A Graphical Validation Tool for a VDM++ Model of a Cardiac Pacemaker

Emma Nicholls

August 2007

Abstract

A Graphical User Interface was designed and implemented for use in the validation of a formal model of a cardiac pacemaker. The models are executed using the VDMTools interpreter and communication with the validation GUI is carried out via the Toolkit's CORBA API. Some changes to the model were required in order to facilitate this communication: these changes were successfully carried out. A scenario editor for the models was designed but not implemented due to time constraints.

I declare that this dissertation represents my own work except where otherwise stated.

Acknowledgements: John Fitzgerald for supervision, advice and support. Richard Hornby, James Clarke, Stella Deacon and the family for far too many things. Finally John Nicholls, for making this possible.

Contents

Chapter 1

Introduction

A set of formal models have been produced describing the behaviour of a pacemaker as specified by the Formal Methods Pacemaker Grand Challenge [7]. These models must now be validated to ensure they meet the requirements for the pacemaker. In order to do this, validation must be carried out both by formal modelling experts (to ensure the models are free of inconsistent behaviour) and by domain experts (to ensure the models produce the appropriate behaviour for any given situation). However, some difficulty may arise due to the disparate nature of the areas of expertise these two groups possess. In short:

> "Software engineers cannot recognise potential difficulties with the requirements because they don't understand the domain; domain experts cannot find these problems because they don't understand the specification." [10]

A means is required by which the models can be validated to the satisfaction of both parties, by allowing the modelling experts to demonstrate clearly to the domain experts the behaviour of a model without the need to explain the model itself.

This problem can be largely resolved through the means of rapid prototyping. A Graphical User Interface can be constructed which allows manipulation of the model, without the need to know the language in which it is written. This allows domain experts to evaluate models of systems they know about. This project focuses on the design and implementation of such a prototype.

The main aim of the project is to construct a Graphical User Interface to allow manipulation of the pacemaker models constructed by Macedo [3] for the Ingeniørhøjskolen i Århus. There are three flavours of model, corresponding to stages in the development process: a sequential model with time steps, a concurrent model with the time steps replaced by a timing thread, and finally a distributed real-time (DRT) model which models the behaviour of the pacemaker when distributed across multiple CPUs. All three models are manipulated through the construction and execution of scenarios which are read into the models from a file which is passed as an argument to the constructor

of the initialisation class (`World`). The validation GUI should allow a user to investigate the behaviour of the models without any technical understanding of their design or implementation language. It should be possible to ensure the model provides the intended behaviour for given situations.

The term 'domain expert' refers to a person whose area of expertise covers (in whole or in part) the problem domain of the system. Domain experts are usually specialists in some field in which the system is required, but are not familiar with the software engineering techniques and languages used to specify the system. As such they are knowledgeable about what the model represents but not the language it is written in. This communication gap must be bridged to avoid the production of a specification which does not meet the intended requirements. Possible domain experts in this case include cardiac specialists and pacemaker designers.

'Formal modelling' refers to the use of a mathematically rigorous model of some aspect of a system in order to aid understanding of that system. It is the software engineering equivalent of the modelling processes which are integral to the specification and design of systems in other engineering disciplines, but is less widely-used in software engineering than in those disciplines. Modelling the entire system would be too complex for reasonable evaluation, but concentrating on an aspect and eliminating unnecessary detail enables the rigorous validation of the model to ensure internal consistency.

The languages used to construct formal models have formally-defined vocabulary, syntax and semantics [10]. This allows the construction of formal specifications which are consistent, complete and unambiguous. [8]. These specifications can be more effectively checked against the requirements of the system than could a specification expressed in a natural language:

> "The formal syntax of a specification language enables requirements and design to be interpreted in only one way, eliminating ambiguity that often occurs when a natural language (e.g., English) or graphical notation must be interpreted by a reader." [8]

The need to clarify everything in order to be able to specify it unambiguously also ensures more effective elicitation and analysis, as indicated by Sommerville:

> "[Formal specificaton] forces a detailed analysis of the requirements, which is an effective way of discovering problems in the specification. In a natural language specification, errors can be concealed by the imprecision of the language. This is not the case if the system is formally specified." [10]

However, this increased accuracy and reduced ambiguity comes at a cost: specifications written in formal languages are considerably less comprehensible to domain experts, customers and other stakeholders not trained in their use than specifications written in natural languages.

The outline of this dissertation is as follows: In this Chapter, the problem to be addressed by the project has been discussed. Background material relevant to

the problem and its solution can be found in Chapter 2 along with an overview of the languages, tools and techniques involved in the solution. Chapter 3 explains more fully the scope and objectives of the project. In chapter 4 the specification and design of the solution system are discussed, with reasoning and, where applicable, alternative approaches. Information is also provided on testing strategies. Chapter 5 presents the solution system. In Chapter 6, the project outcomes are evaluated against the objectives and in Chapter 7 conclusions are drawn and possible areas of further work are indicated. Appendix A contains the source code for the validation prototype.

Chapter 2

Background Material

2.1 Formal Modelling

Formal methods are a set of activities for software design and specification which use mathematics to ensure program correctness. Their key feature is the use of a mathematical specification language to allow the use of logical reasoning in the analysis of specifications written in this language [1]. They are widely held to be most useful for safety-critical or dependable systems, though Anthony Hall of Praxis Systems holds that they are useful for all systems with high cost of failure [5]. Formal methods are not commonly used in software engineering, largely due to perceptions of high cost and the requirement for specialised staff to be involved. Formal modelling is a formal method of software development which combines abstraction with rigour to produce a cost-effective means of creating high-integrity software, and one which can be carried out by developers without the need for extensive mathematical training. Modern formal modelling languages are generally object-oriented due to the high potential for useful abstraction. These main aspects are detailed below.

Abstraction involves the removal of unnecessary detail, which in an object-oriented model can be carried out by looking only at the external view of some or all of the objects involved. This allows unnecessary information to be hidden from the model, resulting in a less complex model which is easier to verify and less likely to contain inconsistencies [9]. Since formal methods scale badly with complexity, this will also make the process cheaper and faster.

Rigour indicates the extent to which objective analysis of a model is possible. A model written in mathematical notation may be fully rigorous, in that all its properties may be proven. In contrast, a model written in a natural language is highly ambiguous and hence subjective. However, a model written in mathematical notation may be difficult to read for customers and domain experts as well as developers themselves, whereas a natural language specification is generally comprehensible to all stakeholders. Formal modelling languages attempt to compromise by providing an unambiguous but readable grammar and syntax,

and usually a means of commenting this language in natural language as well. The models can be verified using mathematics to detect inconsistencies, but are more readable than they would be in a purely mathematical notation.

2.2 Validation

Validation is an essential phase of software development and is concerned with ensuring a specification conforms to the customer's requirements. In comparison its partner phase, verification, is concerned with ensuring the implementation conforms to the specification. Informally-phrased, their difference is:

> 'Verification: "Are we building the product right?"
> Validation: "Are we building the right product?"' [2].

In formal modelling, validation is a means of ensuring that the model behaves in the manner expected of the system it represents. There are three main validation methods in formal modelling: the construction and proof of 'integrity properties', systematic testing and visualisation.

Integrity properties are conditions which must hold, expressed in formal (and hence provable) notation [4]. There are two types: those that must hold in order to ensure internal consistency in a model (invariants), and those that will hold if the model provides an accurate portrayal of the system it represents ('validation conjectures'). Many, though not all, integrity properties can be automatically checked for correctness under given conditions. The simplest integrity checking is that of syntax and type consistency, which can be carried out automatically by the modelling tools [4]. Models may also be validated using systematic testing - this is generally carried out automatically using a prepared set of tests.

Unlike the methods outlined above, visualisation (also known as rapid prototyping) allows for validation by domain experts and customers. This is achieved by constructing a graphical interface to the model to allow manipulation without having to read or understand the underlying language. This method is the focus of this project.

2.3 Languages and Technologies

The main languages used in the project were the VDM++ modelling language, and the Java programming language. The tool used to interpret the models was VDMTools, which has an external API based on CORBA technology. A brief introduction to these languages, tools and technologies is outlined here.

2.3.1 VDM++

VDM++ is an object-oriented version of the Vienna Development Method Specification Language (VDM-SL) which was developed in the 1980s. VDM-SL

expanded and superceded Meta-IV as the notation used by the Vienna Development Method [9] and became an ISO standard in 1996 [4] The Vienna Development method is a model-oriented collection of formal techniques for developing computing systems from models expressed in a rigorous and unambiguous specification language [6] VDM-SL provides objects and classes [9], but not inheritance. The Pacemaker requirements are written in VDM-SL.

VDM++ improves on VDM-SL by providing support for inheritance (hence becoming truly object-oriented [9]). It also provides some concurrency modelling functionality. The pacemaker models were written in VDM++. A brief overview of VDM++ is provided below: this is based on the excellent introductory guide provided in "Validated Designs for Object-Oriented Systems" [4].

VDM Classes

A VDM++ model consists of a collection of class definitions. The language is case sensitive. The basic structure of a class definition is as indicated below: the class may contain any number of these blocks, usually in any order.

```
class className [is subclass of parent1, parent2,...]
  values             // constants defined. A value must be defined
                     // prior to its first use.
    type = value;
  types              // data types defined
    type Type = ...;
  instance variables // fields defined: internal state of object
    varName : Type [:= value];
  operations         // operations defined (incl. constructor)
  functions          // functions defined
  thread             // thread defined (if an active object)
  sync               // concurrency synchronisation constraints
                     // eg. mutex, per
```

Scope

The available access modifiers for VDM++ classes, values and variables are public (indicated by a +), protected (indicated by a `#`), and private (indicated by a -). The default accessibility is Private. Where a field or method from another field is to be used, and name confusion is possible, name qualification should be used: the fully-qualified variable or method name consists of the classname followed by the variable/method name, separated by a backquote, eg. `Pacemaker‘accelerometer`.

An invariant is an expression (which can be applied to instance variables, functions or operations) which declares a condition which must hold in order for the model to remain consistent. The invariant consists of the keyword `inv` followed by the expression which must be met.

Code fragments enclosed in round brackets are known as block statements - their contents will be executed sequentially. Invariants are applied to the

block as a whole rather than after individual statements in the block. Variables declared within a block statement are local variables, and will have a scope limited to that block. Local variables are declared using the `dcl` statement. Declarations are of the form `identifier : type := expression`:

```
(dcl declarations;
 statements
)
```

Operations and Functions

The difference between an operation and a function is that an operation may access or modify the class's instance variables, whereas a function may not. Both operations and functions may be declared implicitly (providing an operation/function signature and a postcondition stating what properties the result must have), or explicitly (providing an expression or algorithm that is to be used to calculate the output).

The RESULT keyword refers to the result of a function or operation. It is most commonly used in the construction of postconditions. VDM++ also contains a 'don't care' pattern, consisting of a hyphen used in place of a type. This indicates that the value doesn't matter for subsequent calculation, and is often used in conjunction with inheritance (for example, where implementation is delegated to a subclass).

