PULSE
```

From LRL to ppm

Considering that the ppm rate is given by the following formula

$$ppm = \frac{numberOfPulses}{timeInMinutes}$$

and converting it to the modelled time unit: (milliseconds)

$$ppm = \frac{60000}{ms}$$

We get

$$ms = \frac{60000}{ppm}$$

The period of the LRL is

$$LRLperiod_{observed} = \frac{numberOfPulses}{numberOfElementsObserved}$$

To compare the observed value to the LRL expressed in ppm, we need to convert it to milliseconds:

$$LRLperiod_{expected} = \frac{60000}{LRL}$$

And because LRL is defined as a minimum, the pacemaker it is modelled as an implicit function stating that the ppm rate is larger or equal the LRL[12].

$$LRLperiod_{observed} <= LRLperiod_{expected}$$

```
functions

    Pacemaker (inp : SenseTimeline) r : ReactionTimeline
    pre len inp > 0
    post let numberOfPulses = len [r (i) | i ∈ inds r · r (i) = PULSE] in
        len r = len inp ∧ numberOfPulses/len r ≤ 60000/LRL
end PacemakerAOO
```

Requirements review

- LRL **Modelled**
- URL is the maximum rate at which the paced ventricular rate will track sensed atrial events. The URL interval is the minimum time between a ventricular event and the next ventricular pace.**Not modelled**[*]

3.4 AAT mode requirements

The AAT code states

A Pace the Atria.

A Sensing the Atria chamber.

T With triggered response to sensing. A sense in the Atria triggers an immediate pace in it.

meaning the pacemaker must pace the atria chamber regarding the valid sensed data from the atria and the programmable parameters.

[*]We discovered this requirement is contradictory and its a minor error in a table from the specification.

Programmable parameter requirements

- LRL1.4 shall be 60 ppm
- URL1.4 shall be 120 ppm
- ARP1.4 shall be 250 ms
- PVARP1.4 shall be 250 ms

Purpose and abstraction level The purpose is to model and validate the requirements of the Pacemaker AAT mode.

- Atrial Amplitude, Atrial Pulse With and Sensivity are discarded because they are not relevant for the propose.
- In this mode the chamber is always the atria so the other chambers are discarded in the model.

VDM-SL module

The model of the requirements start by defining the input of the system as a sequence of senses in a chamber at a given time.

The model of the requirements start by defining the value of the LRL.

```
module PacemakerAAT

    exports all

definitions
values

    LRL : ℕ = 60;

    ARP : ℕ = 250
```

The input is a sequence and of sensed stimuli and the time of it.

```
types
    SenseTimeline = Sense*;
    Sense = NONE | PULSE;
    Time = ℕ₁;
```

The output will be a sequence of the reactions to the input that can be either do nothing or discharge a pulse on the Atria at a particular time.

```
    ReactionTimeline = Reaction*;
    Reaction = NONE | PULSE
```

The pacemaker its modelled as an implicit function stating that the bpm rate is larger or equal the LRL[12] and all the valid stimuli from the input triggers an artificial pulse response. The invalid ones are filtered by the ARP interval.

```
functions

    $Pacemaker\,(inp : SenseTimeline)\;\; r : ReactionTimeline$
    post let $m = \{i \mid i \in$ inds $r \cdot r\,(i) = \mathrm{PULSE}\}$ in
        len $r$ = len $inp \wedge$
        $\forall\, x \in m \cdot ((\exists\, y \in m \cdot y > x) \;\Rightarrow$
                    $(\exists\, z \in m \cdot z \geq x \wedge z - x \leq 60000/LRL)) \wedge$
                   $(\forall\, z \in$ inds $inp \cdot z > x \wedge z - x > ARP)$
end $PacemakerAAT$
```

Requirements review

- LRL **Modeled**
- URL **Not modelled**[†]
- ARP **Modeled**.
- PVARP available to modes with atrial sensing is the time interval following a ventricular event when an atrial cardiac event shall not

  1. Inhibit an atrial pace.
  2. Trigger a ventricular pace.

  Not modelled[‡]

3.5 AAI mode requirements

The AAI code states

A Pace the Atria.

A Sensing the Atria chamber.

I With inhibited response to sensing. A sense in the Atria inhibits a scheduled pace in it.

This means the pacemaker must pace the atria chamber regarding the valid sensed data from the atria and the programmable parameters.

[†] We discovered this requirement is contradictory and its a minor error in a table from the specification.
[‡] We discovered this requirement is contradictory and its a minor error in a table from the specification.

Programmable parameter requirements

- LRL1.4 shall be 60 ppm
- URL1.4 shall be 120 ppm
- ARP1.4 shall be 250 ms
- PVARP1.4 shall be 250 ms

Purpose and abstraction level

The purpose is to model and validate the requirements of the Pacemaker AAI mode.

- Atrial Amplitude, Atrial Pulse With and Sensivity are discarded because they are not relevant for the propose.
- In this mode the chamber is always the atria so the other chambers are discarded in the model.

VDM-SL module

The model of the requirements start by defining the input of the system as a sequence of senses in a chamber at a given time.

The model of the requirements start by defining the value of the LRL.

```
module PacemakerAAI

        exports all

definitions
values

        LRL : ℕ = 60;

        ARP : ℕ = 250
```

The input is a sequence and of sensed stimuli and the time of it.

```
types
        SenseTimeline = Sense*;
        Sense = NONE | PULSE;
```

The output will be a sequence of the reactions to the input that can be either do nothing or discharge a pulse on the Atria at a particular time.

```
        ReactionTimeline = Reaction*;
        Reaction = NONE | PULSE
```

The pacemaker its modelled as an implicit function stating that the bpm rate is larger or equal the LRL[12] and that every valid stimuli from the input triggers an artificial pulse response. The invalid ones are filtered by the ARP interval.

```
functions

    $Pacemaker\,(inp : SenseTimeline)\ r : ReactionTimeline$
    post let $m = \{i \mid i \in \mathsf{inds}\ r \cdot r\,(i) = \mathrm{PULSE}\}$ in
        len $r$ = len $inp \wedge$
        $\forall\, x \in m \cdot ((\exists\, y \in m \cdot y > x) \Rightarrow$
                    $(\exists\, z \in m \cdot z > x \wedge z - x \leq 60000/LRL) \vee$
                    $(\exists\, z \in \mathsf{inds}\ inp \cdot z > x \wedge z - x > ARP \wedge inp\,(z) = \mathrm{PULSE}))$
end $PacemakerAAI$
```

Requirements review

- LRL **Modeled**
- URL **Not modelled**[§]
- ARP **Modeled**.
- PVARP available to modes with atrial sensing is the time interval following a ventricular event when an atrial cardiac event shall not
    1. Inhibit an atrial pace.
    2. Trigger a ventricular pace.

  Not modelled[¶]

[§]We discovered this requirement is contradictory and its a minor error in a table from the specification.
[¶]We discovered this requirement is contradictory and its a minor error in a table from the specification.

3.6 DOO mode requirements

The DOO code states

D Pace the atria and ventricle

O Without sensing the chambers

O Without response to sensing

This means the pacemaker must pace the atria chamber and ventricle discarding any sensed data from the chambers just regarding the programmable parameters.

Programmable parameter requirements

- LRL1.4 shall be 60 ppm.
- URL1.4 shall be 120 ppm.
- Fixed AV1.4 shall be 150 ms.

Purpose and abstraction level Model and validate the requirements of the Pacemaker DOO mode.

- Atrial/Ventricular Amplitude, Pulse Width and Sensitivity are discarded because they are not relevant for us, as they don't add any understanding of the mode of pacing.

VDM-SL module

The model of the requirements start by defining the input of the system as a set of the senses in a chamber at a given time.

```
module PacemakerDOO

        exports all

definitions
types
        Time = ℕ;
        SensedTimeline = (Chamber × Time)-set;
        Chamber = ATRIA | VENTRICLE;
```

Reactions will be an identical set but representing the discharged pulses.

```
        ReactionTimeline = (Chamber × Time)-set
```

The programmable parameters are defined as values.

```
values

    $LRL : \mathbb{N} = 60;$

    $URL : \mathbb{N} = 120;$

    $FixedAV : \mathbb{N} = 150$
```

The Pacemaker system transforms the input set into an output set containing the amount of atrial and ventricular pulses in order to achieve the expected ppm rate. The FixedAV requirement is expressed in the last universal quantifier.

```
functions

    $Pacemaker\ (\mathsf{mk\text{-}}\ (inp, n) : SensedTimeline \times \mathbb{N}_1)\ r : ReactionTimeline$
    post let $nPulsesAtria = \mathsf{card}\ \{i \mid i \in r \cdot i.\#1 = \mathrm{ATRIA}\},$
             $nPulsesVentricle = \mathsf{card}\ \{i \mid i \in r \cdot i.\#1 = \mathrm{VENTRICLE}\}$ in
        $nPulsesAtria/n \geq (LRL/60)/1000 \land$
        $nPulsesVentricle/n \leq (URL/60)/1000 \land$
        $\forall\, \mathsf{mk\text{-}}\ (\mathrm{ATRIA}, ta) \in r \cdot (\exists\, \mathsf{mk\text{-}}\ (\mathrm{VENTRICLE}, tv) \in r \cdot tv = ta + FixedAV)$
end $PacemakerDOO$
```

Requirements review

- LRL **Modeled**.
- URL **Modeled**.
- Fixed AV **Modeled**.

3.7 DDD mode requirements

The **DDD** code states **D** sense both chamber **D** pacing them **D** in a tracked mode. This means the pacemaker must pace the atria and ventricle chambers regarding the sensed data from them. An atrial sense shall cause a ventricular pace after a programmed AV delay, unless a ventricular sense was detected beforehand.

Programmable parameters requirements

- LRL shall be 60 ppm
- URL shall be 120 ppm

- ARP shall be 250 ms
- VRP shall be 320ms
- PVARP shall be 250 ms
- Fixed AV shall be 150 ms

VDM-SL module

The model of the requirements the values used to control this operation mode.

module $PacemakerDDD$

exports all

definitions

values

$LRL : \mathbb{N} = 60;$

$ARP : \mathbb{N} = 250;$

$VRP : \mathbb{N} = 320;$

$PVARP : \mathbb{N} = 250;$

$AVD : \mathbb{N} = 150;$

$VAD : \mathbb{N} = 850$

The input and output of the system is defined as a set of senses in a chamber at a given time. We distinguish between Alarm and Time to increase the readability of the model.

types

$SenseTimeline = (Time \times Chamber)$-set;

$Chamber = \text{ATRIUM} \mid \text{VENTRICLE};$

$Time = \mathbb{Z};$

$Alarm = \mathbb{N};$

$ReactionTimeline = (Time \times Chamber)$-set

Pacemaker initialises the time iteration.

functions

$Pacemaker : Time \times SenseTimeline \rightarrow ReactionTimeline$

$Pacemaker\,(t, s) \triangleq$

$PM\,($mk-$(1, t, s, \{\}, 1000, 0, -ARP, -VRP)).\#1;$

PM iterates the time interval switching between the process decision. Either we have to process a pulse or nothing, passing control to the correspondent handlers.

