Incremental Development of a Distributed Real-Time Model of a Cardiac Pacing System using VDM

Hugo Daniel Macedo[1], Peter Gorm Larsen[2], and John Fitzgerald[3]

[1] Minho University, Portugal
[2] Engineering College of Aarhus, Denmark
[3] School of Computing Science, Newcastle University, UK
hmacedo@di.uminho.pt, pgl@iha.dk, John.Fitzgerald@ncl.ac.uk

Abstract. The construction of formal models of real-time distributed systems is a considerable practical challenge. We propose and illustrate a pragmatic incremental approach in which detail is progressively added to abstract system-level specifications of functional and timing properties via intermediate models that express system architecture, concurrency and timing behaviour. The approach is illustrated by developing a new formal model of the cardiac pacemaker system proposed as a "grand challenge" problem in 2007. The models are expressed using the Vienna Development Method (VDM) and are validated primarily by scenario-based tests, including the analysis of timed traces. We argue that the insight gained using this staged modelling approach will be valuable in the subsequent development of implementations, and in detecting potential bottlenecks within suggested implementation architectures.

1 Introduction

Formal models have a valuable role to play in validating requirements and designs for real-time distributed systems in early development stages. Rapid feedback from the analysis of such models has the potential to reduce the risk of expensive re-working as a consequence of the late-stage detection of defects. However, models that incorporate the description of functionality alongside timing behaviour and distribution across shared computing resources are themselves potentially complex. Moving too rapidly to such a complex model can increase modelling and design costs in the long run. In order to gain full value from formal modelling and analysis, a systematic approach to constructing and validating models is required.

Our current work is focussed on the development and industrial application of formal modelling techniques that satisfy the requirements discussed above. We have developed and applied technology based on the Vienna Development Method (VDM) [1,2,3] and its tool support (VDMTools [4]). Recent work has developed modelling abstractions and test-based analysis tools that support the object-oriented description of distributed real-time systems [5,6]. Our experience applying formal modelling techniques in a variety of industry sectors suggests that an approach to modelling such distributed real-time systems should permit the staged and controlled construction of a formal model, with opportunities for validation at each stage. We have proposed such a staged approach as part of the methodological guidelines accompanying the VDMTools [7].

This paper reports a study in which we have assessed the feasibility of applying an incremental approach to model construction by developing a model for an artificial cardiac pacemaker [8]. The pacemaker specification is that it includes system-level requirements affecting hardware as well as software. We demonstrate how such cross-disciplinary requirements can be introduced gradually into a model in a phased fashion, along with validation of functional and timing requirements.

The Pacemaker specification has been offered by the Software Quality Research Laboratory at McMaster University as a pilot problem in the Grand Challenge in Verified Software [9] and this paper is believed to represent the first attempt at its treatment. However, the present study does not aim to provide comprehensive coverage of the Pacemaker challenge. The intention is to use the problem to pilot the incremental method in an industrially relevant context, using the available tools, as a precursor to tackling more substantial challenges, including the full Pacemaker and other real-time systems.

We introduce the Pacemaker system in Section 2. Section 3 briefly introduces the VDM technology used, our phased approach to model construction and the tool support. The progressive development of the Pacemaker model is described, illustrated by extracts from the series of VDM models developed (Section 4). The test-based approach to validation is discussed in Section 5, including how timing conditions can be checked using this technology. Finally Sections 6 and 7 discuss related work and draw conclusions from the study.

2 The Pacemaker System and Environment

In this study, the pacemaker is treated as an embedded system operating in an environment containing the heart. We first review the elements of the environment that interact with the pacemaker (Section 2.1) and then consider the elements of the pacemaker system itself (Section 2.2).

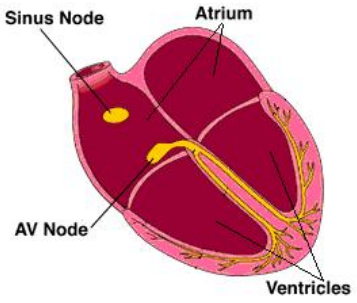

Fig. 1. The natural pacemaker

2.1 Environment: The Heart

The human heart serves as a pump for the circulatory system. It is a muscular shell around four chambers (called atria and ventricles) which contract and relax periodically under the control of natural electrical stimuli. A natural pacemaker orchestrates the functioning of the pump, discharging electrical pulses at specific points (see Fig. 1). In normal functioning, a discharge made at the sinus node subsequently reaches the atrioventricular (AV) node which amplifies it, stimulating the ventricles. If the natural pacemaker is malfunctioning, a physical condition termed Bradycardia may arise in which the heart rate falls below the level expected for the patient. To normalise the heart rate, an artificial pacemaker may be implanted to aid or replace the natural pacemaker. Physicians measure the heart's performance using, among other parameters, the bpm (beats per minute) rate of the heart. We use the term

pulse and *pulses per minute* in reference to pacemaker activity, whereas *beat* and *beats per minute* refer to heart activity.

2.2 System: Artificial Pacemaker

An artificial pacemaker (referred to subsequently as a *pacemaker* [10]) is a system composed of:

Leads: One or more wires, normally two, that both sense and discharge electric pulses.
Device: The implanted batteries and controller.
Device Controller-Monitor (DCM): An external unit that interacts with the device using a wireless connection (not modelled in this paper.)
Accelerometer: A unit inside the device measuring body motion in order to allow modulated pacing.