An explicit operation has the form:

```
opname: param type ==> result type
opname(params) == statements
pre predicate
post predicate
```

The `pre` and `post` keywords indicate conditions which must be held by the input and output respectively. An explicit function is similar to an explicit operation, but with a signature of the form:

```
functionname: param type -> result type.
```

A block of code in VDM++ can be simplified by using a `let` or `def` expression: these define a name to stand for a more complex term. `def` and `let` are almost identical in syntax, but with `def expression1` can contain instance variables.

```
let pattern = expression1 in
   expression2

def pattern = expression1 in
   expression2
```

Looping and branching

VDM++ has the usual looping and branching structures, including for and while statements, and if and case statements. These bear a significant resemblance to their equivalents in procedural programming languages, so their meaning should be clear:

```
for identifier = expression1 to expression2 do
  statement

while predicate do
  statement

Conditionals:
if predicate
then expression1
else expression2

cases expression:
  pattern list 1 -> expression1,
  pattern list 2 -> expression2,
  ...
  pattern list n -> expressionn
  others         -> expressionothers
end
```

Collection types

There are three main collection types in VDM++: set, sequence and map. These are unbounded when used in the language, though this must necessarily be compromised for interpreter implementation. The universal and existential quantifiers are often useful when needing to test properties across entire collections. The universal quantifier (`forall bindlist & predicate`) is true if the predicate is true for all relevant identifiers, otherwise it is false. The existential quantifier (`exists bindlist & predicate`) is true if the predicate is true for at least one identifier.

A set is an unordered collection of values, which cannot contain duplicates. It is represented in VDM++ by a pair of curly brackets enclosing the values, which may be enumerated and comma-separated (eg. `{1,3,5,9}`) or constructed through set comprehension, for example:

```
{{create a value using x | x in set s & test x}
```

A variable is declared to be a set using the phrase `set of setType`. Sets in VDM++ are very flexible about member types, and can contain other sets as members.

To test whether a value is a member of a set, use `value in set setName`. The operator `card` can be used to obtain the cardinality of a set (the number of elements it contains).

The 'let be such that' statement (`let x in set s be st predicate on x`) - picks a random value from s for which the predicate evaluates to true. There must be at least one suitable value in the set, and this must be ensured by use of a precondition if necessary.

A sequence is an ordered collection of values. Unlike sets, sequences can contain duplicates. They are represented in VDM++ by a pair of square brackets enclosing the values, which as with the set may be enumerated and comma-separated or constructed through sequence comprehension. A variable is declared to be a sequence using the phrase `seq of seqType`. The type `seq1` refers to a non-empty sequence. Individual elements in a sequence can be modified using the sequence modifier: `seqName ++ { i |-> expression }` This replaces the element at position i with the result of the expression.

The most commonly-used sequence operators are as follows:

- hd: returns first element of the sequence.
- tl: returns the sequence without its first element.
- inds: returns the sequence's set of indices.
- elems: returns a set containing the elements in the sequence. As a set cannot contain duplicates, duplicate elements are removed.
- conc: concatenate tow sequences.

A mapping is a set of values (the domain) which is associated with another set of values (the range) via a relationship between the two sets. An injective mapping is a mapping in which the relationship is bidirectional: there is a one-to-one relationship between each item in the domain and its associated item in the range so the links can be followed both ways.

In VDM++, a mapping is represented by:

```
{"Name" |-> value1, "Name2" |-> value2, ...}
```

It is declared using the phrase `map type1 to type2`. The injective mapping: is obtained using `inmap type1 to type2`. The operator `dom` refers to the domain of a mapping and the operator `rng` refers to the range.

Types

There are two kinds of type in VDM++: basic types (which are atomic - they cannot be deconstructed in any way) and constructed types (which are composites of basic types and often other constructed types) Any type surrounded by square brackets is considered an optional type: it has possible values of all the values in type plus `nil`.

The basic types consist of:

- bool true,false
- int ...,-1,0,1,...
- nat 0,1,2,...
- nat1 1,2,3,...
- rat rational numbers
- real real numbers
- char ‘a’,‘b’,...,‘1’,‘+’,etc.
- quote type <name> [The quote type contains one value: a quote literal (eg. <NAME>) with the same name as the quote type.]
- token type `mk_token(...)` [A token has no internal structure and serves as a completely abstract type, which can be replaced by more solid types later when the details are better-understood and a good choice can be made. The token is constructed using the `mk_token(tokenName)` operator.]

The VDM++ constructed types consist of the Union type, the Product type and the Record type. An object can also be considered a type: such object reference types are identified by their classname (eg. `IO := new IO()`).

- A Union type consists of a series of types separated by pipes, for example:

  ```
  type1 | type2 | ... | typen!
  ```

  A type which appears anywhere in this list of types is considered to belong to that Union type. This is useful for enumerated types.

- A Product type consists of a series of types separated by asterisks. In order to belong to the product type, a type must be a composite type (a tuple) containing a value from each type. These are constructed using `mk_(type1,type2,...,typen)`.

- A Record type is similar in behaviour to the Product type, but the tuple components cna be given field names and referred to by these names instead of by index.

2.3.2 Java

Java is an object-oriented programming language designed to be platform-independent and secure. There is a rich variety of standard library classes for building user interfaces, handling operating system properties and various other things including CORBA support. The VDMTools CORBA API provides pre-compiled IDL stubs and skeletons for Java and C++ use. Since platform independence was a desirable feature (to match the availability of the Toolkit and allow use of the GUI by as wide a variety of users as possible), the GUI was implemented in Java.

2.4 The Pacemaker Models

The model for which a graphical interface is required has been constructed by Macedo, and is described in detail in the technical report [3]. The work outlines the requirements of the model and three solutions, which develop the internal representation of time from simple sequential with time stepping to a fully distributed real-time model. All the models have identical external appearance: scenarios are executed by creating a new `World` object with an argument file and a pacemaker mode as parameters, and calling its `Run()` method:

```
create n := new World("scenario.arg",<AOO>).Run();
```

The `Run()` method prints the results of the scenario to the standard output device via the `ShowResult()` method in `Environment`:

```
public
showResult : () ==> ()
showResult () ==
   def - = io.writeval[seq of Outline](outlines) in skip;
```

The models are intended to be executed via these methods.

The sequential model consists of two subsystems - a heart regulator subsystem and a rate controller subsystem - shown in Figures 2.1 and 2.2 respectively. The pacemaker's surroundings are represented by the class `Environment`, which is responsible sending stimuli to the pacemaker and receiving outputs. As such, pacemaker scenarios are executed through the `Environment` class at a very high level. The `World` class exists mainly as a means of constructing the pacemaker and its associated `Environment`, and passing the `Environment` the needed information for the execution of a scenario. It also holds a `Timer` object, which provides a representation of time in the system. In the sequential model, time management is performed by stepping, with all activities for the given 'step' carried out before the counter is incremented and the next step begins. The `Environment` class performs the stepping by calling a `StepTime()` method in the `Timer` object, and `Step()` methods in the pacemaker's `heartController` and `rateController` objects.

The concurrent model is broadly similar to the sequential model, the differences being in the way time is simulated. Instead of a sequential stepping mechanism, the concurrent model uses multiple threads and a Wait-Notify mechanism. Each activity thread notifies the environment thread after carrying out its activities for the current time interval, then blocks, and when all threads have completed the environment thread updates the clock and notifies all activity threads that a new interval has started. All changes made to this model are to accommodate this mechanism. The changes mostly consist of modifications to classes to replace `Step()` methods with appropriate thread definitions, the use of permission predicates to indicate whether threads have finished and mutexes to protect variables that are accessed by different methods. A new class (`ClockTick`) is also added to trigger the environment thread at appropriate intervals.

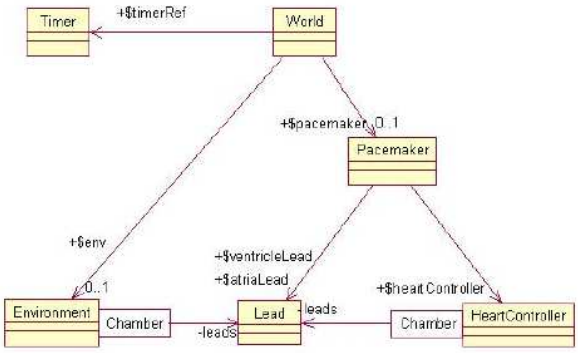

Figure 2.1: The Heart Controller Subsystem of the Sequential Pacemaker model.

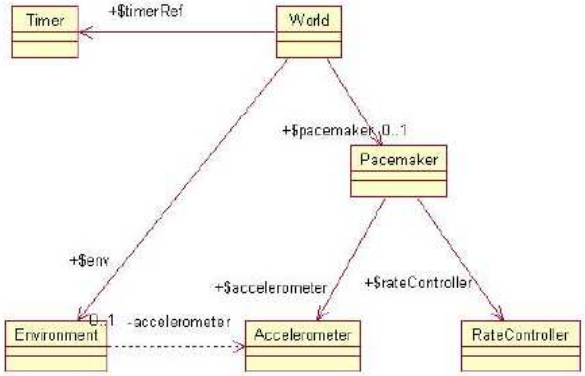

Figure 2.2: The Rate Controller Subsystem of the Sequential Pacemaker model.

Chapter 3

Project Outline

3.1 Scope and Objectives

The overall aim of this project is to develop a graphical user interface to facilitate the validation of an existing formal model of a cardiac pacemaker.

The intended objectives are as follows:

- To produce a requirements specification for the interface and and a description of the actors who will use it and their capabilities.
- To produce an interface in Java to allow communication between the VDMTools interpreter and the graphical interface via CORBA.
- To design and implement a Graphical User Interface in Java which connects to and manipulates the Pacemaker models via the interface above.
- To extend this Graphical User Interface during a series of iterative development cycles consisting of evaluation, design and testing, providing extension functionality in each cycle.
- To demonstrate the Graphical User Interface's functionality on a collection of different scenarios and several different models.
- To design and implement an application which allows the user to build argument files for the Pacemaker models via a Graphical User Interface.
- To extend this application during a series of iterative development cycles consisting of evaluation, design and testing, providing extension functionality in each cycle.
- To become sufficiently familiar with the VDMTools application and its CORBA API to be able to use them effectively.
- To become sufficiently familiar with VDM++ to be able to comprehend, interact with, and if necessary modify, the models.

- To contribute to the Pacemaker model development effort.

The goals for the implementation as originally planned fell into six versions, intended for incremental delivery. Version 1 was to deliver all major functionality for the interaction GUI, with Version 2 delivering the major functionality for the scenario constructor application and the remaining versions providing functionality extensions to these. The outline goals were as follows:

- Version 1 (Core functionality):
  - The application should interface with the VDMTools VDM++ Toolkit via CORBA.
  - The application should allow manipulation of the Distributed Real-time model from the GUI.
  - The manipulation should be carried out by executing the World constructor with the chosen scenario.
  - The display should show the natural heartbeats and the pacemaker actions on a plot with a time scale.
  - The user should be able to select a scenario from a list of all model scenario (.arg) files in a scenario directory.
- Version 2 (Core functionality):
  - The user should be able to build, save, retrieve and edit scenarios using a scenario builder GUI.
  - The scenarios should be saved as valid argument files, as required by the models.
  - The scenario builder should allow the addition or removal of individual pulses from the scenario.
- Version 3:
  - The user should be able to generate pulse sequences from general specifications (for example, a regular sequence of pulses with every fourth one missing)
- Version 4:
  - The user should be able to see the scenario details whilst looking at the results.
- Version 5:
  - The user should be able to select a model on which to run a scenario.
  - The user should be able to choose whether to use the VDMTools VDM++ or VDM-SL Toolkit as the backend.
- Version 6:

- The user should have the option of displaying in the GUI the communication with the Toolbox.
- The user should be able to select a scenario from the filesystem.