```
$PM : (Time \times Time \times SenseTimeline \times ReactionTimeline \times Alarm \times Alarm \times Time \times Time) \rightarrow$
        $ReactionTimeline \times Alarm \times Alarm \times Time \times Time$
$PM\,(\mathsf{mk\text{-}}\,(i, t, s, r, AA, VA, LastA, LastV)) \triangleq$
  if $i = t$
  then mk- $(r, AA, VA, LastA, LastV)$
  else if mk- $(i, \mathrm{ATRIUM}) \in s$
      then $PM\,(c\,(i + 1, t, s, SensedAtrium\,(i, r, AA, VA, LastA, LastV)))$
      elseif mk- $(i, \mathrm{VENTRICLE}) \in s$
      then $PM\,(c\,(i + 1, t, s, SensedVentricle\,(i, r, AA, VA, LastA, LastV)))$
      else $PM\,(c\,(i + 1, t, s, SensedNothing\,(i, r, AA, VA, LastA, LastV)))$;
```

When a atrium sense is detected two things can happen, or this is a valid sense it's outside of the ARP refractory period or is invalid and must be discarded (that's why we invoke SensedNothing in this case).

```
$SensedAtrium : Time \times ReactionTimeline \times Alarm \times Alarm \times Time \times Time \rightarrow ReactionTimeline \times Alarm \times Alarm \times Time \times Time$
$SensedAtrium\,(t, r, AA, VA, LastA, LastV) \triangleq$
  if $t - LastA < ARP \vee VA > 0 \vee t - LastA < PVARP$
  then $SensedNothing\,(t, r, AA, VA, LastA, LastV)$
  else mk- $(r, 0, t + AVD, t, LastV)$;
```

Similar to when a ventricle sense is detected, two things can happen: either this is a valid sense outside of the VRP refractory period or is invalid and must be discarded (that's why we invoke SensedNothing in this case).

```
$SensedVentricle : Time \times ReactionTimeline \times Alarm \times Alarm \times Time \times Time \rightarrow ReactionTimeline \times Alarm \times Alarm \times Time \times Time$
$SensedVentricle\,(t, r, AA, VA, LastA, LastV) \triangleq$
  if $t - LastV < VRP$
  then $SensedNothing\,(t, r, AA, VA, LastA, LastV)$
  else mk- $(r, t + VAD, 0, LastA, t)$;
```

SensedNothing controls if the alarms fired.

```
SensedNothing : $Time \times ReactionTimeline \times Alarm \times Alarm \times Time \times Time \rightarrow ReactionTimeline \times Alarm \times Alarm \times Time \times Time$
SensedNothing $(t, r, AA, VA, LastA, LastV) \triangleq$
  if $AA > 0 \wedge t \geq AA$
  then mk- $(r \cup \{$mk- $(t, \mathrm{ATRIUM})\}, 0, t + AVD, t, LastV)$
  elseif $VA > 0 \wedge t \geq VA$
  then mk- $(r \cup \{$mk- $(t, \mathrm{VENTRICLE})\}, t + VAD, 0, LastA, t)$
  else mk- $(r, AA, VA, LastA, LastV)$;
```

The following is an auxiliary function to curry the result of the recursive invocations of Sensed functions:

```
c : $Time \times Time \times SenseTimeline \times (ReactionTimeline \times Alarm \times Alarm \times Time \times Time) \rightarrow$
    $Time \times Time \times SenseTimeline \times ReactionTimeline \times Alarm \times Alarm \times Time \times Time$
c $(i, t, s,$ mk- $(r, a, v, la, lv)) \triangleq$
  mk- $(i, t, s, r, a, v, la, lv)$
```

```
end PacemakerDDD
```

Questions In the specification one can read that a sensed atrial pulse shall cause a tracked ventricle pulse. Is it the same in the case of a paced event in the atrium?

We have a definition for a refractory period during AV interval. Do we have the same for VA interval?

3.8 XXXR modes requirements

The modes ending in R like AOOR require that we adjust the rate for that we had to model the component that will adjust the rate.

Programmable parameters requirements In this mode the following programmable parameters must be taken into account while pacing:

- LRL shall be 60 ppm.
- MSR shall be 120 ppm.
- ActivityThreshold shall be **Med**.
- ResponseFactor shall be 8.
- ReactionTime shall be 30 s.
- RecoveryTime shall be 5 m.

Purpose and abstraction level The rate controller full functionality is not modelled, we represent all the variables needed to control the increase and decrease in rate. But the changes will be instantaneous disregarding the response and recovery time delays. These are requirements for the rate change and not for the operating mode.

The purpose it to understand the different operation modes not how the rate changes, so its logical that if the rate can change on an operation mode for instance AOOR we model that change but in an abstract way.

VDM-SL module

The rate controller will accept as input a sequence of Accelerometer inputs:

```
module RateController

        exports all

definitions
types
        Input = Time -m-> ActivityData;
```

Activity data is mapped to a subset of nat1 as:

- V-LOW 1
- LOW 2
- MED-LOW 3
- MED 4

- MED-HIGH 5

- HIGH 6

- V-HIGH 7

Time is abstracted as a nat.

$$\begin{aligned} & \mathit{Time} = \mathbb{N}_1; \\ & \mathit{ActivityData} = \mathbb{N}_1 \\ & \mathsf{inv}\ a \triangleq a \leq 7; \end{aligned}$$

The response factor is an integer number betwen 1 and 16.

$$\begin{aligned} & \mathit{RF} = \mathbb{N}_1 \\ & \mathsf{inv}\ \mathit{rf} \triangleq \mathit{rf} \leq 16; \end{aligned}$$

The reaction of our system will be a comand to change the rate, in this case we model the ouput as

$$\begin{aligned} & \mathit{Output} = \mathit{Time} \xrightarrow{m} \mathit{PPM}; \\ & \mathit{PPM} = \mathbb{N}_1 \\ & \mathsf{inv}\ \mathit{ppm} \triangleq \mathit{ppm} \geq 30 \wedge \mathit{ppm} \leq 175 \end{aligned}$$

The programmable parameters are declared as values.

values

$\mathit{LRL} : \mathit{PPM} = 60;$

$\mathit{MSR} : \mathit{PPM} = 120;$

$\mathit{Threshold} : \mathit{ActivityData} = 6;$

$\mathit{ReactionTime} : \mathit{Time} = 150;$

$\mathit{ResponseFactor} : \mathit{RF} = 8;$

$\mathit{RecoveryTime} : \mathit{Time} = 5$

Finally the simulation of the rate controller follows as a relation between the reach of the MSR with a exceeding input value of the treshold, and the LRL as a decrease after the reacovery time form the MSR or the normal functioning of the system.

functions

$Simulate\,(inp : Input)\; out : Output$

pre $0 \notin \mathsf{dom}\; inp$

post $\forall\, t \in \mathsf{dom}\; inp \cdot$
$(out\,(t) = MSR \;\Rightarrow\; inp\,(t - ReactionTime) > Threshold \vee out\,(t-1) = MSR) \wedge$
$\forall\, t \in \mathsf{dom}\; inp \setminus \{1\} \cdot$
$(out\,(t) = LRL \;\Rightarrow\; inp\,(t - RecoveryTime) < Threshold \vee out\,(t - 1) = LRL)$

end $RateController$

3.8.1 Requirements review

- LRL **Modelled**.
- MSR **Modelled**.
- ActivityThreshold **Modelled**.
- ResponseFactor **Not understood**.
- ReactionTime **Modelled**.
- RecoveryTime **Modelled**.

Chapter 4

Sequential model

In this chapter the sequential VDM++ model is presented. A description of the static architecture is done in the first sections and then a description in literate programming style of each of the classes composing the model. In the end of each class description a table containing the test coverage for the class is presented, the zones not covered by the tests being printed in red.

4.1 Static architecture

The sequential model of the pacemaker can be divided into two subsystems, one regulating the heart (Figure 4.1) and the other adjusting the rate at which the heart will be regulated (Figure 4.2).

Heart control subsystem

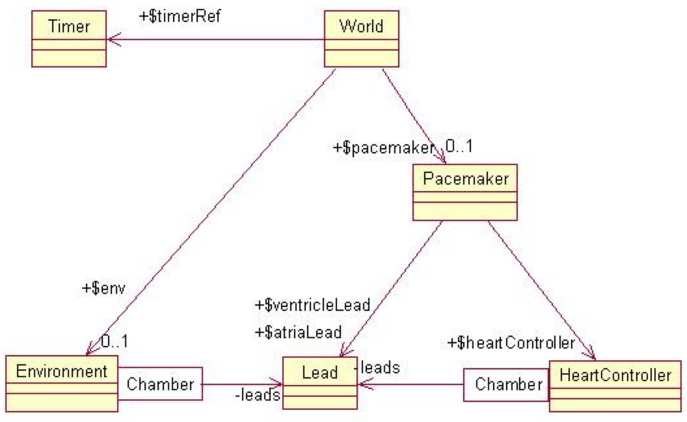


Figure 4.1: Heart control subsystem

World: The main class used to combine system and environment allowing execution of scenarios.

Environment: Used for modelling the environment.

Pacemaker: The system class.

Lead: Models the hardware used to sense and stimulate the heart.

HeartController: The class that regulates and monitors the heart.

Timer: The class used to step time throughout the sequential model.

Rate control subsystem

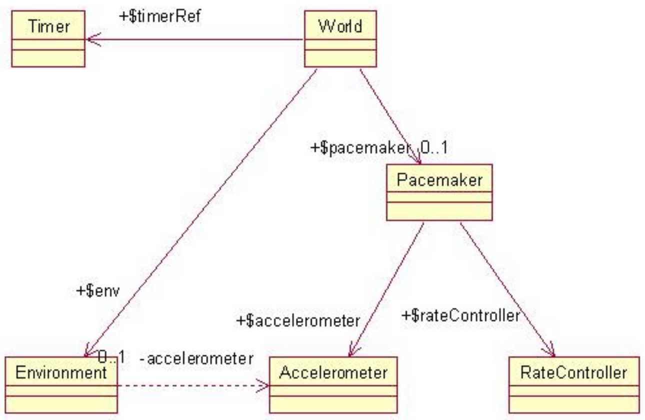


Figure 4.2: Rate control subsystem

RateController: The class controlling the target bpm rate

Accelerometer: Abstracts the accelerometer behaviour

The following classes that make part of the model:

IO: The standard IO library of VDM++

GLOBAL: Superclass providing common type definitions

are not shown.

Model description

The whole model it's divided in system and environment. The environment sends stimuli to the system and receives the outputs from the system. The environment controls the time and execution of the model through a stepping mechanism. At each time step it stimulates the system, increments time and activates the stepping mechanism. An in depth description of this stepping mechanism can be found in[3].

4.2 World class

The World class (as its name tell) is the class modelling the world where the system and environment will co-exist.