A typical configuration consists of one lead attached to the right atrium and another to the right ventricle. The pacemaker has several operating modes that address different malfunctions of the natural pacemaker. The specification document [8] identifies 18 operating modes controlling 26 variables and each of the variables can be configured within a value range. Most of the variables are time-related parameters, defining such properties as the interval between a pace in the atrium and the ventricle or the number of pulses per minute the device should deliver to a given chamber.

The operating modes of the device are classified using a code consisting of three or four characters. For the examples in this paper, the code elements are: chamber(s) paced ("O" for none, "A" for atrium, "V" for ventricle, "D" for both), chamber(s) sensed (same codes), response to sensing ("O" for none in this paper) and a final optional "R" to indicate the presence of rate modulation in response to the physical activity of the patient as measured by the accelerometer. "X" is a wildcard used to denote any letter (i.e. "O", "A", "V" or "D"). Thus "DOO" is an operating mode in which both chambers are paced but no chambers are sensed, and "XXXR" denotes all modes with rate modulation.

3 VDM Modelling Technology for Distributed Real-Time Systems

In our modelling work, we have used VDM [1]. Three dialects of the VDM modelling language are in use, each supporting different forms of system specification. VDM-SL [2] provides facilities for the functional specification of sequential systems with basic support for modular structuring. VDM++ [3] extends VDM-SL with features for object-oriented modelling and concurrency. VICE (VDM++ In Constrained Environments) further extends VDM++ with features for describing real-time computations [11] and distributed systems [5]. Each dialect has formally defined syntax, static and dynamic semantics which extend those of the ISO Standard VDM-SL language [12]. For a detailed introduction to VDM++, the reader is referred to the texts and the VDM Portal [13]. In the remainder of this section, we focus on the features of VDM++ and VICE that have a major role in the modelling of distributed real-time systems.

3.1 Basic VDM Notations

A model in VDM-SL, is composed of type definitions built from simple abstract types such as *bool* or *nat*, and type constructors such as sequences and records. Types may be restricted by predicate invariants. Persistent state variables may be defined. Operations that may modify the state can be defined implicitly, using pre- and postcondition expressions, or explicitly, using imperative statements. Functions are similar to operations except they may not refer to state variables, and are side-effect free.

An object-oriented model in VDM++ is composed of class specifications which may use single or multiple inheritance. The internals of each class definition are similar to those of a regular VDM-SL model. each object's persistent state consists of typed *instance variables*. Operations in VDM++ are re-entrant and their invocation is defined with synchronous (rendezvous) semantics. Operation execution may be constrained by specifying a permission predicate [14], a Boolean expression over *history counters* that acts as a guard for the operation, for example to express mutual exclusion. History counters are maintained per object to count the number of requests (`#req`), activations (`#act`) and completions (`#fin`) per operation.

VDM++ classes may be active or passive. Active classes represent entities that have their own thread of control; passive classes are always manipulated from the thread of control of another active class. A thread is a sequence of statements that is executed to completion, at which point the thread dies. The thread is created whenever the object is created but the thread needs to be started explicitly using a *start* operator. For reactive systems it is possible to specify threads that do not terminate.

Extensions to VDM++ (VICE) support the description and analysis of real-time embedded and distributed systems [5,15]. These include primitives for modelling deployment to a distributed hardware architecture and support for asynchronous communication. Two predefined classes, *BUS* and *CPU*, are available to the specifier to construct the distributed architecture in the model. User-defined classes can be instantiated and deployed on specific *CPU*s. The communication topology between the computation resources in the model can be described using the *BUS* class.

The semantics of VDM++ is extended with a notion of time such that any thread that is running on a computation resource or any message that is in transit on a communication resource can cause time to elapse. Models that contain only one computation resource are compatible to models in plain VDM++.

Operations may be specified as asynchronous in VICE, allowing the caller to resume its own thread of control after the call is initiated. A new thread is created, scheduled and started to execute the body of the asynchronous operation. Statements (*duration* and *cycles*) may be used in operation bodies to specify time delays that are, respectively independent of or dependent upon processor capacity. The time delay incurred by the message transfer over the *BUS* can be made dependent on the size of the message being transferred.

3.2 An Incremental Approach to Model Construction

Faced with the challenge of developing VDM++ models of distributed real-time systems, we have proposed a staged approach [7] which reflects the capabilities of each of the VDM modelling languages.

The analysis of informally expressed requirements leads to a first abstract model giving system-level specification of behaviour. The basic VDM-SL language is well suited to this level of description. Based on the abstract model, we introduce a static architecture, creating a sequential (i.e. non-concurrent) model with structure expressed using the features of VDM++. This model would then be extended to become a concurrent VDM++ design model. The concurrent design model itself is then extended with real-time information using the VICE extensions, and additionally distribution over processors can be described also using the VICE extensions. At this stage it may prove necessary to revisit the concurrent design model, since design decisions made at that stage may prove to be infeasible when real-time information is added to the model (for instance, the model may not be able to meet its deadlines).

The ability to validate the intermediate models developed in this process makes it possible to identify requirements and design defects at an early stage. The initial abstract model need not be directly executable, but subsequent models are likely to be so, making it possible to conduct extensive tests in order to validate design decisions. The VDMTools are intended to provide extensive support for scenario-based testing as a form of validation.