Once more research had been carried out into how to meet these requirements, some points were noted which suggested changes to the implentation plan were needed. It was first noted that the Toolkit's CORBA API did not distinguish between the graphical and command-line versions of the toolkit and would simply connect to the first running instance of the correct kind (VDM-SL or VDM++) of Toolkit made available to it. This requirement was therefore removed. The model selection goal was then moved from Version 5 to Version 1 as this was considered useful to build into the first version. The scenario loading mechanism was removed from Version 6 and the Version 1 requirement changed to accommodate it as it was found that both options were of similar difficulty and the filesystem approach would be more useful. The ability for the user to see the scenario details whilst looking at the results was considered to be core functionality and hence moved to Version 1. As a result of these changes, the goal plan was rebuilt into two goals representing the two applications, with core and extension functionality for each. The validation GUI was considered a higher priority than the scenario builder, as the validation is an objective and the scenario argument files can be written manually. The revised requirements are as follows:

1. Application 1:
   (a) The application should interface with VDMTools via CORBA.
   (b) The application should allow manipulation of a chosen model from the GUI.
   (c) The manipulation should be carried out by executing the chosen model's World constructor with the chosen scenario.
   (d) The display should show the natural heartbeats and the pacemaker actions on a plot with a time scale.
   (e) The user should be able to select a scenario from a list of all the filesystem. The file must end in '.arg'.
   (f) The user should be able to see the scenario details whilst looking at the results.
2. Application 1 Extensions:
   (a) The user should have the option of displaying in the GUI the communication with the Toolbox.
   (b) The pulse display should be animated to show the pacemaker interaction in 'real-time'.
   (c) The scale of the displayed pacemaker traces should be changeable by 'zooming in' or 'zooming out'.

3. Application 2:

   (a) The user should be able to build, save, retrieve and edit scenarios using a scenario builder GUI.

   (b) The scenarios should be saved as valid '.arg' files.

   (c) The scenario builder should allow the addition or removal of individual pulses from the scenario.

4. Application 2 Extensions:

   (a) The user should be able to generate pulse sequences from general specifications (for example, a regular sequence of pulses with every fourth one missing)

The delivered implementation consists of a validation GUI that meets the Version 1 core requirements. All requirements for the core functionality were met, however the extension requirements were not. The Scenario Builder (Application 2) was not implemented due to time constraints, and is considered a target for further work.

3.2 Approach

The validation GUI's implementation can be split into two distinct parts: the CORBA interface and the GUI frontend. These have different characteristics and therefore were best suited to different software process models.

The CORBA interface consists of a set of operations for connecting to the VDMTools toolkit via its CORBA API, loading and checking a model and executing its methods. The requirements for this interface were largely fixed as they were defined by the VDMTools API interfaces and the design of the pacemaker models. As such it was considered that this part of the project was well-suited to the waterfall development model. The CORBA interface is core to the GUI, which cannot function without it, and so it was the first part of the validation GUI to be implemented.

The GUI frontend consists largely of operations for providing a graphical representation of the scenario to be run on a particular model, and the results it provides, along with the necessary event handling, control and initialisation code. An evolutionary development approach was taken to the GUI as this allowed for late changes to be made to the look of the user interface, its functionality and the information it displays. Since all of the feature extensions are to the GUI frontend, this approach suited their potential late introduction. The principle disadvantage of evolutionary development - a poorly-structured application - was countered by designing an underlying architecture for the GUI and evolving the details.

The primary risk to this project was schedule over-run: this was considered to be a high-likelihood risk as the full scope and likely timescale of some

tasks (for example using the Toolkit's CORBA API) was unclear at the project planning stages. The risk reduction strategies employed to counter this were:

1. To allow extra time for unexpected setbacks.
2. To re-evaluate the project's progress at regular intervals and re-plan accordingly.

Other risks included changes to the GUI requirements which could not be easily incorporated without changes to the architecture, which was considered a high risk to the GUI frontend and a low risk to the CORBA interface. The choices of development model were made with this risk assessment in mind.

The initial weeks of the project were devoted to the study of VDM++, CORBA, the VDMTools toolkit and the Pacemaker models. Once this research was largely completed, an implementation schedule was produced taking into account the likely timescales suggested by this work. The initial development was carried out on the CORBA interface, with assistance from 'Validated Designs' [4] and Richard Payne of Newcastle University.

Chapter 4

Specification

The validation package consists of two separate programs: the validation GUI and the scenario builder. The scenario builder allows the creation and modification of argument files to be passed to the model via the validation GUI. Since the scenario builder creates standard argument files for the models, it is not necessary for scenarios to be created using the scenario builder they can be produced manually if desired.

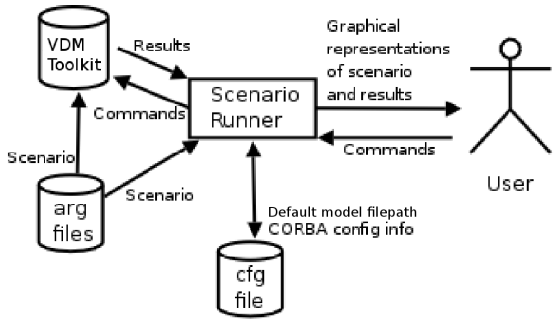

Figure 4.1: Context diagram for Validation GUI.

As can be seen from the context diagram for the validation GUI (Figure 4.1), the system has three main interfaces: to the user, to the filesystem and to the VDMTools toolkit. There are two filesystem locations which must be accessed: these responsibilities can be given to separate classes as the data to be read is very different. The specification for the interfaces is given in section 4.1.

The Scenario Builder has a somewhat simpler context diagram (Figure 4.2) as there is no CORBA interface - there are only two interfaces, between the user and the system, and between the system and the filesystem. The specification for these interfaces is described in section 4.2.

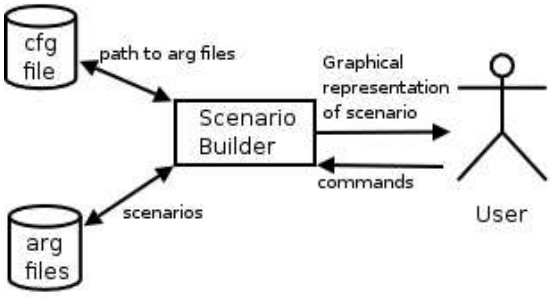

Figure 4.2: Context diagram for Scenario Builder.

4.1 Validation GUI

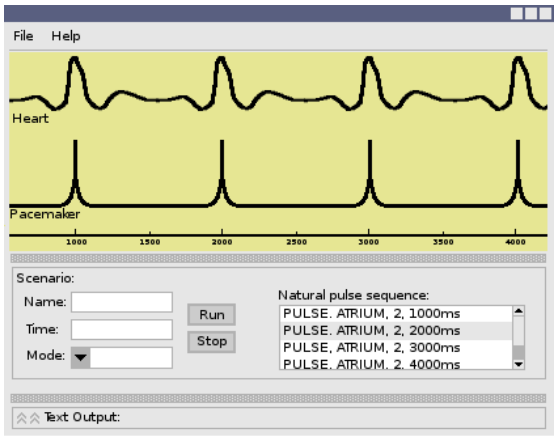

Figure 4.3: Proposed validation GUI Interface

The validation GUI, when initialised, presents the user with a GUI similar to the one in Figure 4.3. The directory containing the executable is searched for a configuration file during initialisation. If no such file is present, hardcoded defaults are used. Prior to the execution of a scenario the display window is blank.

4.1.1 User Interface

The File menu contains the options 'Load Model', 'Load Scenario', 'Preferences' and 'Quit'.

- 'Load Model' changes the model currently in use by the system: this will produce a dialog box with a drop-down list of the models available (those in the models directory) and 'Apply' and 'Cancel' buttons. A model should be loaded by default and identified on the GUI to ensure the user knows which model is currently in use.

- 'Load Scenario' loads a scenario described by an argument file selected by the user: this will produce a dialog box allowing the user to select an argument file from the filesystem. A file which does not end in '.arg' will not be a valid selection, and cannot be selected from the file selection dialog. Once selected, the details of the scenario appear in the Scenario Pane for viewing by the user.

- 'Preferences' allows the user to change the configuration options used in the GUI, namely the location of the default directory for the scenario argument files, the default model and (if any) the CORBA configuration details. This shall be done by dialog box with a text entry box for the directory path, a drop-down list for the default model and text entry fields for CORBA configuration details.

- 'Quit' closes the CORBA link and exits the program.

There should be an additional Help menu containing a 'Quick Help' option and a 'User Manual' option. 'Quick Help' should present a dialog box explaining the basics of how to use the GUI. 'User Manual' should present a window displaying the full user manual.

The Display pane shows a plot of natural pulses (from the argument file) against pacemaker signals (returned by the model) and a y-axis indicating the time since the start of the simulation in milliseconds. The pulse timings for the natural pulses are obtained from the argument file; the timings for the pacemaker signals are obtained from the data returned from the model.

The Scenario pane allows the user to view the currently-selected scenario. 'Name' and 'Time' may not be edited here, as these are part of the scenario argument file, but 'Mode' is changeable. The pulse sequence is displayed as a list which can be perused by the user. Clicking the button marked 'Run' executes the scenario on the currently-selected model. Selecting a pulse in the list will place a line on the graph corresponding to the location of the pulse (if implementation time available).

If the extension is implemented, the 'Text Output' pane will show the user the communication between the scenario executer and the model. This will be an optional screen, which is hidden by default. Clicking on the arrows will change the Text Output pane from minimised to visible or back to minimised.

4.1.2 CORBA Interface

The validation GUI must interface to the VDM Toolbox via its CORBA API. This can be achieved using the CORBA packages provided with the full version

of the VDMTools utilities, and the org.omg.CORBA Java packages. Initialisation consists of the following steps:

- Set up CORBA.
- Get a reference to the VDM Toolkit, either from the COS Naming service indicated in the preferences, or from a stringified object reference.
- Register the GUI as a client with the VDM Toolkit.

It is then necessary to load the model indicated by the preferences:

- Load the VDM++ project corresponding to the model.
- Get a reference to the VDMTools parser and parse the files in the project.
- Syntax and typecheck the files in the project to ensure the model is usable.
- Get a reference to the VDMTools interpreter and initialise it.
- Execute the selected scenario on the model by calling the World constructor.
- Parse the returned results and pass to the appropriate handling code.

When the user exits the program, the following steps are required:

- Close the project in the VDM Toolbox.
- Destroy all the objects used by calling `DestroyTag()`.
- Unregister the client from the server.
- Destroy the CORBA channel. As much of the above as possible is carried out in a self-contained unit.

4.1.3 Filesystem Interface

The validation GUI will need to read and parse the contents of the selected argument file in order to populate the Scenario Pane and `DisplayPanel` with the correct information. The filename itself must be passed to the model as a parameter to the `World` constructor in order to execute the model. The validation GUI will also need to be able to read the configuration file in order to request the correct CORBA Name Service, load the correct default model, set the correct default directory for arg files and supply any additional CORBA parameters to the ORB. It will also be necessary to be able to write these details to the configuration file in the correct places.