```
class World
instance variables
        public static env : [Environment] := nil ;
        public static timerRef : Timer := new Timer ();
```

The World constructor it is responsible for connecting system and environment, and it is invoked with a scenario and a mode to test.

```
operations
public
        World : char* × Mode $\xrightarrow{o}$ World
        World (filename, mode) △
          (   env := new Environment (filename);
              env.addLeadSensor(Pacemaker‘atriaLead) ;
              env.addLeadSensor(Pacemaker‘ventricleLead) ;
              env.addAccelerometer(Pacemaker‘accelerometer) ;
              Pacemaker‘heartController.addLeadPacer(Pacemaker‘atriaLead) ;
              Pacemaker‘heartController.addLeadPacer(Pacemaker‘ventricleLead) ;
              Pacemaker‘heartController.setMode(mode)
          );
```

And Run is the operation that starts a test sequence, printing its value afterwords.

```
public
        Run : () $\xrightarrow{o}$ ()
        Run () △
          (   env.Run() ;
              env.showResult()
          )
end World
```

Test coverage

Test Suite :	tc.info
Class :	World

Name	**#Calls**	**Coverage**
World‘Run	5	√
World‘World	5	√
Total Coverage		**100%**

4.3 Global class

This is the common parent of the other classes where shared knowledge between all the objects is defined.

```
class GLOBAL
```

While poling the leads either a pulse is sensed or nothing. This is modelled by the union type Sense.

```
types
        public Sense = NONE | PULSE;
```

These senses are associated to the chamber where they were produced and again the union type is a good representation of it.

```
        public Chamber = ATRIA | VENTRICLE;
```

Differently the output the accelerometer provides to the heart-controller is defined below consistently to the requirement analysis definition that is a linear order and thus the choice of a subset of the natural numbers.

```
        public ActivityData = ℕ₁
        inv a ≜ a ≤ 7;
```

The heart controller can actuate in two different manners: either do nothing or discharge a pulse. The pulse was refined into two categories to distinguish if the system outputted an totally artificial pulse or a triggered response to sensing.

```
        public Pulse = PULSE | TRI_PULSE;
```

The operation modes are defined as a enumeration of the different quotes corresponding to each mode.

```
        public Mode = AOO | AOOR | AAT | DOO | OFF;
```

Pulses per minute is an instance of nat1

```
        public PPM = ℕ₁
        inv ppm ≜ ppm ≥ 30 ∧ ppm ≤ 175;
```

And to promote readability Time is defined as nat type synonym.

```
        public Time = ℕ
end GLOBAL
```

4.4 Timer class

In the sequential model time abstraction is provided via the Timer class.

```
class Timer is subclass of GLOBAL
```

The instance variable currentTime keeps track of time.

```
instance variables
        currentTime : Time := 0;
```

Time is steping 50 units each time...

```
values
        stepLength : Time = 50
```

...the operation StepTime is called.

```
operations
public
        StepTime : () $\xrightarrow{o}$ ()
        StepTime () $\triangleq$
          currentTime := currentTime + stepLength;
```

And time can be consulted through GetTime.

```
public
        GetTime : () $\xrightarrow{o}$ Time
        GetTime () $\triangleq$
          return currentTime
end Timer
```

Test coverage

Test Suite :	tc.info
Class :	Timer

Name	**#Calls**	**Coverage**
Timer‘GetTime	2005	√
Timer‘StepTime	508	√
Total Coverage		**100%**

4.5 Environment class

The environment class is the class responsible for read a file containing inputs labelled by time and deliver them to the correct system sensor at the right time. It also collects the (re)actions from the system and provides functionality to enable the inspection of them.

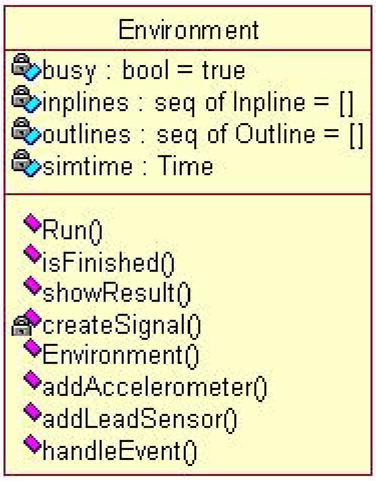


Figure 4.3: Environment class diagram

The starting point is the definition of the types for input and output.

class $Environment$ is subclass of $GLOBAL$

types

public $InputTP = (Time \times Inpline^*)$

inv $inp \triangleq \forall\, line \in$ elems $inp.\#2 \cdot inp.\#1 \geq line.\#4$;

public $Inpline = (Sense \times Chamber \times ActivityData \times Time)$;

public $Outline = (Pulse \times Chamber \times Time)$

Then the IO abstraction is defined as an instance variable of the VDM IO library, and the input and output variables are defined.

instance variables

$io : IO :=$ new $IO\,()$;

$inplines : Inpline^* := []$;

$outlines : Outline^* := []$;

Also a boolean flag indicating that the environment is still sending stimuli to the environment called busy is defined, and simtime represents the amount of time we want to simulate.

$busy : \mathbb{B} :=$ true;

$simtime : Time$;

Then we define the sensors attachment place where the “physician will screw” the leads and where the accelerometer is set up.

```
        $leads : Chamber \xrightarrow{m} Lead := \{\mapsto\};$

$accelerometer : Accelerometer;$
```

The environment constructor reads the test file which name is provided in the variable fname, extracting the amount of time we want to simulate and the inputs in that amount of time.

```
operations
public
        $Environment : \text{char}^* \xrightarrow{o} Environment$
        $Environment\ (fname) \triangleq$
          def mk- (-, mk- $(timeval, input)) = io.freadval[InputTP]\ (fname)$ in
          (    $inplines := input;$
               $simtime := timeval$
          );
```

The operations to connect the environment with the system.

```
public
        $addLeadSensor : Lead \xrightarrow{o} ()$
        $addLeadSensor\ (lsens) \triangleq$
          $leads := leads \dagger \{lsens.getChamber\ () \mapsto lsens\};$
public
        $addAccelerometer : Accelerometer \xrightarrow{o} ()$
        $addAccelerometer\ (acc) \triangleq$
          $accelerometer := acc;$
```

Run is the main operation, starting a session of system stimulation controlling the time and system execution invoking the Step operation on the system components.

```
public
        $Run : () \xrightarrow{o} ()$
        $Run\ () \triangleq$
          (    while $\neg\ (isFinished\ () \wedge$
                         $Pacemaker`heartController.isFinished\ () \wedge$
                         $World`timerRef.GetTime\ () > simtime)$
               do (   $createSignal()$ ;
                      Pacemaker`rateController.$Step()$ ;
                      Pacemaker`heartController.$Step()$ ;
                      World`timerRef.$StepTime()$
                  )
          );
```

The createSignal operation delivers stimuli to the different components, choosing between all the inputs the ones that should be delivered at the current time.

```
private
        $createSignal : () \xrightarrow{o} ()$
        $createSignal\,() \triangleq$
          (   if len $inplines > 0$
              then (   dcl $curtime : Time := World`timerRef.GetTime\,()$,
                           $done : \mathbb{B} :=$ false;
                       while $\neg\, done$
                       do let mk- $(sensed, chamber, accinfo, stime) =$ hd $inplines$ in
                          if $stime \leq curtime$
                          then (   $leads$   (chamber )  $.stimulate(sensed)$ ;
                                   $accelerometer.stimulate(accinfo)$ ;
                                   $inplines :=$ tl $inplines$;
                                   $done :=$ len $inplines = 0$
                               )
                          else $done :=$ true
                   ) ;
              if len $inplines = 0$
              then $busy :=$ false
          );
```

The (re)actions from the pacemaker are delivered to the environment through the handleEvent operation which updates the collection of outputs.