We do not claim that the models introduced at each stage in our approach are formal refinements of their predecessors, although this may sometimes be the case. Our intended output is a comprehensive model of the target system that can serve as a basis for subsequent development, possibly using refinement. We are therefore introducing detail in a staged manner, where the executions at each level might, informally, be seen as providing a finer level of granularity than its predecessor.

3.3 VDM Tool Support

VDM is supported by an industry-strength tool set, VDMTools, owned and developed by CSK Systems [16]. VDM and VDMTools have been used successfully in several large-scale industrial projects, e.g. [17,4]. The tools offer syntax, type and integrity checking capabilities, code generators, a pretty printer and an application programmer interface. The main support for model validation is by means of an interpreter allowing the execution of models written in the large executable subset of the language.

Scenarios defined by the user are essentially test cases consisting of scripts invoking the model's functionality. The interpreter executes the script over the model and returns observable results as well as an *execution trace* containing, for each event, a time stamp and an indication of the part of the model in which it appeared. A separate tool (an Eclipse plug-in) called *showtrace* has been developed for reading execution traces, displaying them graphically so that the user can readily inspect behaviour after the execution of a scenario, and thereby gain insight into the ordering and timing of exchange of messages, activation of threads and invocation of operations.

The existing tools have been further extended to allow explicit logical statements of expected system-level timing properties (termed *timing conjectures*) which can be checked against execution traces [6]. Fig. 2 shows the *showtrace* output resulting from the analysis of three validation conjectures (C1-C3) from the pacemaker study (see Section 5). The main window shows a fragment of the execution trace, with time on the horizontal axis. Processing on each architectural unit is shown by horizontal lines (colours

are used to denote thread start-up, kill and scheduling). Thin arrows indicate message passing and fat arrows indicate thread swapping. The conjectures are shown at the bottom of the window. Circular marks on the traces show conjecture violations, e.g. the circle showing a counterexample to the timing conjecture `C3`, where an event occurrence breaches an expected temporal separation.

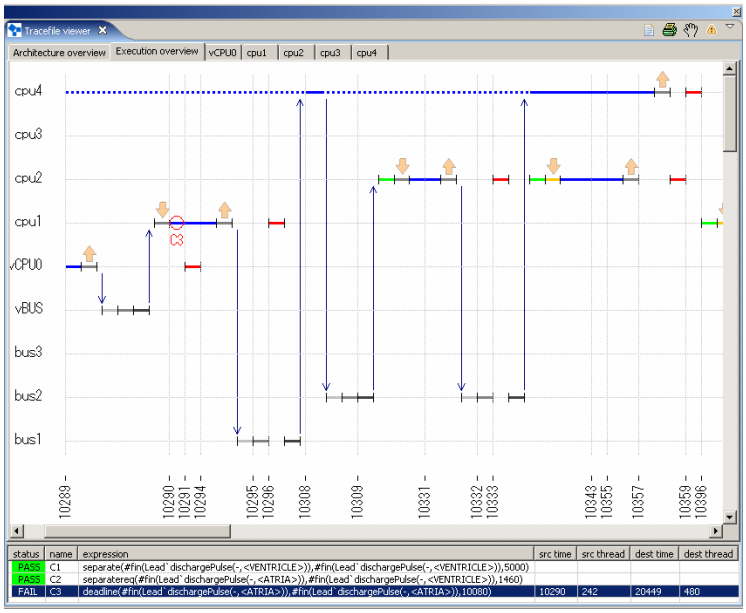

Fig. 2. Showtrace tool demonstrating validation conjecture violation

4 The Pacemaker Models

We describe the incremental development of a model of the Pacemaker challenge problem conducted in order to evaluate the incremental development approach outlined above using an industrially relevant application. At this stage, we have not attempted a comprehensive attack on the Pacemaker problem. In this context, the overall purpose of the modelling work on the Pacemaker has been to clarify and validate the system's informally stated requirements as defined at [8], from where additional tutorial material on cardiac timing cycles and pacing modes is also available. A full-scale attempt on the pacemaker challenge would also rely on extensive domain background from texts [10] and domain experts.

Following our staged approach, in order to manage the complexity of the model itself, the construction was done in four steps, each involving the construction of a new model at a lower level of abstraction from its predecessor. We will term them *Abstract*, *Sequential*, *Concurrent* and *Distributed Real-Time* (DR-T) respectively.

Of the 19 modes of the pacemaker, eight have been modelled so far, covering 18 of the 26 controlling variables. The initial *Abstract* model consists of several modules totalling 652 lines of VDM-SL. The three subsequent object-oriented models are larger but of almost equal size: the *Sequential* model is 872 lines of VDM++ and the *Concurrent* model is 879 lines whereas the *DR-T* model is 811 lines of VDM++. So the *Abstract* model is a bit smaller and simpler than the *Sequential* model. Although the sizes of the sequential, concurrent and DR-T models are similar, they get progressively more complex as they include concurrency and the distribution.

Note that we do not claim that this is a *formal* refinement process. The initial *Abstract* model is informally refined by a *Sequential* model by adding structuring information. Neither of these models the concurrency of the environment and the system; instead they simulate fixed time steps controlled from the environment. In the *Concurrent* model both the environment and the relevant parts of the system are organised with concurrent threads that are synchronized by permission predicates. In each of these three models, time is present explicitly as an abstraction whereas in the final *Distributed Real-Time* model time is implicit, allowing us to express more realistic timing behaviour while validating this model.