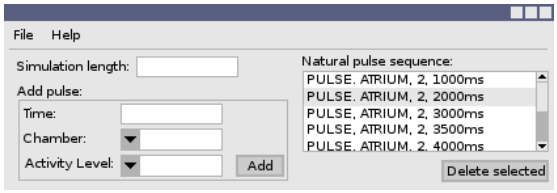

Figure 4.4: Proposed validation GUI Interface

4.2 Scenario Builder

4.2.1 User Interface

The Graphical User Interface, when initialised, presents the user with a GUI similar to the one in Figure 4.4. Initially no model is loaded but the user is able to specify a simulation length and add pulses to the (initially empty) pulse sequence. If a model is loaded, its length and the pulses already specified are displayed in the appropriate locations and can be edited. New scenarios and modifications to loaded scenarios may both be saved using the 'Save File' option.

The argument files are of the format required by the pacemaker models and are stored in a default directory whose path is specified in the configuration file. The user can opt to load from or save to a different location on the filesystem if desired.

To add a pulse, the user must specify a valid time in the 'Time' text entry field and choose a heart chamber and an activity level from the drop-down menus provided. The Activity Level is specified in word rather than number form but will be stored in the arg file in number form. Clicking the 'Add' button adds the indicated pulse to the pulse sequence. Duplicate pulses will be discarded. The user can delete one or more pulses from the pulse sequence by selecting them in the list and clicking the 'Delete Selected' button.

The File menu contains the options 'Load File', 'Save File' and 'Quit'. There will be an additional Help menu containing the same options as the equivalent menu in the validation GUI. If the extension is implemented, the 'File' menu will bear an additional option: 'Generate sequence'. This will allow the user to automatically generate a sequence of pulses which can then be edited by hand.

4.2.2 Filesystem Interface

In order to save a scenario, the scenario builder will need to open an argument file in the indicated location on the filesystem and write the contents to it, overwriting any previous file with that name. If this is to occur, a dialog box should alert the user to the situation and request confirmation that this is the desired outcome. The builder should automatically append the '.arg' extension if this is not supplied by the user.

To load a scenario, the scenario builder will need to read and parse the contents of the selected argument file in order to populate the Scenario Pane and result pane with the correct information.

Chapter 5

Implementation

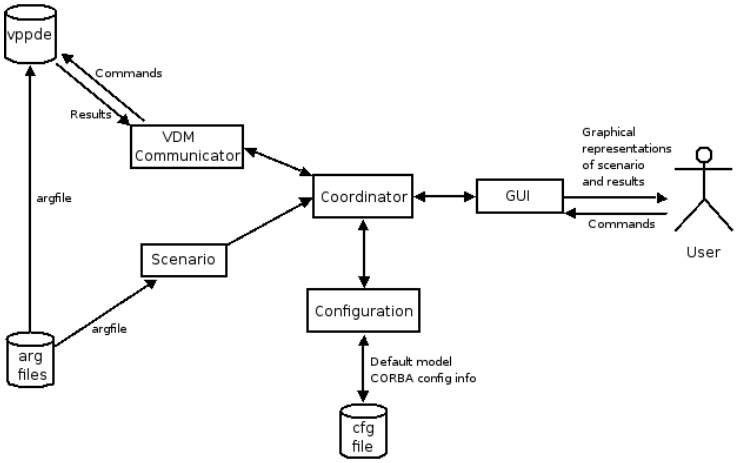

Figure 5.1: Data flow diagram for validation GUI: Subsystem level.

The data flow diagram for the validation GUI is shown in Figure 5.1. The diagram consists of a central coordinator and four subsystems, each interfacing to a different external source. The `VDMCommunicator`, `Scenario` and `Configuration` subsystems are each simple enough to be implemented by one class; `GUI` is further subdivided into a class which is responsible for the construction and maintenance of the graphical user interface itself and a class which is responsible for drawing and refreshing the pulse traces. The implementation also contains a class which represents a pulse in the system (with two subclasses, one for input pulses and the other for output pulses), and a parser class which is used by `Scenario` and `VDMCommunicator` to produce pulses from strings of input or output data.

This chapter is divided into two sections: the first covers the implementation of the `VDMCommunicator` as this represents the CORBA interface subsystem, and the second covers the implementation of the rest of the GUI. Full source code and Javadoc for the project can be found in the Appendix.

5.1 CORBA Interface

The CORBA interface subsystem forms the client half of a client-server architecture, the server being the VDMTools toolkit. The communication between the `VDMCommunicator` class and the toolkit falls into three main categories: initialisation, shutdown and scenario execution. The initialisation phase consists of obtaining a reference to the Toolkit and registering the client, and is carried out when the `VDMCommunicator` object is created. `Scenario` execution is carried out by loading the specified model, executing the commands:

```
VDMGeneric result = null;
String cmd =
    "create n := new World(\"" + argfile + "\"," + mode + ")";
itp.EvalCmd(cmd);
result = itp.EvalExpression(clientId,"n.Run()");
```

and retrieving the result from the returned `VDMSequence` object once the simulation is complete. The shutdown phase consists merely of destroying any objects left in the simulation using a call to `DestroyTag(clientId)`, then closing the connection cleanly.

The method of returning results provided in the models is to print the resulting pulse sequence to the Toolkit's terminal. Since this data cannot not be retrieved via the API, it was necessary to make some alterations to the models in order to be able to return the results to the GUI: an additional `ReturnResult()` method was added to the `Environment` class of each model, which returns the results as a sequence of characters (`seq of char`). The added methods are all identical and merely returned the contents of the `outlines` instance variable - the full method is shown below.

```
public
returnResult: () ==> seq of Outline
returnResult () == return outlines;
```

It was also necessary to add the line `env.returnResult()` to the method `World‘Run()` and change its return type from `void` to `seq of Outline`:

```
public Run: () ==> seq of Environment‘Outline
Run () == (env.Run();
           env.showResult();
           env.returnResult());
```

Once these modifications were made the models were accessible from the validation GUI.

It was also necessary to make a change to the DRT model in order to correct a scaling problem: the model uses a different internal timescale to the other models and as such it was necessary to divide the outgoing pulse times by ten in order to keep them on the same scale as the output from the other models. However, the inverse process was not carried out on incoming data which resulted in the DRT models requiring different scenario files to the other models. The model was therefore altered to perform the necessary data conversion on incoming data, thus internalising the scale change and allowing the use of the same scenario files for all models.

To perform this conversion, a second private function was defined in the DRTPacemaker Environment class which performed the inverse of the outgoing conversion function:

```
inconvert: seq of Inpline -> seq of Inpline
inconvert (s) ==
   [mk_(s(i).#1,s(i).#2,s(i).#3,floor(s(i).#3 * 10))
                                                | i in set inds s];
```

The incoming data was then altered to pass the data to this method before use:

```
public
Environment : seq of char ==> Environment
Environment (fname) ==
  def mk_(-,mk_(timeval,input)) = io.freadval[InputTP](fname)
  in (inplines := inconvert(input);
      simtime  := timeval*10
     );
```

5.2 GUI frontend

The validation GUI frontend has a largely object-oriented architecture, with tasks assigned to classes and hidden from the rest of the program. The visual appearance (Figure 5.2) of the GUI largely resembles that in the specification.

The 'File' menu contains the options: 'Load Model', 'Load Scenario', 'Preferences' and 'Exit'. 'Load Model', 'Load Scenario' and 'Exit' do as indicated in the specification. 'Preferences' is currently non-functional.

The Display Panel shows a natural heart trace and a paced heart trace, which are initially blank. When a scenario is loaded, the pulse sequence is displayed in the top half of the display panel. When a scenario is executed, the resulting pulse sequence is displayed in the bottom half of the display panel. Atrial pulses are marked with a short vertical line and ventricular pulses with a tall vertical line, reflecting the relative strengths of the pulses in the heart (though not to

scale) and serving to distinguish one type from the other at a glance. Triggered pulses are marked with a 'T'.

The control panel contains two pulse lists: one for the pulse sequence in the scenario and the other for the pulse sequence returned as the result of the scenario. Pulse data listed here is produced by the `toString()` method of each `Pulse` object. The model name, scenario name and scenario length are listed on the left hand side of the control panel, above the mode selection list and the 'Run' button. The selection list contains all the pacemaker operation mode options currently supported in the models, but these are specified in the code and so must be changed when the models acquire support for other modes. The excessive space on the left hand side of the control panel is due to unexpected Java layout manager positioning; it would be preferential to remedy this but it is not a functional aspect and hence is low-priority.

The typical use of the validation GUI is to run a chosen scenario on a chosen model with a chosen mode. The course of action required to do this is as follows:

- Start GUI. The VDMTools toolkit should already be running before this is attempted, or the GUI will not be able to initialise. Any valid VDMTools VDM++ toolbox should be acceptable. If there is a valid instance of the toolbox available, execute the Jar file and the GUI will load.
- Load Model. Loading a model simply involves selecting 'Load Model' from the 'File' menu, and selecting a file from the file system. Once a model has been selected, press 'Open' and the model will be loaded. This will be indicated by a change in the model name displayed on the GUI.
- Load Scenario. To load a scenario, follow a similar procedure to that used to load a model: select 'Load Scenario' from the 'File' menu and select a file from the file system. Unlike the file selection dialog used to load a model, the 'Load Scenario' file dialog only allows the selection of files ending in '.arg'. Once a scenario has been selected, press 'Open' and the scenario will be loaded: the scenario's name will be displayed on the GUI and the pulse sequence for the scenario will be loaded in the Scenario pulse list and the heart trace display.
- Run Scenario. To run the scenario, select a mode from the drop-down list and click the 'Run' button. This button will not work if no scenario is loaded. If a valid scenario and model are loaded, the scenario is executed on the model and the results will appear in the result pulse sequence list and the heart trace display.

5.3 Class descriptions

A brief overview of the functionality of each class in the application is given below.

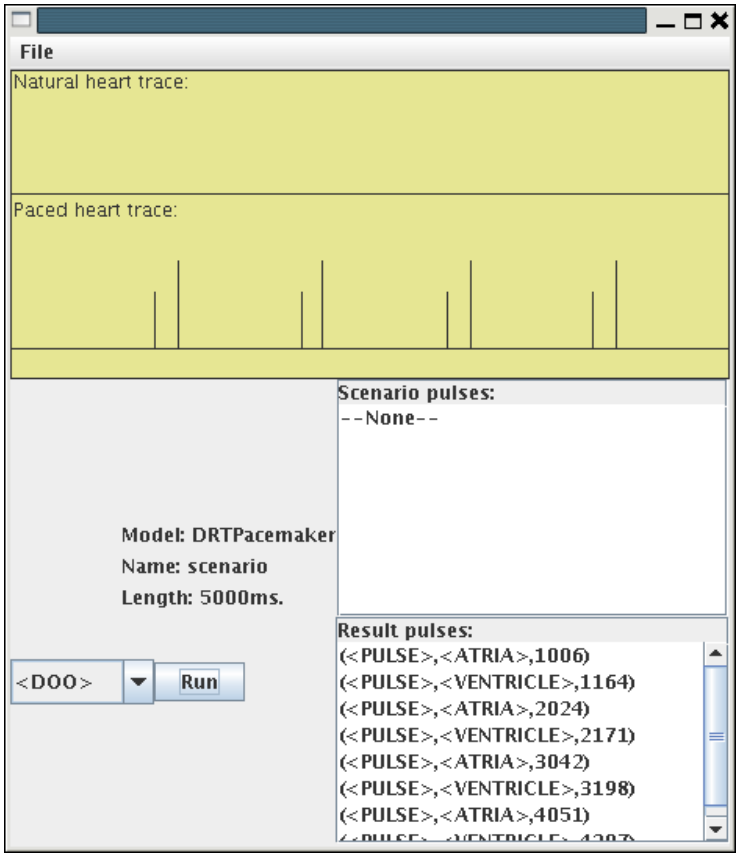

Figure 5.2: The appearance of the validation GUI (as implemented).