```
public
        $handleEvent : Pulse \times Chamber \times Time \xrightarrow{o} ()$
        $handleEvent\,(p, c, t) \triangleq$
          $outlines := outlines \curvearrowright [$mk-$ (p, c, t)]$;
```

ShowResult is an operation used to inspect the output collection.

```
public
        $showResult : () \xrightarrow{o} ()$
        $showResult\,() \triangleq$
          def - $= io.writeval[Outline^*]\,(outlines)$ in
          skip;
```

The last operation called isFinished its used to have a mean of knowing if the environment finished the stimulation of the system phase.

```
public
        $isFinished : () \xrightarrow{o} \mathbb{B}$
        $isFinished\ () \triangleq$
            return $inplines = [] \land \neg\ busy$
end $Environment$
```

Test coverage

Test Suite :	tc.info
Class :	Environment

Name	**#Calls**	**Coverage**
Environment‘Run	5	$\surd$
Environment‘isFinished	513	$\surd$
Environment‘showResult	5	$\surd$
Environment‘Environment	5	$\surd$
Environment‘handleEvent	25	$\surd$
Environment‘createSignal	508	$\surd$
Environment‘addLeadSensor	10	$\surd$
Environment‘addAccelerometer	5	$\surd$
Total Coverage		**100%**

4.6 Pacemaker class

This class models the pacemaker system and will contain the components of it as static variables the first ones are the atrial and ventricular leads responsible for discharge and sense electrical pulses.

```
class $Pacemaker$
instance variables
        public static $atriaLead : Lead :=$ new $Lead\ (\mathrm{ATRIA});$
        public static $ventricleLead : Lead :=$ new $Lead\ (\mathrm{VENTRICLE});$
```

The Accelerometer component and the RateContoller.

```
        public static $accelerometer : Accelerometer :=$ new $Accelerometer\ ();$
        public static $rateController : RateController :=$ new $RateController\ ();$
```

The final declaration is the HeartController component.

```
        public static $heartController : HeartController$ := new $HeartController$ ();

end $Pacemaker$
```

4.7 Lead class

The Lead models a Pacemaker lead that senses and discharges pulses from and to the heart.

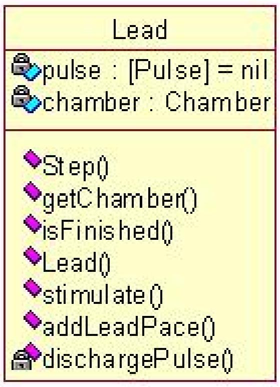


Figure 4.4: Lead class diagram

Each lead its attached to a specific chamber and this is captured using the instance variable chamber.

```
class $Lead$ is subclass of $GLOBAL$
instance variables
        private $chamber : Chamber$;
        private $scheduledPulse : [(Time \times Pulse)]$;
```

Also a scheduledPulse variable keeps track of a possible pulse that was scheduled by the HeartContorller.

```
operations
public
        $Lead : Chamber \xrightarrow{o} Lead$
        $Lead\,(chm) \triangleq$
          (     $chamber := chm$;
                $scheduledPulse :=$ nil
          );
```

getChamber is an auxiliar operation that inspect the chamber where this lead is attached to.

```
public
        $getChamber : () \xrightarrow{o} Chamber$
        $getChamber\ () \triangleq$
            return $chamber$;
```

Whenever there's an electrical pulse in the chamber corresponding to this lead the environment will call the following stimulate operation and the lead will transmit it immediately to the HeartController.

```
public
        $stimulate : Sense \xrightarrow{o} ()$
        $stimulate\ (s) \triangleq$ Pacemaker`heartController.
            $sensorNotify(s, chamber)$ ;
```

The stepping mechanism in this class it's to check each time unit if there is a scheduled pulse and that is made by followPlan.

```
public
        $Step : () \xrightarrow{o} ()$
        $Step\ () \triangleq$
            $followPlan()$ ;
```

A lead its in a finish state if there's no scheduledPulse.

```
public
        $isFinished : () \xrightarrow{o} \mathbb{B}$
        $isFinished\ () \triangleq$
            return $scheduledPulse =$ nil ;
```

The following operation its used by the HeartController when a pulse should be delivered. Its logic is: if a pulse is to be delivered right now does it, otherwise schedules it. Because there will be always just one scheduled pulse a precondition is set accordingly.

```
public
        $addLeadPace : Pulse \times Time \xrightarrow{o} ()$
        $addLeadPace\ (p, t) \triangleq$
            if $t \leq World`timerRef.GetTime\ ()$
            then $dischargePulse(p)$
            else (    $scheduledPulse :=$ mk- $(t, p)$;
                      return
                 )
        pre  $t > World`timerRef.GetTime\ () \Rightarrow scheduledPulse =$ nil  ;
```

Then the private function dischargePulse delivers the pulse to the environment.

```
private
        $dischargePulse : Pulse \overset{o}{\rightarrow} ()$
        $dischargePulse\,(p) \triangleq$ World‘env.
          $handleEvent(p, chamber, World‘timerRef.GetTime\,())$ ;
```

The followPlan is invoked each Step in order to discharge the pulse that eventually was scheduled.

```
private
        $followPlan : () \overset{o}{\rightarrow} ()$
        $followPlan\,() \triangleq$
          (    dcl $curTime : Time := World‘timerRef.GetTime\,()$;
               if $scheduledPulse \neq$ nil
               then if $(curTime \geq scheduledPulse.\#1)$
                    then (   $dischargePulse(scheduledPulse.\#2)$ ;
                             $scheduledPulse :=$ nil
                         )
          )
end $Lead$
```

Test coverage

Test Suite :	tc.info
Class :	Lead

Name	**#Calls**	**Coverage**
Lead‘Lead	10	√
Lead‘Step	1016	√
Lead‘stimulate	20	√
Lead‘followPlan	1016	√
Lead‘getChamber	20	√
Lead‘isFinished	260	√
Lead‘addLeadPace	25	√
Lead‘dischargePulse	25	√
Total Coverage		**100%**

Pulse interval

As will be clear after the description of the HeartController class the central concern of the pacemaker is to keep the stimulating the different chambers at a certain ppm rate. Our approach to achieve a certain ppm rate was to define the interval between pulses and depending on the mode discharge pulses each time interval. For instance if a heart should have at least 60 atrial ppm - supposing the functioning mode is AOO - we know that this means 1 pps so the time interval between pulses will be 1 second.

4.8 HeartController class

This is the core class monitoring and regulating the heart.

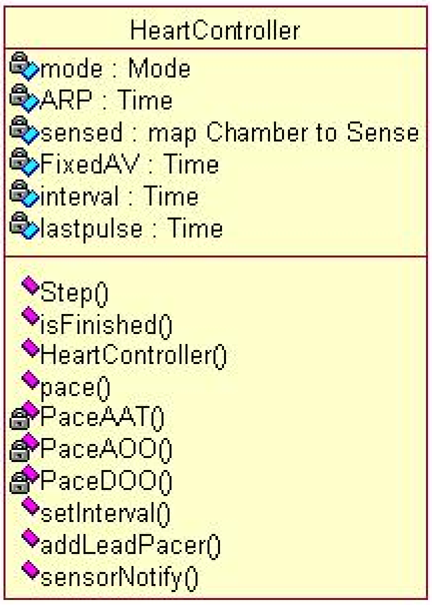


Figure 4.5: HeartController class diagram

class $HeartController$ is subclass of $GLOBAL$

instance variables

$leads : Chamber \xrightarrow{m} Lead;$

$sensed : Chamber \xrightarrow{m} Sense;$

$mode : Mode;$

$FixedAV : Time;$

$lastPulse : Time;$

$ARP : Time;$

$interval : Time;$

In a few words these variables mean:

- leads: the leads attached to the pacemaker
- sensed: keeps track of the last sense for each chamber
- mode: current operation mode

- lastPulse: stores the time of the last atrial pace event
- ARP : the ARP parameter
- interval : is the interval between pulses to achieve the expected rate

The following operation is the constructor with the default values for the instance variables.

```
operations
public
        $HeartController : () \xrightarrow{o} HeartController$
        $HeartController\ () \triangleq$
          (     $leads := \{\mapsto\};$
                $sensed := \{\mapsto\};$
                $mode :=$ OFF;
                $FixedAV := 150;$
                $lastPulse := 0;$
                $ARP := 250;$
                $interval := Pacemaker`rateController.getInterval\ ()$
          );
```

The addLeadPacer operation its used to attach a lead to the Pacemaker.

```
public
        $addLeadPacer : Lead \xrightarrow{o} ()$
        $addLeadPacer\ (lead) \triangleq$
          $leads := leads \dagger \{lead.getChamber\ () \mapsto lead\};$
```

The right pacing mode its chosen by pace that also refreshes the sensed map.

```
public
        $pace : () \xrightarrow{o} ()$
        $pace\ () \triangleq$
          (     cases  $mode$:
                  AOO $\rightarrow PaceAOO()$ ,
                  AAT $\rightarrow PaceAAT()$ ,
                  DOO $\rightarrow PaceDOO()$ ,
                  OFF $\rightarrow$ skip,
                  others $\rightarrow$ error
                end;
                $sensed := \{\mapsto\}$
          );
```

And each time step we pace and after it we call step in the leads

```
public
        $Step : () \overset{o}{\rightarrow} ()$
        $Step\ () \triangleq$
          (    $pace()$ ;
               for all $key \in$ dom $leads$
               do $leads$    (key )  .$Step()$
          );
```

Pace in the AOO mode follows from the VDM-SL specification discarding all the sensed activity and pacing each time interval.

```
private
        $PaceAOO : () \overset{o}{\rightarrow} ()$
        $PaceAOO\ () \triangleq$
          let $curTime : Time = World`timerRef.GetTime\ ()$ in
          if $(interval + lastPulse \leq curTime)$
          then (    $lastPulse := curTime$;
                    $leads$    (ATRIA )  .$addLeadPace$(PULSE, $curTime$)
               )
          else skip
        pre  ATRIA $\in$ dom $leads$ ;
```

AAT mode provides triggered responses and discards refractory senses.