4.1 Abstract Model

The first model is expressed in VDM-SL, the simplest of the VDM modelling languages, lacking the object-orientation and concurrency features of VDM++. It is organised in modules, each of which corresponds to an operating mode of the pacemaker and defines a single function `PacemakerM` (where `M` is the mode), derived from the requirements. The key type and function definitions have the following form:

```
SenseTimeline = set of (Sense * Time);
ReactionTimeline = set of (Reaction * Time);
functions
PacemakerM (inp : SenseTimeline) r : ReactionTimeline
post ...
```

where `Sense` and `Reaction` are enumerated types representing the presence or absence of a pulse. Each `PacemakerM` function is expressed in VDM-SL in an implicit style by means of a postcondition characterising the events trace that should result from correct functioning of the pacemaker over an input sense trace. The implicit style is used because it is not intended that the function should be directly executed; it serves primarily as a means of clarifying requirements.

The abstract models support the formalisation of our understanding of the system requirements. For example, during construction of the abstract models it became apparent that the requirements in [8] for some modes place constraints on ventricular pace events even when the ventricle is not being paced. Domain experts later reported this is an error in the natural language specification. We also identified areas of incompleteness, for example the requirements as modelled in `PacemakerDOO` [18] do not take account of certain unstated requirements on intervals between atrium pulses.

The post-conditions of the `PacemakerM` functions have the potential to serve as test oracles on the models developed in subsequent phases, provided suitable abstraction functions are implemented. The post-condition formulations were also valuable and we

were also able to use them to help design the validation conjectures that were applied to the analysis of the final (distributed real-time) model. For example, the following post-condition from `PacemakerDOO` gives rise to the validation conjecture `C2` used on test traces and shown in Section 5.2:

```
forall mk_(<ATRIA>,ta) in set r &
   (exists mk_(<VENTRICLE>,tv) in set r & tv = ta + FixedAV);
```

4.2 Sequential and Concurrent Models

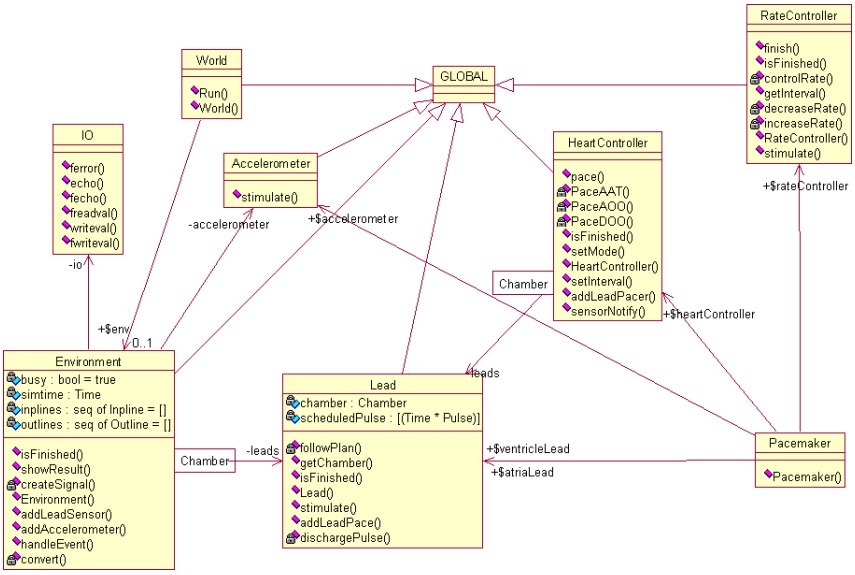

Fig. 3. A UML class diagram describing the common static structure

The sequential design model describes both the data that is to be computed, and how it is to be structured into static classes, without commitment to a specific dynamic architecture. The Pacemaker model is structured around a class `Pacemaker` coexisting with an `Environment` class in a given `World`. Figure 3 shows the classes that are common to all the VDM++ models and their associations. The diagram is derived automatically from the VDM++ models using the VDMTools link to IBM Rational Rose. In the generated class diagram, we do not show associations arising from the use of public static methods. However, the public static instance variables in the `Pacemaker` class are shown as directed associations (labelled '`+$`') sign from `Pacemaker` to the relevant target class.

The main feature of the static architecture is its division into the environment and the pacemaker system. The `Environment` class controls the production of stimuli delivered via `Leads`. The `Pacemaker` class represents the technical system. The `HeartController` class monitors incoming stimuli and generates pulses. The `RateController` class is used for rate adap-

tation control in "XXXR" operating modes, and will not be further discussed in this paper (for full details see [18,19]) and the `Accelerometer` class is coping with the motion data.

The initial sequential model is then used to analyse the system behaviour without taking concurrency issues into account. In the sequential step, the environment affects the flow-of-control by passing signals to the pacemaker system. The environment also provides the simulated time increments using an additional `Timer` class (not shown in Figure 3). This model is executable and so may be validated with respect to informal requirements by running tests through the VDM++ interpreter.