- `VDMCommunicator` implements the interface to the VDM Toolkit, communicating via CORBA. It allows the connection to, execution of scenarios upon, and disconnection from a currently-running `VDMApplication`. Basic communication log data is also stored in the `VDMCommunicator` object, and can be retrieved via an accessor method. `VDMCommunicator` is essential to the functionality of the program: if a connection to the toolkit cannot be made, the validationGUI cannot be used.

- Configuration `Configuration` represents the configuration data for the utility. The configuration data (consisting of a default model and a hostname and port for the CORBA name service) is held in member fields, and when a value is changed in the object it is written to the underlying configuration file. `Configuration` provides accessor and modifier methods for the configuration data. During initialisation a `Configuration` object is constructed from the configuration file in the directory containing the ex-

ecutable. The default model is stored in the `Coordinator` object and the CORBA information is used by the `VDMCommunicator` object during initialisation to attempt to connect to the CORBA name service when trying to obtain a reference to the toolkit. If the configuration file cannot be read or is malformed, a `Configuration` object cannot be made from it and the application uses a default constructor to produce a `Configuration` object with default values instead. These default values are unlikely to be what the user intended so it is important that the configuration file is customised.

The format of a configuration file is as follows:

```
\path\to\default\model\model.prj
hostname
portnumber
```

If any line is missing the Configuration object will be unable to read the file. However, the Configuration object is not able to spot invalid data: this will be revealed when VDMCommunicator is instantiated.

- Scenario `Scenario` represents a scenario in the system. Each contains in member fields the data from a scenario file. `Scenario` provides methods for accessing the attributes of a scenario, but cannot modify the argument files. `Scenario` reads the contents of an argfile and parses the result to retrieve the scenario length and a list of pulses in a `String`, which is passed to a `PulseParser` for conversion into an `ArrayList` of `Pulse` objects.

- PulseParser `PulseParser` is a utility class which parses a `String` into a `List` of pulses. The type of parser is indicated by a flag passed to the constructor: "Input" will result in a `PulseParser` which looks for and creates `InPulse` objects, "Output" will result in a `PulseParser` which looks for and creates `OutPulse` objects. However the return type is `ArrayList<Pulse>` due to the need for both kinds to be returnable. The parser functions through the use of a regular expression as a `Scanner` delimiter to split the input string into a stream of tokens, which the parser then validates for expected properties and constructs pulses from. The parser is not error-tolerant: an unexpected token will result in a failure to parse the string.

- Pulse `Pulse` represents a generic cardiac pulse in a pacemaker or heart pulse sequence. This is an abstract class which provides common functionality for `InPulse` and `OutPulse`: it is intended for extension by `InPulse` and `OutPulse`. It provides accessor methods for the common fields (time and chamber) but does not override `toString()`.

- InPulse `InPulse` represents an input pulse in a heart pulse sequence. An input pulse consists of a pulse type (always <PULSE>), a heart chamber to which it applies (either <ATRIA>or <VENTRICLE>), an activity

level representing feedback from an accelerometer indicating how active the pacemaker's user is (and hence how fast their pulse rate must be) and an occurrence time in milliseconds since the start of the simulation. `InPulse` provides accessor methods for these properties and an overridden `toString()` method which will return a `String` representation of the format used in a scenario argument file.

- OutPulse `OutPulse` represents an output pulse in a pacemaker pulse sequence. An output pulse consists of a pulse type (either <PULSE>or <TRI_PULSE>), a heart chamber to which it applies (either <ATRIA>or <VENTRICLE>), and an occurrence time in milliseconds since the start of the simulation. `OutPulse` provides accessor methods for these properties and an overridden `toString()` method which will return a `String` representation of the format used in a returned result.

- GUserInterface `GUserInterface` is the class responsible for construction and maintenance of the Graphical User Interface for the validation software, and the handling of events within it. A `Coordinator` object which calls the constructor must pass a reference to itself as a parameter to the `GUserInterface`, which will use it to pass instructions on to the `Coordinator` object when events are encountered. Event listeners, when triggered, call a method in the `Coordinator` or `DisplayPanel` or show a dialog requesting further information then pass this information on to the Coordinator. The graphing of results is delegated to `DisplayPanel`.

- DisplayPanel The `DisplayPanel` is a custom JPanel for displaying pacemaker traces. Its sole tasks are to plot the scenario and result pulse sequences onto the display pane and maintain the display's accuracy. The graph scale is represented by a scale factor, which is calculated by dividing the length of the simulation by the current size of the display pane. Each pulse's occurrence time is multiplied by this scale factor and drawn onto the graph at the resulting place. The `DisplayPanel` draws pulses differently according to whether they are atrial or ventricular, triggered or independent.

- Coordinator The `Coordinator` is the central class responsible for coordinating the activities of the other components in order to produce a functioning validation GUI. The class sets up the various parts of the GUI when instantiated, and passes information between the `GUserInterface` and `VDMCommunicator`.

Chapter 6

Evaluation and Conclusions

In this section the objectives outlined in Chapter 3 are examined and the measure of success in meeting them is indicated along with evidence of their successful completion (where applicable). More general objectives are also considered along with general reflection on the successes and failures of the project, what should have been carried out differently and what areas of further work are suggested.

Dedicating the initial stages of project to research into the language and the API was useful as it gave a good grounding for use later in the project, however this work phase lasted too long and implementation was begun later than necessary for a high level of completion to be possible.

Splitting the implementation into a CORBA-interfacing backend and a GUI frontend was successful as the fixed requirements and relatively simple yet critical implementation of the backend made it an excellent candidate for initial development; once this was stable and functioning correctly the implementation of the GUI could proceed more smoothly. It was very rarely necessary to perform any work on the backend after the first implementation phase which allowed most of the implementation time to be dedicated to work on the GUI.

The weekly project meetings were useful to the progression of the project as the interim deadlines made it easier to determine when a stage was taking longer than it should, and the regular feedback on the GUI design enabled the rapid identification of deficiencies in the GUI design, which enabled the scheduling of work to remedy them.

The evolutionary development approach followed in the development of the GUI was useful in that it enabled the early construction of GUI prototypes with functionality capable of evaluation, which allowed the identification of some points which could be usefully changed. However, this process was also slow and, coupled with the lack of testing throughout the implementation, resulted in a failure to implement the extension functionality.

Testing was not carried out early enough or often enough, with the result that later-stage testing could not be built on top of existing basic foundations but had to be constructed from scratch, which was slow, time-consuming and

prone to reworking after tests that revealed errors in the code. In addition, the models themselves were not fully tested before implementation began which resulted in defects being revealed during the implementation of the validation GUI itself. Such errors could be difficult to distinguish from errors in the GUI and sometimes required reworking of either the GUI code or the model code. No time was allowed for this in project planning, which contributed to the time management problem. It is worth noting however, that the need to make changes to the models themselves aided the meeting of the objectives due to providing more experience and hence familiarity with VDM++.

During the design phases of the CORBA-interface backend, too much time was spent researching CORBA itself, most of which was of tangential relevance to the problem at hand. Far too much time was also devoted to getting the CORBA nameservers to work, despite these being unnecessary due to the API's getVDMToolkit method using the stringified object reference fallback approach if the CORBA name service was not found or was not helpful. etc. It was not necessary to attempt to fix this.

The total amount of work represented by the objectives outlined for this project was very high, and it is considered likely that the requirements were not wholly met largely because there was too much work to be successfully carried out during a project of such short duration. The objectives are considered in more detail below:

Objectives:

- To produce a requirements specification for the interface and and a description of the actors who will use it and their capabilities.

The requirements specification for the interface was produced and can be seen in Chapter 4. The description of the actors who will use the system and their capabilities has not been carried out, partly due to time constraints and partly due to the lack of suitable example users.

These objectives were met:

1. To produce an interface in Java to allow communication between the VDMTools interpreter and the graphical interface via CORBA.

2. To design and implement a Graphical User Interface in Java which connects to and manipulates the Pacemaker models via the interface above.

3. To become sufficiently familiar with the VDMTools application and its CORBA API to be able to use them effectively.

4. To become sufficiently familiar with VDM++ to be able to comprehend, interact with, and if necessary modify, the models.

5. To contribute to the Pacemaker model development effort.

Evidence for the meeting of Objective 1 can be found in the `VDMCommunicator` class, which carries out this task. As the `VDMCommunicator` class functions correctly, it is also considered to be evidence for the meeting of Objective 3. The

`validationgui` package itself is evidence for the meeting of Objective 2. As the `VDMCommunicator` class communicates correctly with the models, and as it was necessary to perform minor modifications to the model to enable the `VDMCommunicator` class to retrieve the results of a scenario execution, Objective 4 is considered to have been met. Objective 5 is also considered to have been met as the validation GUI has been implemented and is a significant addition to the Pacemaker model project.

These objectives were not met:

1. To extend this Graphical User Interface during a series of iterative development cycles consisting of evaluation, design and testing, providing extension functionality in each cycle.

2. To design and implement an application which allows the user to build argument files for the Pacemaker models via a Graphical User Interface.

3. To extend this application during a series of iterative development cycles consisting of evaluation, design and testing, providing extension functionality in each cycle.

The objectives were not met as it was found that the time available was insufficient to implement them. They are considered to be a good candidates for future work.

1. To demonstrate the Graphical User Interface's functionality on a collection of different scenarios and several different models.

The GUI is in a suitable state for demonstration.

What was learned: During the project, a lot was learned about VDM++ and a passing familiarity with CORBA was reached. The implementation gave experience of Java's Exception handling mechanisms and most importantly, it was found that the parts of the project that did not go to plan reinforced the need to document all stages of the process.

If the project were repeated, this author would:

- Begin the project sooner. Project selection activities could have been carried out far earlier than they were and the late start to the project resulted in an already limited amount of available time being reduced still further.

- Allow more time for writing up, and start it earlier. Writing up should have begun as soon as sufficient preliminary work had been carried out to make dissertation planning possible. Planning and implementation documentation, both formal and working notes, should have been kept neatly organised with a view to incorporating relevant material into the dissertation, and proper notes should have been kept from background reading.

- Test early, test often. Test plans should have been written prior to the implementation of the units they were testing, and regression testing should

have been carried out throughout. This approach would have saved time and resulted in far better test coverage.

- Focus on core requirements, without losing sight of need to plan architecture around potential extensions. An attempt to complete too many requirements can easily result in the completion of fewer of them, due to the time spent attempting to keep all requirements moving. The requirements should be worked on in order of priority, with only enough work carried out on the extension functionality to keep its integration into the system feasable.