```
private
        $PaceAAT : () \overset{o}{\rightarrow} ()$
        $PaceAAT\ () \triangleq$
          let $curTime : Time = World`timerRef.GetTime\ ()$ in
          if ATRIA $\in$ dom $sensed \wedge sensed$ (ATRIA) = PULSE
          then if $curTime - lastPulse \leq ARP$
               then skip
               else (    $lastPulse := curTime$;
                         $leads$    (ATRIA )  .$addLeadPace$(TRI_PULSE, $curTime$)
                    )
          elseif $(interval + lastPulse \leq curTime)$
          then (    $lastPulse := curTime$;
                    $leads$    (ATRIA )  .$addLeadPace$(PULSE, $curTime$)
               )
          else skip
        pre  ATRIA $\in$ dom $leads$ ;
```

Pace in the DOO mode discards all the sensed activity and stimulates both atria and ventricle with a fixed delay each time interval.

```
private
        $PaceDOO : () \xrightarrow{o} ()$
        $PaceDOO\ () \triangleq$
          let $curTime : Time = World`timerRef.GetTime\ ()$ in
          (   if $(interval + lastPulse \leq curTime)$
              then (   $lastPulse := curTime;$
                       $leads\ (\mathrm{ATRIA})\ .addLeadPace(\mathrm{PULSE}, curTime)$ ;
                       $leads\ (\mathrm{VENTRICLE})\ .addLeadPace(\mathrm{PULSE}, curTime +$
$FixedAV)$
                   )
              else skip
          )
        pre $\{\mathrm{ATRIA}, \mathrm{VENTRICLE}\} \subseteq$ dom $leads$ ;
```

Is finished depends on the leads isFinished.

```
public
        $isFinished : () \xrightarrow{o} \mathbb{B}$
        $isFinished\ () \triangleq$
          return $\forall\, key \in$ dom $leads \cdot$
                    $leads\ (key).isFinished\ ();$
```

This is the lead handler that its called each time a pulse is sensed.

```
public
        $sensorNotify : Sense \times Chamber \xrightarrow{o} ()$
        $sensorNotify\ (s, c) \triangleq$
          (   $sensed := sensed \dagger \{c \mapsto s\}$
          );
```

To switch the operating mode one should use.

```
public
        $setMode : Mode \xrightarrow{o} ()$
        $setMode\ (m) \triangleq$
          (   $mode := m$
          );
```

And setInterval is the operation used by the RateController to adjust the interval between pulses.

```
public
        $setInterval : Time \xrightarrow{o} ()$
        $setInterval\ (t) \triangleq$
          $interval := t$
end $HeartController$
```

Test coverage

Test Suite :	tc.info
Class :	HeartController

Name	**#Calls**	**Coverage**
HeartController‘Step	508	√
HeartController‘pace	508	92%
HeartController‘PaceAAT	202	√
HeartController‘PaceAOO	101	√
HeartController‘PaceDOO	104	√
HeartController‘setMode	5	√
HeartController‘isFinished	133	√
HeartController‘setInterval	20	√
HeartController‘addLeadPacer	10	√
HeartController‘sensorNotify	20	√
HeartController‘HeartController	5	√
Total Coverage		**99%**

4.9 Accelerometer

This class models the accelerometer physical device containing one only operation called stimulate which will hand in the stimulus to the rateController that process the information from this sensor.

```
class $Accelerometer$ is subclass of $GLOBAL$
operations
public
        $stimulate : ActivityData \xrightarrow{o} ()$
        $stimulate\ (a) \triangleq$ Pacemaker‘rateController.
          $stimulate(a)$
end $Accelerometer$
```

Test coverage

Test Suite :	tc.info
Class :	Accelerometer

Name	**#Calls**	**Coverage**
Accelerometer‘stimulate	20	√
Total Coverage		**100%**

4.10 RateController class

The RateController is the class that models the rate adaptation control.

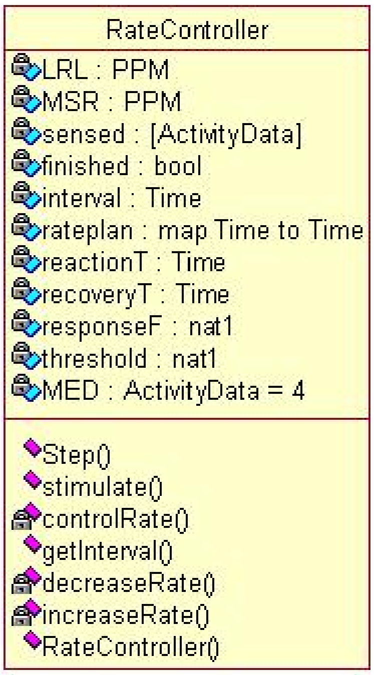


Figure 4.6: RateController class diagram

To accomplish that it has the sensed instance variable where the last accelerometer value is stored and the interval corresponds to the actual rate interval.

class $RateController$ is subclass of $GLOBAL$
instance variables
$\quad sensed : [ActivityData];$
$\quad interval : Time;$
$\quad finished : \mathbb{B};$

The other variables are used to control the value of the interval with an invariant restricting the values to the ranges defined in[17].

```
        $LRL : PPM$;
        $MSR : PPM$;
        $threshold : \mathbb{N}_1$;
        $reactionT : Time$;
        $recoveryT : Time$;
        $responseF : \mathbb{N}_1$;
        inv $threshold < 8 \wedge$
            $reactionT \in \{10, \ldots, 50\} \wedge$
            $recoveryT \in \{2, \ldots, 16\} \wedge$
            $responseF \leq 16$
```

The Constructor initialises the instance variables with the default values as consulted in[17];

```
operations
public
        $RateController : () \xrightarrow{o} RateController$
        $RateController\ () \triangleq$
          (   $LRL := 60$;
              $MSR := 120$;
              $threshold := MED$;
              $reactionT := 10$;
              $recoveryT := 2$;
              $responseF := 8$;
              $sensed :=$ nil ;
              $interval := 1/((LRL/60)/1000)$;
              $finished :=$ false
          );
```

This is the method that should be used to inspect which is the actual value of the maximum interval between atrial events in order to achieve a bpm rate above or equal the LRL defined.

```
public
        $getInterval : () \xrightarrow{o} Time$
        $getInterval\ () \triangleq$
          return $interval$;
```

Each time step the controlRate operation will be invoked if there was some input from the accelerometer.

```
public
        $Step : () \xrightarrow{o} ()$
        $Step\ () \triangleq$
          if $sensed \neq$ nil
          then $controlRate()$ ;
```

The control of the rate is done regarding a threshold.

```
private
        $controlRate : () \xrightarrow{o} ()$
        $controlRate\ () \triangleq$
          (    if $sensed > threshold$
               then $increaseRate()$
               elseif $sensed < threshold$
               then $decreaseRate()$
               else skip;
               $sensed :=$ nil
          );
```

Stimulate is the handler the accelerometer will call to deliver input.

```
public
        $stimulate : ActivityData \xrightarrow{o} ()$
        $stimulate\ (ad) \triangleq$
          $sensed := ad;$
```

These are the operations modelling the change in rate, at this modelling stage the increase as mentioned above is done immediately.

```
private
        $increaseRate : () \xrightarrow{o} ()$
        $increaseRate\ () \triangleq$
          (    $interval := 1/((MSR/60)/1000);$
               Pacemaker‘heartController.$setInterval(interval)$
          );
```

Decreasing the rate its also instantaneously.

```
private
        $decreaseRate : () \xrightarrow{o} ()$
        $decreaseRate\ () \triangleq$
          (    $interval := 1/((LRL/60)/1000);$
               Pacemaker‘heartController.$setInterval(interval)$
          )
```

To improve readability the accelerometer outputs (ActivityData) are defined as values.

```
values
        V-LOW : ActivityData = 1;
        LOW : ActivityData = 2;
        MED-LOW : ActivityData = 3;
        MED : ActivityData = 4;
        MED-HIGH : ActivityData = 5;
        HIGH : ActivityData = 6;
        V-HIGH : ActivityData = 7
end RateController
```

Test coverage

Test Suite :	tc.info
Class :	RateController

Name	**#Calls**	**Coverage**
RateController‘Step	508	√
RateController‘stimulate	20	√
RateController‘controlRate	20	84%
RateController‘getInterval	5	√
RateController‘decreaseRate	20	√
RateController‘increaseRate	0	0%
RateController‘RateController	5	√
Total Coverage		**79%**

Chapter 5

Concurrent model

The concurrent model of the pacemaker has the same structure as the sequential one. Following the guidelines[3] the time abstraction is changed and the stepping mechanism is substituted by threading and concurrent synchronisation. In this chapter the relevant changes to the sequential model are shown and the new time abstraction mechanism explained.

The time abstraction mechanism

What this section intends to synthetically explain is the Wait Notify mechanism that described in detail in [3].

In the concurrent model the stepping mechanism is substituted by a threaded model where the execution flow executes in parallel using a synchronisation mechanism based in a Wait Notify pattern. So each time step the environment thread blocks itself calling **Wait** and allows the other threads to execute, after execution each thread **Notify** the environment and block.

5.1 Environment class highlights

The isFinished operation is redefined and blocks until the simulation is finished because of a permission predicate defined below.

```
public
        $isFinished : () \xrightarrow{o} ()$
        $isFinished\,() \triangleq$
          skip
```

The Step function it's substituted by a thread definition as mentioned. This thread will execute the createSignal operation and through NotifyAndIncTime will increment the clock and notify the system threads allowing them to execute after it blocks itself via the

WaitRelative operation.