In the sequential model the environment class contains an explicit `Run` operation which operates a form of command loop, stepping through the input timeline of sensed events, delivering signals to the system (through the operation `createSignal`). Having created the signals, it calls a `Step` operation in `rateController` and later in `heartController`, the two parts of the system model that will become concurrent in subsequent models. In addition the global time variable is stepped forward.

In the third phase of model development concurrency is introduced. The static structure is maintained with the exception of a more elaborate modelling of time enabling synchronisation of time in concurrent threads [18]. The environment model is freed of the responsibility to control the system model but still controls the time. The concurrent model has the same structure as the sequential one, but the lock-step stepping mechanism is substituted by threading and synchronisation. With the introduction of concurrency the need for synchronisation arises and this is achieved through the use of VDM++ permission predicates. We claim that the main benefit of this model is to study and find possible concurrency issues, for instance, we have found some race conditions caused by errors introduced in our specification that were corrected at this level.

4.3 Distributed Real-Time Model

The final step in model construction is the introduction of distribution over CPUs in a topology determined by the configuration of a bus. Time "annotations" allow time to be built in and managed by the VDM++ interpreter so again the static structure is preserved except that the explicit modelling of time now disappears.

The deadlines and periods in the requirements [8] are stated in milliseconds and in certain cases in fractions of milliseconds. We therefore use 0.1 ms as the unit of time below.

The `Environment` class now only delivers signals to the system as a periodic thread:

```
thread periodic (1000,10,900,0) (createSignal);
```

The operation `createSignal` will be invoked approximately every 1000 time units and a jitter of up to 10 time units can be allowed for the periodic invocation of this thread. The third parameter indicates that there is going to be at least 900 time units between two wake-ups of this periodic thread. Finally, the last parameter indicates that no offset is required for the invocation of it. This is a feature that is most valuable if a number of threads are started at the same time, and there is a desire to carry them out in a special order.

The `createSignal` operation looks at the remaining input events in the input lines (`inplines`) which each have a time associated with them. If this time has been reached for one or more of the events the relevant leads and accelerometer must be notified. It is defined as:

```
createSignal : () ==> ()
createSignal () ==
   (if len inplines > 0
```

```
 then (dcl curtime : Time := time,
           done : bool := false;
       while not done do
          let mk_(sensed,chamber,accinfo,stime) = hd inplines
          in if stime <= curtime
             then
             (leads(chamber).stimulate(sensed);
              accelerometer.stimulate(accinfo);
              inplines := inplines(2,...,len(inplines));
              done := len inplines = 0
             )
             else done := true
      );
 if len inplines = 0 then busy := false;
);
```

Expressing Time Requirements Time "annotations" are used to record the durations allocated to particular actions. For example, consider the requirement that "The Atrial pulse width must be 0.4 milliseconds." [8], appendix. This is a requirement on the operation that describes the discharge of a pulse. The **duration** statement of VDM++ is used to specify the width (40 in the 0.1 ms units).

```
private async
dischargePulse : Pulse * Chamber ==> ()
dischargePulse (p,c) ==
   if(c = <ATRIUM>)
   duration(40) World`env.handleEvent(p,c,time);
```

Notice also the use of **async** so that the caller of this operation will not block waiting for it to terminate.

Modelling the System Distribution The system class defines the distribution architecture. Instance variables are defined as follows (all declared public static) to define the physical objects present in the system architecture:

```
atriumLead      : Lead            := new Lead(<ATRIUM>);
ventricleLead   : Lead            := new Lead(<VENTRICLE>);
accelerometer   : Accelerometer   := new Accelerometer();
rateController  : RateController  := new RateController();
heartController : HeartController := new HeartController();
```

The architectural components in the model are, for the purposes of illustration, four CPUs. Each CPU definition in the model indicates the scheduling policies (Fixed Priority (FP) and First-Come-First-Served (FCFS) respectively) and processor capacity (number of cycles per unit time). For the CPUs that will just run a single thread we used a FCFS scheduling algorithm. For the case of CPU4 that contains both the `rateController` and `heartController` threads we opted for a fixed priority policy, assuming that the stimulation of the patient heart is more important than adjusting the rate of the stimulation.

```
cpu1 : CPU := new CPU(<FCFS>,3E2);
cpu2 : CPU := new CPU(<FCFS>,3E2);
```

```
cpu3 : CPU := new CPU(<FCFS>,3E2);
cpu4 : CPU := new CPU(<FP>  ,3E2);
```

In order to define the communication topology, we create three bus objects linking the specified CPUs with a certain bandwidth (`1E6`) and the chosen network control protocol, in this case `FCFS`.

```
bus1 : BUS := new BUS(<FCFS>,1E6,{cpu1,cpu4});
bus2 : BUS := new BUS(<FCFS>,1E6,{cpu2,cpu4});
bus3 : BUS := new BUS(<FCFS>,1E6,{cpu3,cpu4});
```

The final element in this system class is the constructor operation that deploys the functionality across the resources. Here the leads are deployed on two processors (representing the physical wires and lead controllers) and the accelerometer is deployed using the same approach. The remaining devices are deployed on `cpu4`:

```
public Pacemaker: () ==> Pacemaker
Pacemaker () ==
  (cpu1.deploy(atriumLead);
   cpu2.deploy(ventricleLead);
   cpu3.deploy(accelerometer);
   cpu4.deploy(rateController);
   cpu4.deploy(heartController);
   cpu4.setPriority(HeartController`pace,3);
   cpu4.setPriority(RateController`adjustRate,1);
  );
```

Implicitly there is always a virtual `CPU` and `BUS` where elements run that are not explicitly deployed to a `CPU`. Also the communication between objects in different CPUs with no explicit bus connecting them will occur using the virtual bus.