Further work. The extension functionality intended for the validation GUI includes building zoom functionality into the DisplayPanel to enable the user to view the scenario and resulting data at different scales. Other useful features which were not incorporated due to time constraints include the help and preferences menus, the distinguishing of triggered pulses from independently-generated pulses and the highlighting of the pulse which is currently selected in the pulse list. Incorporating some or all of these features into further work would be an advantage.

It was also envisaged that the pulse trace could be animated: there are two possible aims here. The first is to take the data retrieved from the current models (which is returned at the end of the simulation) and replay it in order to animate the pulse traces. The other forms part of a grander ambition: it is hoped that future versions of the models will be able to support continuous simulation with manipulation carried out at a lower level. For example, the user could set up a simulation in which a pacemaker in a certain mode was activated and left to respond to stimuli from a simulated heart, with the user able to manipulate the stimuli received by the pacemaker to see how the model reacts to the changes. This is not possible with the existing models, which require a fixed time and a complete set of input data to be present at the beginning of the simulation. The modifications to the model needed to produce the continuously-running behaviour and the subsequent adaptation (or replacement) of the validation GUI to suit the new purpose would make a good future project.

The second application in the objectives for this project was not reached at all: the scenario builder GUI. This would also be a good target for a future project.

Bibliography

[1] Bicarregui, Fitzgerald, Lindsay, Moore, and Richie. *Proof in VDM: A Practitioner's Guide*. Springer-Verlag, 1994.

[2] Barry W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[3] Hugo Daniel dos Santos Macedo. Understanding and validating pacemaker requirements. Draft report for MSc project., 2007.

[4] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object-oriented Systems*. Springer-Verlag, 2005.

[5] Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7:11–19, 1990. Issue 5.

[6] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall, second edition, 1989.

[7] Software Quality Research Laboratory. Pacemaker formal methods challenge. `http://www.cas.mcmaster.ca/sqrl/pacemaker.htm(Accessed24/08/07)`.

[8] Roger S Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, sixth edition, 2005.

[9] F.D Rolland. *Programming with VDM*. Macmillan Computer Science Series. Macmillan, 1992.

[10] I. Sommerville. *Software Engineering*. Addison-Wesley, eighth edition, 2007.

Appendix A

Source code

A.1 Configuration

```
package validationgui;
import java.util.Scanner;
import java.io.File;
import java.io.FileWriter;

/**
 * Represents the configuration data for the validation GUI. Provides
 * accessor and modifier methods for the attributes and writes the
 * changes to the configuration file.
 *
 * The attributes are:
 *  - The default model: the model to be loaded when the GUI is initialised.
 *  - The ORBhost: the hostname of the CORBA naming service used to find the toolbox. If
 *  - The ORBport: the port number on which the CORBA name server listens.
 *
 * @author Emma Nicholls
 * @version 0.1
 */
public class Configuration
{
    private String filename; // The configuration filename.
    private String defModel; // Default model filename and path.
    private String orbHost;  // Leave to VDMToolkit to determine validity.
    private String orbPort;  // ditto.

    /**
     * Constructs a new Configuration object, obtaining the field data
     * from the given filename. Throws an exception if the file cannot
```

```
 * be read or is in an invalid format.
 * @param fname the absolute path to the configuration file.
 */
Configuration(String fname) throws Exception
{
    filename = fname;
    try
    {
        Scanner s = new Scanner(new File(filename));
        if (s.hasNextLine())
        {
            defModel = s.nextLine();
            if (s.hasNextLine())
            {
                orbHost = s.nextLine();
            }
            else throw new Exception();
            if (s.hasNextLine())
            {
                orbPort = s.nextLine();
            }
            else throw new Exception();
        }
        else throw new Exception();
    }
    catch(Exception e)
    {
        throw new Exception("Unable to read configuration file: using default settin
    }
}

/**
 * Default constructor. Initialises fields to:-
 * Default model: "Models/DRTPacemaker/Pacemaker.prj"
 * ORB hostname: "localhost"
 * ORB port: "2809"
 */
Configuration()
{
    defModel = "Models/DRTPacemaker/Pacemaker.prj";
    orbHost = "localhost";
    orbPort = "2809";
}

/**
 * Sets the default model to the model specified by the given path
```

```
 * and filename. If a valid filename is supplied, the configuration
 * file is updated.
 * @param path the absolute path to the model's project file.
 */
void setDefaultModel(String path) throws Exception
{
    defModel = path;
    writeToFile();
}

/**
 * Sets the name service hostname to the given hostname and updates
 * the configuration file.
 * @param host the hostname on which to look for the CORBA nameservice.
 */
void setOrbHost(String host) throws Exception
{
    orbHost = host;
    writeToFile();
}

/**
 * Sets the name service port to the given port number and updates
 * the configuration file.
 * @param port the port on which to look for the CORBA nameservice.
 */
void setOrbPort(String port) throws Exception
{
    orbPort = port;
    writeToFile();
}

/**
 * Returns the model which will be loaded by default when a
 * VDMCommunicator is constructed.
 * @return the default model.
 */
String getDefaultModel()
{
    return defModel;
}

/**
 * Returns the host on which the CORBA nameservice is expected to be.
 * The default value is localhost.
 * @return the nameserver's hostname.
```

```
     */
    String getOrbHost()
    {
        return orbHost;
    }

    /**
     * Returns the port on which the CORBA nameservice is expected to be.
     * The default value is 900.
     * @return the port number.
     */
    String getOrbPort()
    {
        return orbPort;
    }

    private void writeToFile() throws Exception
    {
        FileWriter out = new FileWriter(filename);
        out.write(defModel + "\n");
        out.write(orbHost + "\n");
        out.write(orbPort + "\n");
        out.flush();
    }
}
```

A.2 Coordinator

```
package validationgui;
import java.util.List;
import java.util.ArrayList;
import javax.swing.JOptionPane;
/**
 * Write a description of class Coordinator here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class Coordinator {
    private VDMCommunicator vcom; // The VDM Communicator object
    private String model;    // The current pacemaker model
    private String paceMode; // The current pacemaker mode
    private List<String> dataLog; // The log of communication with the model
    private Scenario curScen; // The current scenario.
    private Configuration conf; // The Configuration object.
```

```
    private GUserInterface gui; // the GUI.

    /**
     * Loads the Configuration from file, loads the VDMCommunicator and
     * orders the building of the GUI.
     */
    Coordinator() {
        try {
            conf = new Configuration("vdmgui.cfg");
        }
        catch(Exception e) {
            conf = new Configuration();
            handleError(e);
        }
        try {
            vcom = new VDMCommunicator(conf);
        }
        catch(Exception e) {
            handleError(e);
        }
        dataLog = new ArrayList<String>();
        model = conf.getDefaultModel();
        paceMode = "<OFF>";
        curScen = null;
        gui = new GUserInterface(this);
    }

//    static void initialise() {}

    /**
     * Better to have this ask if you should still exit?
     */
    void shutdown ()
    {
        try {
            vcom.shutdown();
        }
        catch(Exception e)
        {
            handleError(e);
        }
        System.exit(0);
    }

    List<String> getLogData()
    {
```

```
    return dataLog;
}

String getMode()
{
    return paceMode;
}

void setMode(String mode)
{
    paceMode = mode;
}

void setModel(String fpath)
{
    model = fpath;
}

void loadScenario(String filename)
{
    try {
        curScen = new Scenario(filename);
    }
    catch(Exception e) {
        handleError(e);
    }
    gui.loadScenario(curScen);
}

void runScenario() {
    try {
        if (model != null && curScen != null) {
            List<Pulse> result = vcom.runScenario(model, curScen.getFileName(), pacel
            gui.plotResult(result);
        }
    }
    catch (Exception e) {
        handleError(e);
    }
}

String getModelName() {
    return model;
}

public static void main(String[] args) {
```

```
        Coordinator c = new Coordinator();
    }

    private void handleError(Exception e) {
        if (gui != null) {
            gui.showErrorDialog("Error: " + e);
        }
        else System.out.println("Error: " + e);
    }
}
```

A.3 DisplayPanel

```
package validationgui;
import java.awt.*;
import java.awt.geom.Line2D;
import java.util.List;
import java.util.ArrayList;
import javax.swing.JPanel;
/**
 * Custom panel for displaying pacemaker traces.
 *
 * @author Emma Nicholls
 * @version 0.1
 */
class DisplayPanel extends JPanel {
    private Dimension size;
    private List<Pulse> scenario = new ArrayList<Pulse>();
    private List<Pulse> result = new ArrayList<Pulse>();
    private int scenLength = 0;

    /**
     * Constructs a new DisplayPanel with the given preferred size.
     */
    DisplayPanel(Dimension d) {
        size = d;
        this.setPreferredSize(d);
        this.setOpaque(true);
        this.setBackground(new Color(230,230,147));
        this.setVisible(true);
    }

    void plotScenario(Scenario s) {
        clear();
        scenLength = s.getTime();
```

```
    scenario = s.getPulseSequence();
    repaint();
}

void plotResult(List<Pulse> result) {
    this.result = result;
    repaint();
}

void highlightPulseTime(int t) {}

void clear() {
    scenario.clear();
    result.clear();
    scenLength = 0;
}

/**
 * Overridden Component method.
 */
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D) g;
    int w = this.getWidth();
    int h = this.getHeight();
    double halfh = h / 2.0;
    double decih = h / 10.0;
    double scenHeight = halfh - decih;
    double resHeight = (double) h - decih;
    g2.draw(new Rectangle(0,0,w-1, h-1));

    g2.draw(new Line2D.Double(0.0, scenHeight, (double) w, scenHeight)); // scenario
    g2.draw(new Line2D.Double(0.0, resHeight, (double) w, resHeight)); // result tra
    g2.drawString("Natural heart trace:",2,12);
    g2.drawString("Paced heart trace:",2,(int) scenHeight + 15);

    double scaleFactor = (double) scenLength / (double) w;
    for (Pulse p : scenario) {
        double pos = p.getOccurrenceTime() / scaleFactor;
        if (p.getChamber().equals("<ATRIA>")) {
            g2.draw(new Line2D.Double(pos, scenHeight, pos, scenHeight - (halfh/1.5)
        }
        else {
            g2.draw(new Line2D.Double(pos, scenHeight, pos, scenHeight - (halfh/1.5)
        }
    }
```

```
        for (Pulse p : result) {
            double pos = p.getOccurrenceTime() / scaleFactor;
            if (p.getChamber().equals("<ATRIA>")) {
                g2.draw(new Line2D.Double(pos, resHeight, pos, resHeight - (halfh/1.5) +
            }
            else {
                g2.draw(new Line2D.Double(pos, resHeight, pos, resHeight - (halfh/1.5) +
            }
        }
    }
}
```