```
thread

        (
start
(new ClockTick     (threadid )  ) ;
            while true
            do (    if busy
                    then createSignal() ;
                    if World`timerRef.GetTime () ≥ simtime
                    then (     Pacemaker`heartController.finish() ;
                               Pacemaker`rateController.finish()
                         ) ;
                    World`timerRef.NotifyAndIncTime() ;
                    World`timerRef.WaitRelative(0)
               )
        )
```

Because of the concurrency introduction handleEvent and showResult that both manipulate the outlines instance variable are made mutex.

Notice how the operation isFinished is released just when the Environment is not busy (no more inputs to be delivered) and the whole amount of simulation time is equal or above the desired.

```
sync
        mutex(handleEvent, showResult);
        per isFinished  ⇒  ¬ busy ∧ World`timerRef.GetTime () ≥ simtime
end Environment
```

5.2 HeartController class highlights

The changes in this class are the Step operation now transformed in a thread. This thread executes every 50 units time period, calls pace and then blocks another 50 by calling Wait-

Relative.

```
thread

        (   while true
            do (   World‘timerRef.$WaitRelative(50)$ ;
                   $pace()$
               )
        )
sync
        per $isFinished \Rightarrow sensed = \{\mapsto\} \land finished$;
        mutex($sensorNotify, pace$)
```

Because of the concurrency introduction the operations manipulating the sensed variables are constrained by the mutex predicate.

Because isFinished was redefined as in the environment to a empty skip operation a permission predicate is defined characterising the finished state of this class i.e. there should be no sensed activity to process and the simulation time was reached as signalled by the finished flag.

5.3 Lead class highlights

To the lead class the changes are the same a thread definition substitutes the Step function.

```
thread

        while true
        do (   World‘timerRef.$WaitRelative(5)$ ;
               $followPlan()$
           )
```

And synchronization permission predicates are introduced.

```
sync
        per $followPlan \Rightarrow scheduledPulse \neq$ nil ;
        per $isFinished \Rightarrow scheduledPulse =$ nil ;
        mutex($addLeadPace$);
        mutex($dischargePulse$)
```

5.4 RateController class highlights

The usual Step by thread substitution is done in the RateController class.

```
thread

        while true
        do $controlRate()$
```

And syncronization is added.

```
sync
        mutex($stimulate$);
        mutex($increaseRate$, $decreaseRate$, $getInterval$);
        per $isFinished \Rightarrow finished$;
        per $controlRate \Rightarrow sensed \neq$ nil
```

5.5 ClockTick class

ClockTick is a class containing a thread that its used as a spark plug for the Environment. Each time increment the Environment sleeps and gives a running chance to the other threads. So if all the other threads are blocked this Clock tick will be always ready to wake up the Environment.

The class is created maintains as an instance variable the thread id **tid** of the thread that it's supposed to be notified each clock tick.

```
class $ClockTick$ is subclass of $GLOBAL$
instance variables
        $tid : \mathbb{N} := -1$;
```

The constructor recieves the thread id.

```
operations
public
        $ClockTick : \mathbb{N} \xrightarrow{o} ClockTick$
        $ClockTick\ (t) \triangleq$
          $tid := t$
```

And the thread definition follows first the thread in this case the Environment is notified and then ClockTick blocks itself until the time is incremented.

```
thread

        while true
        do (    World‘timerRef.$NotifyThread(tid)$ ;
                World‘timerRef.$WaitRelative(1)$
           )
end $ClockTick$
```

Chapter 6

Distributed real-time model

The DR-T model of the pacemaker has the same structure as the sequential and the concurrent model. The time abstraction is now provided by the toolbox due to the VICE extensions. The system class is changed to a representation of the physical architecture. In this chapter the relevant changes to the previous models are highlighted.

6.1 Pacemaker class

This is the pacemaker system class. It contain the components that must be defined as static variables.

```
system Pacemaker
instance variables
        public static atriaLead : Lead := new Lead (ATRIA);
        public static ventricleLead : Lead := new Lead (VENTRICLE);
        public static accelerometer : Accelerometer := new Accelerometer ();
        public static rateController : RateController := new RateController ();
        public static heartController : HeartController := new HeartController ();
        public static dcu : DCU := new DCU ();
```

In the system class we also define the physical architecture. The following lines show how to define the architectural components, in this case two cpu's indicating inside each constructor the scheduling algorithm and capacity.

```
        cpu1 : CPU := new CPU (FP, 1);
        cpu2 : CPU := new CPU (FCFS, 1000000);
```

To define the communication topology, we define a bus object linking the CPUs with a certain bandwidth and the chosen network control protocol in this case FCFS - first come first served.

```
$bus1 : BUS$ := new $BUS$ (FCFS, 1000000, $\{cpu1, cpu2\}$);
```

The final element in this system class is the constructor that shall deploy the different functionality across the different resources.

```
operations
public
        $Pacemaker : () \xrightarrow{o} Pacemaker$
        $Pacemaker\ () \triangleq$
          (    $cpu1.deploy(atriaLead)$ ;
               $cpu1.deploy(ventricleLead)$ ;
               $cpu1.deploy(accelerometer)$ ;
               $cpu1.deploy(rateController)$ ;
               $cpu1.deploy(heartController)$ ;
               $cpu1.setPriority(HeartController`pace, 3)$ ;
               $cpu1.setPriority(RateController`adjustRate, 1)$ ;
               $cpu2.deploy(dcu)$
          )
end $Pacemaker$
```

6.2 Environment class

createSignal delivers stimuli to the different components but now it makes use of the built-in time.

```
private
        $createSignal : () \xrightarrow{o} ()$
        $createSignal\ () \triangleq$
          (    if len $inplines > 0$
               then (   dcl $curtime : Time$ := time,
                            $done : \mathbb{B}$ := false;
                        while $\neg\ done$
                        do ...
                    ) ;
               if len $inplines = 0$
               then $busy$ := false
          );
```

And the thread is transformed into a periodic thread.

```
thread

        periodic (1000)(createSignal)
```

6.3 HeartController class highlights

The only change is the concurrent thread now becoming a periodic thread invoking the pace operation.

```
thread

        periodic (200)(pace)
```

6.4 Lead class

The Lead class remains equal unless the definition of the dischargePulse where now we can use a duration VICE primitive to explicit model the width of a pulse. The lead class abstracts a:

- Wire
- Discharging electricity
- Requirement: 0.4 ms width

```
private
        dischargePulse : Pulse $\overset{o}{\rightarrow}$ ()
        dischargePulse (p) $\triangleq$
duration
(4) World‘env.
          handleEvent(p, chamber, time) ;
```

The thread blocking through WaitRelative is now defined as a “beautiful” periodic

```
thread

        periodic (50)(followPlan)
```

Chapter 7

Model validation

To validate our models we will use testing and to the particular case of the distributed real-time model we will analyse the execution traces provided by the VDMTools using the showtrace[3] tool and elaborating some validation conjectures over that traces following the work from[4].

7.1 Model testing

The testing is done using a test scenario methodology and a simple script to exercise the models and collect the test coverage.

```
tcov read tc.info
p new World("scenario.arg",<MODE>).Run()
tcov write tc.info
```

To verify the outputs we could have used the post condition derived functions from the VDM-SL modules where the specification of the output of the different modes is done, but that was not accomplished in this work and a manual verification of the results was done.

7.1.1 Scenarios

To each scenario there's an input file containing a pair time of simulation desired and a sequence of stimuli to the system.

Broken heart

Story	The natural pacemaker failed and no natural stimulation is sensed.
Motivation	We want to see the pacemaker substituting the natural one.
Expected	Total substitution of the natural pacemaker through artificial pulses.

```
mk_(5000,[])
```

Good heart

Story	The natural pacemaker is working as it should.
Motivation	Test the XXT modes and assure the rate is set by the natural pacemaker.
Expected	We want to see the pacemaker coping with the natural pacemaker.

```
mk_(5000,
 [mk_(<PULSE>,<ATRIA>,2,1000),mk_(<PULSE>,<VENTRICLE>,2,1150),
  mk_(<PULSE>,<ATRIA>,2,2000),mk_(<PULSE>,<VENTRICLE>,2,2150),
  mk_(<PULSE>,<ATRIA>,2,3000),mk_(<PULSE>,<VENTRICLE>,2,3150),
  mk_(<PULSE>,<ATRIA>,2,4000),mk_(<PULSE>,<VENTRICLE>,2,4150),
  mk_(<PULSE>,<ATRIA>,2,5000)])
```

Sometimes heart

Story	The natural pacemaker sometimes fail one atria or ventricle stimulation.
Motivation	Test the XXT modes and assure the rate is set by the natural pacemaker.
Expected	Atrial and ventricle fail fixed with artificial pulses.

```
mk_(5000,
 [mk_(<PULSE>,<ATRIA>,4,900), mk_(<PULSE>,<VENTRICLE>,4,1150),
                              mk_(<PULSE>,<VENTRICLE>,4,2150),
  mk_(<PULSE>,<ATRIA>,4,2900),mk_(<PULSE>,<VENTRICLE>,4,4150),
  mk_(<PULSE>,<ATRIA>,4,4900)])
```

Double heart

Story	The sensors detect refractory pulses in the atria.
Motivation	Test if the AAT modes is taking into account the ARP parameter.
Expected	Discard of extra sensed activity.

```
mk_(5000,
 [mk_(<PULSE>,<ATRIA>,4,900), mk_(<PULSE>,<ATRIA>,4,950),
  mk_(<PULSE>,<ATIRA>,4,1900),mk_(<PULSE>,<ATRIA>,4,1940)])
```

7.1.2 Test results

A pacemaker operating in the Off mode in the good heart scenario.

```
new World("scenarioGoodHeart.arg",<OFF>).Run()
```

returns the output an empty sequence of (re)actions.

```
[]
```

Pacemaker operating in the DOO mode in the broken heart scenario.

```
new World("scenarioBrokenHeart.arg",<DOO>).Run()
```

The output is as expected the pacing of both chambers achieving the LRL rate 60 bpm with +/- 8 ms tolerance thus one pulse each 1000 ms and a fixed delay of 150 ms +/- 8 ms between the atrial and ventricular pulse.