Whenever a `Lead` is stimulated it simply delegates this stimulus on to the statically declared `heartController` as a notification:

```
public
stimulate : Sense ==> ()
stimulate (s) ==
  Pacemaker`heartController.sensorNotify(s,chamber);
```

When the `heartController` is notified it adds the new signal that has been sensed to a mapping of sensed entries and that is then taken into account subsequently by a periodic thread determining whether a pace is necessary depending upon the mode of operation of the pacemaker.

```
public
sensorNotify : Sense * Chamber ==> ()
sensorNotify (s,c) ==
  sensed := sensed ++ {c |-> s};
```

5 Validation

5.1 Validation of Abstract, Sequential, Concurrent and DR-T Models

A systematic testing approach [3] was used to validate the models derived during the staged development process. “Validation” in this context refers to the activity of gain-

ing confidence that the formal models developed are consistent with the requirements expressed in the requirements document [8]. A comprehensive model of the full Pacemaker specification would additionally entail domain experts in both the setting of validation conjectures and the detailed review of the model; for the initial study reported here, we had only limited access to domain expertise.

Test scenarios were defined to model interesting situations such as the absence of input pulses. These were run over the several models while collecting the test coverage data for each model. Tests developed for the abstract model can be used, adapted, as regression tests in the later model development phases.

To validate the sequential model we used the re-shaped scenarios, augmenting them with new tests derived from the process of constructing and debugging of the model and its algorithmic subtleties. The validation process involves loading the chamber senses into the `Environment` which will deliver them at the correct time to the respective lead. During the simulation reactions (pulses delivered by the leads) were collected by the environment and then displayed. All of the test scenarios were reused in the validation of the *Concurrent* model and the *DR-T* model using the same paradigm.

5.2 Timing Conjectures and their Validation

The capabilities of VDMTools have been extended to support automated checking of timing-related conjectures on traces derived from runs of test scenarios over VDM++ models [6]. A simple language of standard conjecture forms has been defined and the semantics have been embedded directly into the tool set. The result of checking conjectures on a trace is displayed using the trace display format, with conjecture violation points identified as shown in Figure 2. These timing conjectures are not part of the requirements; they are assertions that the developers expect to hold over the traces derived from scenario executions. The timing conjectures analysed so far in this pilot study are naive; a fuller set would be derived from interactions with domain experts in a full-scale modelling and verification study.

The forms of timing conjecture used in the pacemaker study are: *separations*, *required separations* and *deadlines* [6]. Separation conjectures describe a minimum separation between occurrences of specified events, should the events occur. A *Separation* conjecture is a 5-tuple $separate(e_1, c, e_2, d, m)$ where e_1 and e_2 are the names of events, c is a state predicate, d is the minimum acceptable delay between an occurrence of e_1 and the next following occurrence of e_2 provided that c evaluates to true at the occurrence time of e_1. If c evaluates to false when e_1 occurs, the validation conjecture holds independently of the occurrence time of e_2. The Boolean flag m, when set to true, indicates a requirement that the occurrence numbers of e_1 and e_2 should be equal. This allows the designer to record conjectures that describe some coordination between events. The *Required separation* conjecture is similar to the separation conjecture but additionally requires that e_2 does indeed occur. The *Deadline* conjecture places a maximum delay on the occurrence of e_2. Again, the *m* option may be used to link the occurrence numbers of the events. A validation conjecture $deadline(e_1, c, e_2, d, m)$; if c holds, d is the maximum tolerable delay between e_2 and e_2.

Validation conjectures can be proposed for the test scenarios on the Distributed Real-Time model. For example, a conjecture might be stated that the minimum delay

between a ventricular pace event and the next ventricular pace shall be 500 ms. After converting it to the modelled time unit i.e. 0.1 ms, this is expressed as the following conjecture C1:

```
separate(#fin(Lead'dischargePulse(-,<VENTRICLE>), true,
         #fin(Lead'dischargePulse(-,<VENTRICLE>), 5000, false)
```

A requirement that, after an atrial event there must be a ventricular pace after 150 ms ($\pm$ 4 ms), leads to the following conjecture which includes a requirement that the second event occurs, C2:

```
separatereq(#fin(Lead'dischargePulse(-,<ATRIUM>), true,
            #fin(Lead'dischargePulse(-,<VENTRICLE>), 1460, true)
```

A requirement on the maximum delay between pulses being, say, 100 $\pm$8 ms would be expressed as a deadline conjecture as follows, C3:

```
deadline(#fin(Lead'dischargePulse(-,<ATRIUM>), true,
         #fin(Lead'dischargePulse(-,<ATRIUM>), 1080, false)
```

The three validation conjectures above have been applied to test runs of the validation scenarios. In several cases this identified violations in the model and in this way the model could be improved as such bottlenecks were discovered [20,21].