A.4 GUserInterface

```
package validationgui;
import java.util.List;
import javax.swing.*;
import javax.swing.filechooser.FileFilter;
import java.awt.Dimension;
import java.awt.event.*;
import java.io.File;

/**
 * Constructs and maintains the Graphical User Interface for the
 * validation software.
 *
 * @author Emma Nicholls
 * @version 0.1
 */
public class GUserInterface {
    private DisplayPanel dp; // The DisplayPanel showing the heart/pace traces.
    private Coordinator parent; // The invoking Coordinator object.
    private JFrame frame; // The top level frame.
    private JLabel timeLabel; // The JLabel displaying the length of the scenario.
    private JLabel scenNameLabel; // The JLabel displaying the name of the scenario.
    private JLabel modelNameLabel; // The JLabel displaying the name of the current mode
    private JComboBox mode; // The drop-down menu allowing selection of a mode.
    private JList sPulseList; // The JList displaying the pulses in the current scenario
    private JList rPulseList; // The JList displaying the pulses in the results from the
    private DefaultListModel sListModel; // The ListModel for the scenario JList.
    private DefaultListModel rListModel; // The ListModel for the result JList.
    private Scenario curScen; // The scenario currently being displayed.
    private static final String[] modeList = {"<OFF>","<AOO>","<AOOR>","<AAT>","<DOO>"};
```

```
GUserInterface(Coordinator p) {
    parent = p;
    curScen = null;

    frame = new JFrame();
    frame.setContentPane(new Box(BoxLayout.Y_AXIS));
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // need to change so handl

    JMenu file = new JMenu("File");
    JMenuBar m = new JMenuBar();
    frame.setJMenuBar(m);
    m.add(file);

    JMenuItem lmodel = new JMenuItem("Load Model");
    lmodel.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            chooseModel();
        }
    });
    file.add(lmodel);

    JMenuItem lscen = new JMenuItem("Load Scenario");
    lscen.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            chooseScenario();
        }
    });
    file.add(lscen);

    JMenuItem pref = new JMenuItem("Preferences");
    pref.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            editPreferences();
        }
    });
    file.add(pref);

    JMenuItem exit = new JMenuItem("Exit");
    exit.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            parent.shutdown();
        }
    });
    file.add(exit);

    buildGUI();
```

```
}

void loadScenario(Scenario s) {
    curScen = s;
    scenNameLabel.setText(s.getName());
    rListModel.clear();
    updateList(sListModel,s.getPulseSequence());
    dp.plotScenario(s);
}

void plotResult(List<Pulse> r) {
    updateList(rListModel,r);
    dp.plotResult(r);
}

void highlight(Pulse p) {
    dp.highlightPulseTime(p.getOccurrenceTime());
}

void showErrorDialog(String message) {
    JOptionPane.showMessageDialog(frame,message);
}

private void buildGUI() {

    dp = new DisplayPanel(new Dimension(500,200));
    frame.add(dp);

    // Scenario section - Layers indicated by letter (eg. a - top layer,
    // b - next layer down, etc. 'l' is left, 'r' is right. 'm' is middle
    Box a = new Box(BoxLayout.X_AXIS); // Holds scenario information
    Box bl = new Box(BoxLayout.Y_AXIS); // Holds scenario overview info
    Box br = new Box(BoxLayout.Y_AXIS); // Holds pulse sequence(s)
    Box blcm = new Box(BoxLayout.Y_AXIS); // Holds name and time info.
    Box blcb = new Box(BoxLayout.X_AXIS); // Holds mode and run items.

    blcm.setMaximumSize(new Dimension(150,90));

    // Put framework together
    bl.add(blcm);
    bl.add(blcb);
    a.add(bl);
    a.add(br);
    frame.add(a);

    // Add important bits to framework
```

```
String scenName = "No scenario loaded.";
int scenTime = 0;
if (curScen != null) {
    scenName = curScen.getName();
    scenTime = curScen.getTime();
}

modelNameLabel = new JLabel("Model: " + getModelName(parent.getModelName()));
modelNameLabel.setMaximumSize(new Dimension(150,20));
blcm.add(modelNameLabel);
scenNameLabel = new JLabel("Scenario: " + scenName);
scenNameLabel.setMaximumSize(new Dimension(150,20));
blcm.add(scenNameLabel);
timeLabel = new JLabel("Length: " + scenTime + "ms.");
timeLabel.setMaximumSize(new Dimension(150,20));
blcm.add(timeLabel);

mode = new JComboBox(modeList);
mode.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        JComboBox cb = (JComboBox)e.getSource();
        parent.setMode((String) cb.getSelectedItem());
    }
});
mode.setMaximumSize(new Dimension(100,30));
blcb.add(mode);
JButton run = new JButton("Run");
run.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        parent.runScenario();
    }
});
blcb.add(run);

sListModel = new DefaultListModel();
sPulseList = new JList(sListModel);
sPulseList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
JScrollPane ssp = new JScrollPane(sPulseList);
ssp.setColumnHeaderView(new JLabel("Scenario pulses:",SwingConstants.LEFT));
ssp.setMaximumSize(new Dimension(250,100));
br.add(ssp);

rListModel = new DefaultListModel();
rPulseList = new JList(rListModel);
rPulseList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
JScrollPane rsp = new JScrollPane(rPulseList);
```

```
    rsp.setColumnHeaderView(new JLabel("Result pulses:",SwingConstants.LEFT));
    //rsp.setMaximumSize(new Dimension(250,100));
    br.add(rsp);

    frame.pack();
    frame.setVisible(true);
}

private void chooseModel() {
    JFileChooser chooser = new JFileChooser();
    int returnVal = chooser.showOpenDialog(frame);
    if(returnVal == JFileChooser.APPROVE_OPTION) {
        modelNameLabel.setText("Name: " + getModelName(chooser.getSelectedFile().get
        parent.setModel(chooser.getSelectedFile().getAbsolutePath());
    }
}

private void chooseScenario()
{
    JFileChooser chooser = new JFileChooser();
    chooser.setFileFilter(new ArgFileFilter());
    int returnVal = chooser.showOpenDialog(frame);
    if(returnVal == JFileChooser.APPROVE_OPTION) {
        parent.loadScenario(chooser.getSelectedFile().getAbsolutePath());
        scenNameLabel.setText("Name: " + curScen.getName());
        timeLabel.setText("Length: " + curScen.getTime() + "ms.");
    }
}

private String getModelName(String modelPath) {
    int start = 0;
    int end = 0;
    if (modelPath.contains("/")) {
        end = modelPath.lastIndexOf("/"); // First / from end.
        start = modelPath.lastIndexOf("/", end-1); // Second / from end.
    }
    else if (modelPath.contains("\\")) {
        end = modelPath.lastIndexOf("\\"); // First \ from end.
        start = modelPath.lastIndexOf("\\", end-1); // Second \ from end.
    }
    else end = modelPath.length();
    return modelPath.substring(start + 1,end);
}

private void updateList(DefaultListModel lm, List<Pulse> ps) {
    lm.clear();
```

```
        if (ps.size() == 0) lm.addElement("--None--");
        for (Pulse p : ps) {
            lm.addElement(p.toString());
        }
    }

    private void editPreferences() {}

    class ArgFileFilter extends FileFilter {
        public boolean accept(File f) {
            if (f.isDirectory()) {
                return true;
            }

            String ext = getExtension(f);
            if (ext != null) {
                if (ext.equals("arg"))
                {
                    return true;
                }
                else return false;
            }
            return false;
        }

        String getExtension(File f) {
            String fname = f.getName();
            String ext = null;
            int i = fname.lastIndexOf('.');

            if (i > 0 && i < fname.length() - 1) {
                ext = fname.substring(i+1).toLowerCase();
            }
            return ext;
        }

        public String getDescription() { return "ARG files"; }
    }
}
```

A.5 InPulse

```
package validationgui;
/**
 * Represents an input pulse in a pacemaker or heart pulse sequence.
```

```
 *
 * @author Emma Nicholls
 * @version 0.1
 */
class InPulse extends Pulse
{
    private final int activityLevel; // The accelerometer activity level.

    /**
     * Constructs a new InPulse with the given chamber, activity level
     * and time interval. IllegalArgumentException thrown for invalid
     * parameter values.
     * @param c the chamber - must be one of "&lt;ATRIA>" or "&lt;VENTRICLE>".
     * @param al the activity level - must be in range {1..7}.
     * @param t the occurrence time - must be positive.
     */
    InPulse(String c, int al, int t) throws IllegalArgumentException {
        super(c,t);
        if (al > 0 && al < 8) {
            activityLevel = al;
        }
        else throw new IllegalArgumentException();
    }

    /**
     * Returns the activity level, which will be an integer between 1 and 7.
     * @return the activity level
     */
    int getActivityLevel() {
        return activityLevel;
    }

    /**
     * Returns a string representation of the pulse in the format used
     * in the scenario argument files.
     * @return the string representation of the pulse
     */
    public String toString() {
        return "(<PULSE>," + getChamber() + "," + activityLevel + "," + getOccurrenceTim
    }
}
```

A.6 OutPulse

```
package validationgui;
```

```
/**
 * Represents an output pulse in a pacemaker or heart pulse sequence.
 *
 * @author Emma Nicholls
 * @version 0.1
 */
class OutPulse extends Pulse
{
    String pulseType;
    /**
     * Constructs a new OutPulse with the given chamber and time interval.
     * IllegalArgumentException thrown for invalid parameter values.
     * @param p the pulse type - must be one of "&lt;PULSE>" or "&lt;TRI_PULSE>".
     * @param c the chamber - must be one of "&lt;ATRIA>" or "&lt;VENTRICLE>".
     * @param t the occurrence time - must be positive.
     */
    OutPulse(String p, String c, int t) throws IllegalArgumentException {
        super(c,t);
        if (!(p.equals("<PULSE>") || p.equals("<TRI_PULSE>"))) throw new IllegalArgument
        pulseType = p;
    }

    /**
     * Returns the pulse type. This will be one of "&lt;PULSE>" or "&lt;TRI_PULSE>"
     */
    String getPulseType() {
        return pulseType;
    }

    /**
     * Returns a string representation of the pulse in the format used
     * in the scenario argument files.
     * @return the string representation of the pulse
     */
    public String toString() {
        return "(" + getPulseType() + "," + getChamber() + "," + getOccurrenceTime() + "
    }
}
```

A.7 Pulse

```
package validationgui;
/**
 * Represents a pulse in a pacemaker or heart pulse sequence.
 * Abstract class to be extended by InPulse and OutPulse. Provides
```

```
 * common functionality.
 *
 * @author Emma Nicholls
 * @version 0.1
 */
abstract class Pulse
{
private final String chamber; // The heart chamber the pulse applies to.
private final int occTime; // The time at which the pulse occurs.

/**
 * Constructs a new Pulse with the given chamber and time interval.
 * IllegalArgumentException thrown for invalid parameter values.
 * @param c the chamber - must be one of "&lt;ATRIA>" or "&lt;VENTRICLE>".
 * @param t the occurrence time - must be positive.
 */
    Pulse(String c, int t) throws IllegalArgumentException {
        if ((c.equals("<ATRIA>") || c.equals("<VENTRICLE>"))
            && (t >= 0))  {
            chamber = c;
            occTime = t;
        }
        else throw new IllegalArgumentException();
    }

    /**
     * Returns the chamber name, which will be one of "&lt;ATRIA>" or "&lt;VENTRICLE>".
     * @return the chamber
     */
    String getChamber() {
        return chamber;
    }

    /**
     * Returns the occurrence time, which will be a positive integer.
     * @return the occurrence time
     */
    int getOccurrenceTime() {
        return occTime;
    }
}
```