```
[mk_(<PULSE>,<ATRIA>,1006),mk_(<PULSE>,<VENTRICLE>,1155),
 mk_(<PULSE>,<ATRIA>,2008),mk_(<PULSE>,<VENTRICLE>,2161),
 mk_(<PULSE>,<ATRIA>,3011),mk_(<PULSE>,<VENTRICLE>,3160),
 mk_(<PULSE>,<ATRIA>,4013),mk_(<PULSE>,<VENTRICLE>,4163),
 mk_(<PULSE>,<ATRIA>,5014)]
```

A simulation of the AOO pacemaker with the sometimes heart scenario

```
new World("scenarioSometimesHeart.arg",<AOO>).Run()
```

results in the stimulation of the atria at the expected rate ignoring the natural pacemaker pulses.

```
[mk_(<PULSE>,<ATRIA>,1006),
 mk_(<PULSE>,<ATRIA>,2008),
 mk_(<PULSE>,<ATRIA>,3012),
 mk_(<PULSE>,<ATRIA>,4013),
 mk_(<PULSE>,<ATRIA>,5017)]
```

To finalise a simulation of the AAT pacemaker with the double heart scenario

```
new World("scenarioDoubleHeart.arg",<AAT>).Run()
```

results in the stimulation of the atria at the expected rate but with the triggered responses reaction to the natural pacemaker pulses and discard of the refractory ones.

```
[mk_(<TRI_PULSE>,<ATRIA>,906),
 mk_(<TRI_PULSE>,<ATRIA>,1908),
 mk_(<PULSE>,<ATRIA>,2912),
 mk_(<PULSE>,<ATRIA>,3913),
 mk_(<PULSE>,<ATRIA>,4917)]
```

7.2 Traces analysis

Validation conjectures are predicates over the execution traces produced while interpreting the specifications. There are three types of high level conjectures

- Separate
- SepRequire
- DeadlineMet

7.2.1 Validation conjectures

URL

Requirement: The minimum delay allowed between a ventricular event and the next ventricular pace that shall be 0.5 s thus 5000 ms.

This leads to the following conjecture:

```
Separate( #fin(Lead‘dischargePulse(-,<VENTRICLE>))
        , true
        , #fin(Lead‘dischargePulse(-,<VENTRICLE>))
        , 5000
        , false )
```

Fixed AV Delay

Requirement: After an atrial event there must be an ventricle pace after 150 ms +/- 4ms.

```
SepRequire( #fin(Lead‘dischargePulse(-,<ATRIA>))
          , true
          , #fin(Lead‘dischargePulse(-,<VENTRICLE>))
          , 1460
          , true )
```

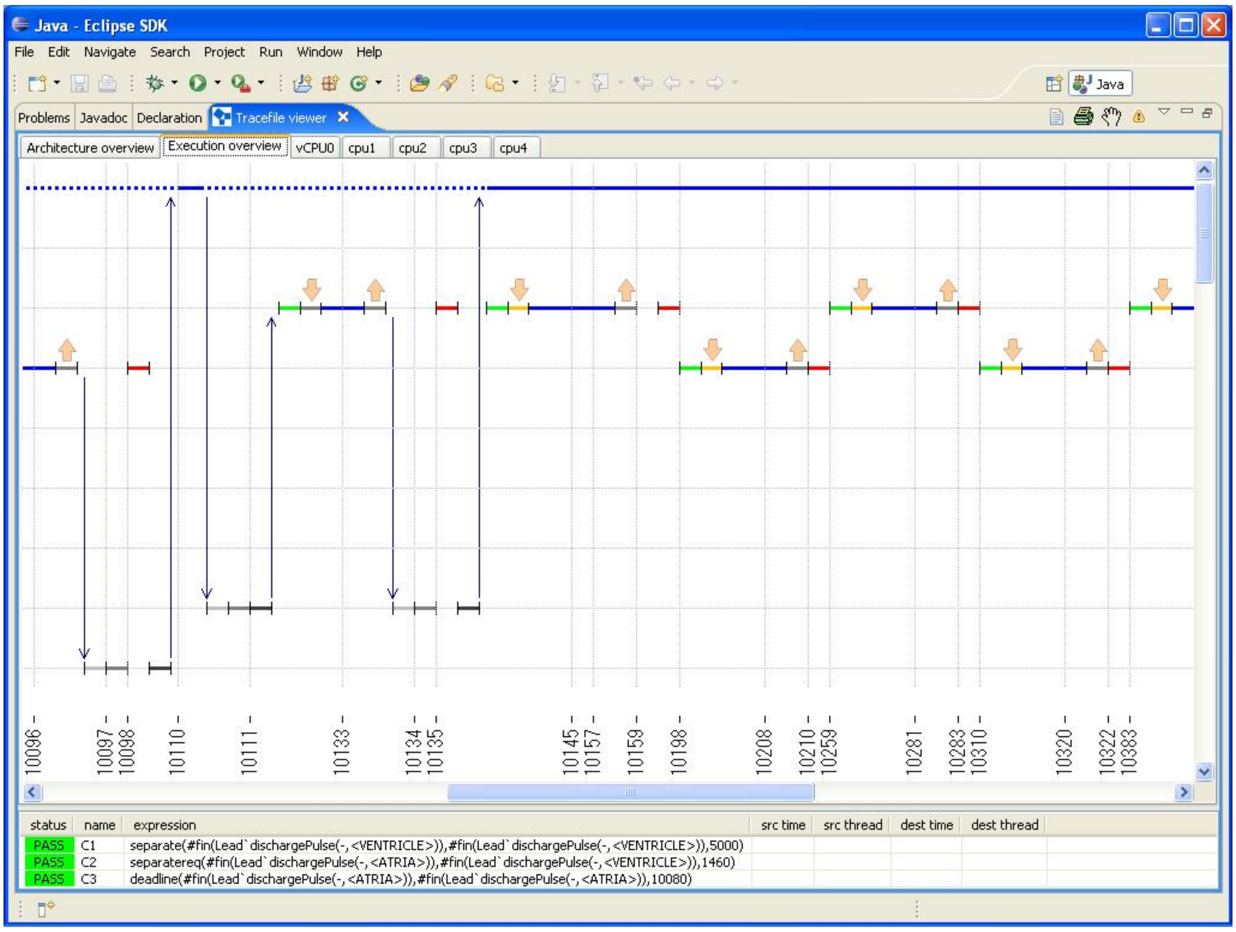
LRL

Requirement: The maximum delay allowed between pulses is 1000 ms +/- 8.