6 Related Work

Efforts are being made to support the incremental development of formal models, but has not so far been extended to model-oriented specifications of real-time systems with explicit deployment. Work in SCTL/MUST [22] addresses the iterative production of early-stage models of real-time systems. As in our approach, validation by testing is supported and the model production process feeds back into requirements scenarios. The Credo project [23] focuses on modelling and analysis of evolutionary structures for distributed services and also includes formal models similar to those described here but without so far considering deployment issues. Our incremental approach also has similarities with refinement-oriented approaches, such as those in event-based B work [24] but here the focus is more on the formal aspects of the refinement, not explicitly addressing time or deployment.

Related work by Suhaib et al. [25] proposes a methodology derived from that of eXtreme Programming, in which "user stories" are expressed as LTL formulae representing properties which are model-checked. On each iteration, new user stories are addressed. The ordering of properties is significant for the practical tractability of the analysis on each iteration. In the context of research on real-time UML [26], a combination of UML and SDL [27] with a rigorous semantic foundation. However, in this work the ability to carry out the validation is more limited when deployment is considered. Burmester et al. [28] describe support for an iterative development process for real-time system models in extended UML by means of compositional model checking, and Uchitel et al. [29] address the incremental development of message sequence charts, again model-checking the models developed in each iteration.

Regarding our particular pacemaker example, we believe that the McMaster pilot problem example has not previously been attacked. It is always a challenge to be first with a new case study and the work presented here should only be seen as the first step

in attacking the pacemaker challenge. From the cardiac pacing domain, the work that comes closest to ours is that of Liu and others addressing safety analysis of a pacemaker product line using state-based modelling [30]. They have a very similar split between the environment and system; the main difference is that they use Rhapsody's executable state models for simulations in contrast to our model-oriented specification supporting deployment. The focus there is on safety analysis while ours is on the incremental development of models.

7 Concluding Remarks and Further Work

Our objective in the work reported here was to assess the feasibility of using an incremental approach in the production of a useful model of an industrially relevant real-time distributed system. The pacemaker case study suggests that such an approach can yield a viable model that can be subjected to validation against system-level properties at an early stage in the development process. The study encourages us to apply the approach to a wider range of examples. The study revealed that the regression test suite built from the validation activities on the intermediate models was valuable in validating the later, more complex models.

It is important to stress that we have not attempted a comprehensive attack on the full Pacemaker specification [8]. Our validation activity in particular, has been preliminary. We would like as a next stage to seek domain expert involvement in the definition of validation conjectures.

The scheduling models in the current VICE formalism limit the range of exploration supported. An important future task is extending this range and increasing the configurability of the model set. In the pacemaker example, we would like to explore a wider range of scheduling assumptions.

Our approach has been pragmatic, driven by the aim of providing a fully formal modelling approach with a low barrier to industrial adoption. As a consequence we have emphasised validation by animation rather than verification by proof. The state of the art in VDM tool support reflects this. Facilities such as the application programmer interfaces and dynamic link libraries in VDMTools allow for co-simulation [31,32] in which models of the environment (e.g. a Matlab model of electrical activity in the heart) can be linked to VDM++ models of discrete event controllers. We plan to do this for the Pacemaker application in order to explore the fault tolerance characteristics of alternative candidate architectures.

Looking forward to proof-based verification, modern implementations of proof support for VDM [33] currently handle a subset of the modelling language but require further work to adapt them to the needs of distributed real-time models in VDM++. Once the tool support is enhanced, enabling analysis of proof obligations for the Pacemaker model, this will be carried out.

We have not yet dealt with the relationship between the incremental addition of detail and formal refinement. In particular, we would like to be able to drive useful proof obligations out of the "refinement" steps. An examination of this issue must address the treatment of atomicity in the abstract and sequential models (for example in handling the maintenance of invariants). To encourage adoption, we feel it is essential that we automate a larger part of the validation process.

Acknowledgments We are grateful to Brian Larson of Boston Scientific and to anonymous FM'08 referees for comments and suggestions on this paper. José Nuno Oliveira, Shin Sahara, Marcel Verhoef, Sander Vermolen and Zoe Andrews contributed to the development of the formalism, tool and method extensions. Fitzgerald's work is supported by the EU Framework 7 Deploy project and the UK EPSRC platform project on Trustworthy Ambient Systems.