A.8 PulseParser

```
package validationgui;
```

```
import java.util.List;
import java.util.ArrayList;
import java.util.Scanner;

/**
 * Generic parser for obtaining Pulses from either input or output data.
 *
 * @author Emma Nicholls
 * @version 0.1
 */
public class PulseParser
{
    private Scanner s;
    private ArrayList<Pulse> pulses;
    private String dt;

    public PulseParser(String dt) throws Exception {
        if (!(dt.equals("Input") || dt.equals("Output"))) throw new Exception("Data type
        this.dt = dt;
    }

    /**
     * Parses the given String to produce either input or output pulses
     * as specified by the input type parameter. Returns a List of Pulses.
     * A data type of "Input" will result in a List of InPulses, a data type
     * of "Output" will result in a List of OutPulses. If the type specified
     * does not match the input data, an exception will be thrown.
     *
     * @param ds the data String to be parsed.
     */
    public ArrayList<Pulse> parseDataString(String ds) throws Exception {
        if (ds == null) throw new Exception("Error: Parser passed null data string.");
        s = new Scanner(ds);
        s.useDelimiter("(mk_|[\\(\\)\\[\\], \\t\\r\\n])+");
        pulses = new ArrayList<Pulse>();
        while(s.hasNext()) {
            parsePulse();
        }
        return pulses;
    }

    public String getDataType() {
        return dt;
    }

    private void parsePulse() throws Exception {
```

```
        String p = s.next();
        String chamber = "";
        int act = 0;
        int time = 0;
        if (p.equals("<PULSE>") || p.equals("<TRI_PULSE>")) {

            if (s.hasNext()) chamber = s.next();
            else throw new Exception();

            if (dt.equals("Input")) {
                if (s.hasNextInt()) act = s.nextInt();
                else throw new Exception();
            }

            if (s.hasNextInt()) time = s.nextInt();
            else throw new Exception();
        }
        else throw new Exception();

        if (dt.equals("Input")) pulses.add(new InPulse(chamber,act,time));
        else pulses.add(new OutPulse(p,chamber,time));
    }


}
```

A.9 Scenario

```
package validationgui;

import java.util.List;
import java.util.Scanner;
import java.io.File;
import java.util.ArrayList;

/**
 * Represents a scenario in the system. Provides methods for accessing
 * and modifying the attributes of the scenario.
 *
 * @author Emma Nicholls
 * @version 0.1
 */
public class Scenario {
    private String filename; // The absolute path to the scenario arg file.
    private String name; // The name of the scenario.
```

```
private int time; // The length of the scenario.
private List<Pulse> pulseSequence = null; // The list of pulses in the scenario.

/**
 * Constructs a new Scenario object, obtaining the field data
 * from the given filename. The filename should be an absolute path
 * (for example "/home/user/scenario.arg").
 * @param fname the absolute path of the scenario argument file.
 */
Scenario(String fname) throws Exception {
    filename = fname;
    if (fname == null || !fname.endsWith(".arg")) throw new Exception("Invalid scena
    name = getName(fname);

    Scanner s = new Scanner(new File(filename));
    s.useDelimiter("(\\A[ \\s]*mk_[ \\s]*\\(|,[ \\s]*\\[|\\][ \\s]*\\)|[ \\s])+");

    if (!s.hasNextInt()) {   // Malformed scenario arg file.
        throw new Exception("Malformed scenario arg file: cannot parse scenario leng
    }
    else time = s.nextInt();
    if (time < 0) throw new Exception("Malformed scenario arg file: Scenario length
    String data = "";
    if (s.hasNext()) {
        data = s.next();
    }
    PulseParser pp = new PulseParser("Input");
    pulseSequence = pp.parseDataString(data);
}

/**
 * Returns the scenario name.
 * @return the scenario name.
 */
String getName() {
    return name;
}

/**
 * Returns the absolute path of the scenario's arg file, for example
 * "/home/user/scenario.arg".
 * @return the absolute path to the scenario arg file.
 */
String getFileName() {
    return filename;
}
```

```
    /**
     * Returns the length of the scenario.
     * @return the scenario length.
     */
    int getTime() {
        return time;
    }

    /**
     * Returns the pulse sequence.
     * @return the pulse sequence.
     */
    List<Pulse> getPulseSequence() {
        if (pulseSequence != null) return pulseSequence;
        else return new ArrayList<Pulse>();
    }

    /**
     * Produces a scenario name from its filename. The name will be that of the
     * argument file without its extension.
     * @param fname the scenario's argument filename.
     * @return the scenario name.
     */
    private String getName(String fname) {
        int start = 0;
        int end = fname.lastIndexOf(".");
        if (fname.contains("/")) start = fname.lastIndexOf("/");
        else if (fname.contains("\\")) start = fname.lastIndexOf("\\");
        return fname.substring(start + 1,end);
    }
}
```

A.10 VDMCommunicator

```
package validationgui;
import org.omg.CORBA.*;
import jp.co.csk.vdm.toolbox.api.*;
import jp.co.csk.vdm.toolbox.api.corba.ToolboxAPI.*;
import jp.co.csk.vdm.toolbox.api.corba.VDM.*;
import java.util.List;
import java.util.ArrayList;
import java.util.LinkedList;

/**
```

```
 * Facilitates communication with the VDM Toolkit via CORBA.
 * VDMCommunicator allows the connection to, execution of scenarios
 * upon, and disconnection from a currently-running VDMApplication.
 * Basic communication log data is also stored in the VDMCommunicator
 * object. Exceptions are thrown by all public methods, and must be
 * handled by the owning class.
 *
 * @author Emma Nicholls
 * @version 0.2
 */
class VDMCommunicator
{
    private VDMApplication app; // The Toolkit reference.
    private VDMInterpreter itp; // The Interpreter reference.
    private VDMParser pars;     // The Parser reference.
    private VDMProject proj;    // The Project reference.
    private VDMTypeChecker tc;  // The Type Checker reference.
    private LinkedList<String> commData; // The communications log.
    private short clientId; // The client's session ID.

    /**
     * Constructs a new VDMCommunicator using the CORBA host/port
     * provided by the given Configuration object, and initialises
     * it using the default model also provided by the Configuration
     * object.
     * Will throw an exception if:
     *      - Unable to connect to the toolkit.
     *      - The connection to the toolkit is lost.
     * @param c the Configuration object containing CORBA configuration
     * details.
     */
    VDMCommunicator(Configuration c) throws Exception
    {
        String[] orbcfg = new String[4];
        orbcfg[0] = "-ORBInitialPort";
        orbcfg[1] = c.getOrbPort();
        orbcfg[2] = "-ORBInitialHost";
        orbcfg[3] = c.getOrbHost();

        commData = new LinkedList<String>();
        getVDMToolkit(orbcfg);
    }

    /**
     * Loads the given model and executes the given scenario with
     * the given pacemaker mode. Returns the result of the scenario
```

```
 * as an list of Pulses. Will throw an exception if the
 * connection to the application is lost, or if the model, scenario
 * or mode are invalid.
 * @param model the absolute path of the model on which to run
 * the scenario.
 * @param argfile the path to the argument file containing the scenario
 * data.
 * @param mode the pacemaker mode to run the scenario upon.
 * @return a List of output pulses.
 */
List<Pulse> runScenario(String model, String argfile, String mode)
                                                        throws Exception
{
    loadModel(model);

    List<Pulse> l = new ArrayList<Pulse>();
    StringBuffer sb = new StringBuffer();

    if (app != null) {
        VDMGeneric result = null;
        String cmd = "create n := new World(\"" + argfile + "\"," + mode + ")";
        itp.EvalCmd(cmd);
        commData.add("Interpreter: EvalCmd(" + cmd + ")");

        result = itp.EvalExpression(clientId,"n.Run()");
        commData.add("Interpreter: EvalExpression(n.Run())");

        VDMSequence output = VDMSequenceHelper.narrow(result);
        VDMGeneric elem;
        int len = output.Length();
        for (int i=0; i<len; i++) {
            elem = output.Hd();
            output.ImpTl();
            sb.append(elem.ToAscii() + " ");
            commData.add("Retrieved: " + elem.ToAscii());
        }
        PulseParser pp = new PulseParser("Output");
        l = pp.parseDataString(sb.toString());
        itp.EvalCmd("destroy n");
        commData.add("Interpreter: EvalCmd(destroy n)");
    }
    else throw new Exception();
    return l;
}

/**
```

```
 * Returns the communications data log as a list of Strings.
 * @return the list of data log entries.
 */
List<String> getCommData() throws Exception
{
    List<String> log = new ArrayList<String>();
    while (commData.size() > 0) {
        log.add(commData.remove());
    }
    return log;
}

/**
 * Cleanly closes the connection to the Toolkit. This is done by
 * unregistering the client and closing the connection. Once closed,
 * the connection cannot be reopened - a new VDMCommunicator must
 * be constructed. Will throw an exception if communication with the
 * Toolkit is lost prior to the closure of the connection.
 */
void shutdown() throws Exception
{
    app.DestroyTag(clientId);
    commData.add("DestroyTag(" + clientId + ")");
    app.Unregister(clientId);
    commData.add("Unregister(" + clientId + ")");
}

/**
 * Obtains a reference to a running instance of VDMTools and registers
 * the client with the toolkit. This can be either vppde or vppgde. An
 * exception is thrown if the connection cannot be made.
 * @param corbargs the CORBA configuration details. Contains the hostname
 * and port number for the CORBA name service (if available).
 */
private void getVDMToolkit(String[] corbargs) throws Exception
{
    ToolboxClient toolboxClient = new ToolboxClient();
    org.omg.CORBA.Object obj =
        toolboxClient.getVDMApplication(corbargs, ToolType.PP_TOOLBOX);
    app = VDMApplicationHelper.narrow(obj);
    commData.add("Toolkit reference obtained.");
    clientId = app.Register();
    commData.add("Register()");
    app.PushTag(clientId);
    commData.add("PushTag(" + clientId + ")");
}
```

```
    /**
     * Loads the specified model and initialises the interpreter.
     * The method will throw an exception if the model cannot be loaded
     * or if communication with the Toolbox is lost.
     * @param model the absolute path of the model's project file
     * (for example "/home/user/model/ExampleModel.prj").
     */
    private void loadModel(String model) throws Exception
    {
        if (proj==null) proj = app.GetProject();
        commData.add("GetProject()");
        proj.Open(model);
        commData.add("Open(" + model + ")");
        if (pars==null) pars = app.GetParser();
        FileListHolder files = new FileListHolder();
        int count = proj.GetFiles(files);
        String flist[] = files.value;
        pars.ParseList(flist);
        if (typeCheck())
        {
            if (itp == null) itp = app.GetInterpreter();
            itp.Initialize();
        }
        else throw new Exception();
    }

    /**
     * Checks the parsed model for type correctness. Returns true if the
     * check is successful, false if errors are found. Throws an exception
     * if the check cannot be carried out.
     * @return true if the type check passes, false if not.
     */
    private boolean typeCheck() throws Exception
    {
        if (tc == null) tc = app.GetTypeChecker();
        boolean noTypeErrors = true;
        ModuleListHolder classes = new ModuleListHolder();
        int count = proj.GetModules(classes);
        String[] cl = classes.value;
        for (int i=0; i<count; i++) {
            if (!tc.TypeCheck(cl[i])) noTypeErrors = false;
        }
        return noTypeErrors;
    }
}
```