```
deadlineMet( #fin(Lead‘dischargePulse(-,<ATRIA>))
           , true
           , #fin(Lead‘dischargePulse(-,<ATRIA>))
           , 10080
           , false )
```

7.2.2 Traces

After evaluating the model with the validation conjectures the showtrace tool was used to inspect the traces and figure 7.1 show the result of it.


Java - Eclipse SDK
File Edit Navigate Search Project Run Window Help
Java
Problems Javadoc Declaration Tracefile viewer
Architecture overview Execution overview vCPU0 cpu1 cpu2 cpu3 cpu4
10096
10097
10098
10110
10111
10133
10134
10135
10145
10157
10159
10198
10208
10210
10259
10281
10283
10310
10320
10322
10383

status	name	expression	src time	src thread	dest time	dest thread
PASS	C1	separate(#fin(Lead`dischargePulse(-,<VENTRICLE>)),#fin(Lead`dischargePulse(-,<VENTRICLE>)),5000)				
PASS	C2	separatereq(#fin(Lead`dischargePulse(-,<ATRIA>)),#fin(Lead`dischargePulse(-,<VENTRICLE>)),1460)				
PASS	C3	deadline(#fin(Lead`dischargePulse(-,<ATRIA>)),#fin(Lead`dischargePulse(-,<ATRIA>)),10080)				

Figure 7.1: Screen from showtrace tool

Chapter 8

Conclusion

The overall feeling is that the level of abstraction was correct while suffering the necessary changes throughout the modelling process. Because the goal was to develop three models: (sequential, concurrent and distributed real-time) the amount of time dedicated to modelling and understanding the pacemaker different operating modes was too short. OFF, AOO, AAT, AAI, DOO, DDD, AOOR and DDDR were the modelled modes. Although this is still really far from the 19 different modes, 18 from the 26 variables involved while controlling the heart were covered.

The model evolution from Sequential to DR-T revealed itself very useful because one can build a regression test suite helping on the debugging of the more complex models and easily after changing the model discover if it continues valid.

When a bug is discovered we can filter easily if the problem arises due to:

- algorithm in the case the sequential model test fails
- synchronisation, if the detected at concurrent model level
- or if its a problem of timing or distribution causing the bad execution on the DR-T model

If we were starting modelling now we would focus on the regular modes instead of the rate adaptive ones, focusing just in the non rate adaptive mode, because now that the relations between the different became brighter we can conclude that the XXXR modes are just the normal ones with rate changing. This means that AOOR is just the AOO with the rate adaptation process. The same relation happens between analogous modes, AOO and VOO, which have similar algorithms and so we just need to study one to understand this kind of modes, XOO in this case. Instead of considering these criticisms as a failure we conclude that this is normal and part of the modelling process:

- Build a model representation of our understanding
- Gain insight while building and reasoning about the model

- Adjust the model

8.1 General requirements review

- The Pacemaker shall support single and dual chamber rate adaptive pacing [17, p. 13 s. 3.1]. **Modelled**
- The device shall output pulses with programmable voltages and widths (atrial and ventricular) which provide electrical stimulation to the heart for pacing[17, p. 16 s. 3.4]. **Not necessary for the purpose**
- The atrial and ventricular pacing pulse amplitudes shall be independently programmable[17, p. 16 s. 3.4.1]. **Not necessary for the purpose**
- The atrial and ventricular pacing pulse width shall be independently programmable[17, p. 16 s. 3.4.2]. **Not necessary for the purpose**
- The following bradycardia operating modes shall be programmable: Off, AOO, AAT, . . . , DDDR [17, p. 16 s. 3.5]. **Modelled so far Off, AOO, ,AAI, AAT, DOO, DDD, AOOR, DDDR.**
- The device shall have the ability to adjust the cardiac cycle in response to metabolic need as measured from body motion using an accelerometer[17, p. 32]. **Modelled**.
- The accelerometer shall determine the rate of increse of the pacing rate [17, p. 33 s. 5.7.4].**Partially modelled**.
- The accelerometer shall determine the rate decrease of the pacing rate[17, p. 33 s. 5.7.5]. **Partially modelled**.

8.2 Related work

Emma Nichols from Newcastle University is building a GUI interface to the models using the Corba API distributed with VDMTools. Thanks to this work we can easily “view” the system working and facilitate the communication with the stakeholders that don’t understand VDM notation.

A group leaded by Jeremy Bryans at Newcastle University is designing a VDM Mondex specification, in the context of the **Mondex Case Study** [5] challenge. In this work VDM++ is used as a modelling language with a particular interest in refinement, proof, testing and tools.

The method is also being used at Minho University in the context of the **Verifiable File System** [10] mini-challenge. The objective is to write a formal specification in VDM++ to disambiguate and formalise POSIX file-system prose. This is starting at the moment as a practical assignment during the course MFP2 given by José Nuno Oliveira.

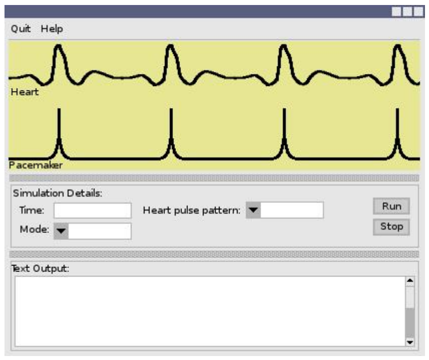


Figure 8.1: Pacemaker GUI prototype

An approach with abstract and concrete models is presented in the Electronic Purse case study [18] a security critical Smartcard product. In this work the simplicity of the abstract model was valuable to express the security properties in a client understandable way. We feel that our VDM-SL models work similarly.

8.3 Future work

The RateController needs to take into account the reaction time and recovery time variables while changing the rate. Also the meaning of the response factor needs to be clarified.

To fully understand the different operation modes we need to understand the meaning of all the variables related to XXD tracked modes.

In the future the possible architecture could be as described in the diagram of picture 8.2 integrating the actual work with the models of the other components.

Another improvement would be the study and adjustment of the physical architecture, as also the time values in the threads definitions that actually are just empirical values.

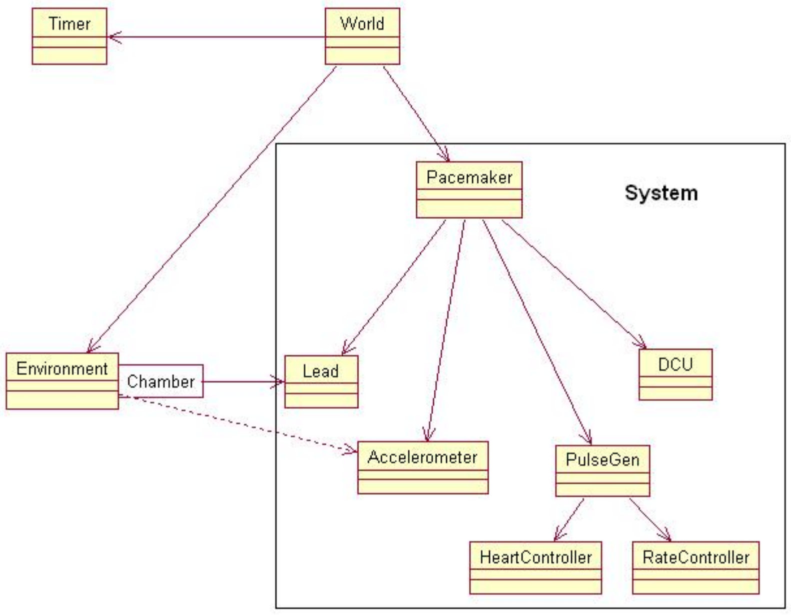


Figure 8.2: Rate control subsystem

Bibliography

[1] Alan Chan. The physics factbook, 2001. [Online; accessed 01-April-2007]. [cited at p. 3]

[2] Zhou Chaochen, C. A. R. Hoare, and Anders P. Ravn. A calculus of durations. *Inf. Process. Lett.*, 40(5):269–276, 1991. [cited at p. 8]

[3] CSK. *Developement Guidelines for Real-Time Systems Using* ***VDMTools***. CSK, 2006. [cited at p. 30, 51, 61]

[4] John Fitzgerald, Peter Gorm Larsen, Simon Tjell, and Marcel Verhoef. Validation Support for Distributed Real-Time Embedded Systems in VDM++. Technical Report CS-TR:1017, School of Computing Science, Newcastle University, April 2007. [cited at p. 8, 11, 61]

[5] Gerhard Schellhorn, Holger Grandy, Dominik Haneberg, and Wolfgang Reif. The Mondex Challenge: Machine Checked Proofs for an Electronic Purse. Technical report, Lehrstuhl für Softwaretechnik und Programmiersprachen, Universität Augsburg, February 2006. [cited at p. 68]

[6] CSK VDMTools Group. The dynamic semantics of csk vdm-++ vice. Technical report, CSK Coorporation, Japan, 2005. Confidential. [cited at p. 10]

[7] H. Rischel, K.M. Hansen, A.P. Ravn. Formal requirements and design for a gas-burner. Technical report, ProCoS Rep. ID/DTH HR4, 1991. [cited at p. 8]

[8] Tony Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003. [cited at p. 3]

[9] Biomedical Engineering BE513: Biomedical Instrumentation. Ecg, 2007. [Online; accessed 01-April-2007]. [cited at p. 4, 77]

[10] Rajeev Joshi and Gerard J. Holzmann. A mini challenge: Build a verifiable filesystem. 2006. [cited at p. 68]

[11] Software Quality Research Laboratory. Pacing modes, 2007. [Online; accessed 07-June-2007]. [cited at p. 5]

[12] Software Quality Research Laboratory. Timing cycles, 2007. [Online; accessed 07-June-2007]. [cited at p. 16, 18, 20]

[13] Li Li and He Jifeng. Towards a denotational semantics of timed rsl using duration calculus. Technical Report 161, UNU/IIST, P.O.Box 3058, Macau, April 1999. Accepted for publication by Chinese Journal of Advanced Software Research. [cited at p. 8]

[14] St. Jude Medical. Pacemaker implantation, 2007. [Online; accessed 01-April-2007]. [cited at p. 5, 77]

[15] Paul Mukherjee and Victoria Stavridou. Decomposition in real-time safety-critical systems. *Real-Time Syst.*, 14(2):183–202, 1998. [cited at p. 9]

[16] UCSF School of Medicine. Pacemaker module, 2007. [Online; accessed 01-April-2007]. [cited at p. 5]

[17] Boston Scientific. Pacemaker system specification. pages 1–35, January 2007. [cited at p. 3, 5, 13, 14, 47, 68]

[18] Jim Woodcock Susan Stepney, David Cooper. An electronic purse specification, refinement, and proof. Technical report, Oxford University Computing Laboratory, 2000. [cited at p. 69]

[19] Wikipedia. Artificial pacemaker – Wikipedia, the free encyclopedia, 2007. [Online; accessed 01-April-2007]. [cited at p. 5]

[20] Wikipedia. Bradycardia — Wikipedia, the free encyclopedia, 2007. [Online; accessed 7-June-2007]. [cited at p. 4]

[21] Jim Woodcock. First steps in the verified software grand challenge. *Computer*, 39(10):57–64, 2006. [cited at p. 3]

[22] C. Zhou and M. R. Hansen. *Duration Calculus: A Formal Approach to Real-Time Systems.* EATCS: Monographs in Theoretical Computer Science. Springer, 2004. [cited at p. 8]

Appendices

Appendix A

Differences between the two Concrete Syntaxes

Below is a list of the symbols which are different in the mathematical syntax and the ASCII syntax:

Mathematical syntax	ASCII syntax
$\cdot$	`&`
$\times$	`*`
$\leq$	`<=`
$\geq$	`>=`
$\neq$	`<>`
$\overset{o}{\rightarrow}$	`==>`
$\rightarrow$	`->`
$\Rightarrow$	`=>`
$\Leftrightarrow$	`<=>`
$\mapsto$	`|->`
$\underline{\triangle}$	`==`
$\uparrow$	`**`
$\dagger$	`++`
$\overset{\text{m}}{\cup}$	`munion`
$\triangleleft$	`<:`
$\triangleright$	`:>`
$\triangleleft\!\!-$	`<-:`
$\triangleright\!\!\!\!-$	`:->`
$\subset$	`psubset`
$\subseteq$	`subset`
$\curvearrowright$	`^`

Mathematical syntax	ASCII syntax
$\bigcap$	`dinter`
$\bigcup$	`dunion`
F	`power`
$\ldots$-set	`set of ...`
$\ldots^*$	`seq of ...`
$\ldots^+$	`seq1 of ...`
$\ldots\xrightarrow{m}\ldots$	`map ... to ...`
$\ldots\xleftrightarrow{m}\ldots$	`inmap ... to ...`
μ	`mu`
$\mathbb{B}$	`bool`
$\mathbb{N}$	`nat`
$\mathbb{Z}$	`int`
$\mathbb{R}$	`real`
$\neg$	`not`
$\cap$	`inter`
$\cup$	`union`
$\in$	`in set`
$\notin$	`not in set`
$\wedge$	`and`
$\vee$	`or`
$\forall$	`forall`
$\exists$	`exists`
$\exists!$	`exists1`
λ	`lambda`
ι	`iota`
$\ldots^{-1}$	`inverse ...`

In the following pages I present the minutes of the OvertureTool group that I was responsible for.

List of Figures

List of Tables