References

1. Jones, C.B.: Systematic Software Development Using VDM. Second edn. Prentice-Hall International, Englewood Cliffs, New Jersey (1990) ISBN 0-13-880733-7.
2. Fitzgerald, J., Larsen, P.G.: Modelling Systems – Practical Tools and Techniques in Software Development. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK (1998) ISBN 0-521-62348-0.
3. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object–oriented Systems. Springer, New York (2005)
4. Fitzgerald, J.S., Larsen, P.G.: Triumphs and Challenges for the Industrial Application of Model-Oriented Formal Methods. In Margaria, T., Philippou, A., Steffen, B., eds.: Proc. 2nd Intl. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2007). (2007) Also Technical Report CS-TR-999, School of Computing Science, Newcastle University.
5. Verhoef, M., Larsen, P.G., Hooman, J.: Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In Misra, J., Nipkow, T., Sekerinski, E., eds.: FM 2006: Formal Methods, Lecture Notes in Computer Science 4085 (2006) 147–162
6. Fitzgerald, J.S., Larsen, P.G., Tjell, S., Verhoef, M.: Validation Support for Real-Time Embedded Systems in VDM++. In Cukic, B., Dong, J., eds.: Proc. HASE 2007: 10th IEEE High Assurance Systems Engineering Symposium, IEEE (November 2007) 331–340
7. CSK: Development Guidelines for Real Time Systems using VDMTools. Technical report, CSK Systems (2008)
8. Boston Scientific: Pacemaker system specification. Technical report, Boston Scientific (January 2007) http://www.cas.mcmaster.ca/sqrl/_SQRLDocuments/PACEMAKER.pdf.
9. Woodcock, J.: First Steps in the Verified Software Grand Challenge. Computer **39**(10) (2006) 57–64
10. Ellenbogen, K.A., Wood, M.A.: Cardiac Pacing and ICDs. 4th edn. Blackwell (2005)
11. Mukherjee, P., Bousquet, F., Delabre, J., Paynter, S., Larsen, P.G.: Exploring Timing Properties Using VDM++ on an Industrial Application. In Bicarregui, J., Fitzgerald, J., eds.: Proceedings of the Second VDM Workshop. (September 2000) Available at www.vdmportal.org.
12. Larsen, P.G., Hansen, B.S., et al.: Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language (December 1996)
13. Overture Group: The VDM Portal. *http://www.vdmportal.org* (2007)
14. Lano, K.: Logic specification of reactive and real-time systems. Journal of Logic and Computation **8**(5) (1998) 679–711
15. Verhoef, M., Larsen, P.G.: Interpreting Distributed System Architectures Using VDM++ – A Case Study. In Sauser, B., Muller, G., eds.: 5th Annual Conference on Systems Engineering Research. (March 2007) Available at http://www.stevens.edu/engineering/cser/.
16. CSK: VDMTools homepage. *http://www.vdmtools.jp/en/* (2007)

17. Kurita, T., Oota, T., Nakatsugawa, Y.: Formal specification of an embedded IC for cellular phones. In: Proceedings of Software Symposium 2005, Software Engineers Associates of Japan (June 2005) 73–80 (in Japanese).
18. Macedo, H.: Validating and Understanding Boston Scientific Pacemaker Requirements. Master's thesis, Minho University, Portugal (October 2007)
19. Macedo, H.: VDM models of the Pacemaker Challenge (2007) http://www.vdmportal.org/twiki/bin/view/Main/PacemakerCaseStudy.
20. Sørensen, R.A., Nygaard, J.M.: Evaluating Distributed Architectures using VDM++ Real-Time Modelling with a Proof of Concept Implementation. Master's thesis, Enginering College of Aarhus (December 2007)
21. Verhoef, M.: Modeling and Validation Distributed Embedded Real-Time Systems. PhD thesis, Radboud University Nijmegen (2008)
22. Vilas, A.F., Arias, J.J.P., Redondo, R.P.D., Martinez, A.B.B.: Formalizing Incremental Design in Real-time Area: SCTL/MUS-T. In: Proceedings of the 26 th Annual International Computer Software and Applications Conference (COMPSAC02), IEEE (2002)
23. de Boer, F.: CREDO: Modeling and analysis of evolutionary structures for distributed services (2007) http://www.cwi.nl/projects/credo/.
24. Lecomte, T.: Event B Reference Manual. Technical report, MATISSE/ClearSy (2001)
25. Suhaib, S.M., Mathaikutty, D.A., Shukla, S.K., Berner, D.: XFM: An Incremental Methodology for Developing Formal Models. ACM Transactions on Design Automation of Electronic Systems **10**(4) (October 2005) 589–609
26. Douglas, B.P.: Real Time UML – Advances in the UML for real-time systems. Third edn. Addison-Wesley (2004)
27. de Jong, G.: A UML-Based Design Methodology for Real-Time and Embedded Systems. In: Proceedings of the 2002 Design, Automation and Test in Europe Conference and Exhibition (DATE.02), IEEE (2002)
28. Burmester, S., Giese, H., Hirsch, M., Schilling, D.: Incremental Design and Formal Verification with UML/RT in the FUJABA Real-Time Tool Suite. In: Proceedings of the International Workshop on Specification and vaildation of UML models for Real Time and embedded Systems, SVERTS2004, UML2004 (2004)
29. Uchitel, S., Kramer, J., Magee, J.: Incremental Elaboration of Scenario-Based Specifications and Behavior Models Using Implied Scenarios. ACM Transactions on Software Engineering and Methodology **13**(1) (January 2004) 37–85
30. Jing Liu, J.D., Lutz, R.: Safety analysis of software product lines using state-based modeling. Journal of Systems and Software **80**(11) (November 2007) 1879–1892
31. Verhoef, M., Visser, P., Hooman, J., Broenink, J.: Co-simulation of Real-time Embedded Control Systems. In Davies, J., Gibbons, J., eds.: Integrated Formal Methods: Proc. 6th. Intl. Conference. Lecture Notes in Computer Science 4591, Springer-Verlag (July 2007) 639–658
32. Andrews, Z.H., Fitzgerald, J.S., Verhoef, M.: Resilience Modelling through Discrete Event and Continuous Time Co-Simulation. In: Proc. 37th Annual IFIP/IEEE Intl. Conf. on Dependable Systems and Networks (Supp. Volume), IEEE Computer Society (June 2007) 350–351
33. Vermolen, S.: Automatically Discharging VDM Proof Obligations using HOL. Master's thesis, Radboud University Nijmegen, Computer Science Department (August 2007)