Overture Technical Report Series
No. TR-006

September 2013

Development Process of Distributed Embedded Systems using VDM

by

Peter Gorm Larsen and Sune Wolff
Department of Engineering, Aarhus University
Finlandsgade 22, DK-8200 Århus N, Denmark

Nick Battle
Fujitsu Services
Lovelace Road, Bracknell,
Berkshire. RG12 8SN, UK

John Fitzgerald and Kenneth Pierce
School of Computing Science
Newcastle University
UK

Document history

Month	Year	Version	Version of Overture.exe
April	2010		0.2
May	2010	1	0.2
February	2011	2	1.0.0
September	2013	3	2.0.0

Contents

Preface

This document is intended to provide readers who already have experience with general VDM concepts from language manuals [LangManPP] or books [Fitzgerald&05] and/or courses and it is also assumed that the reader has general knowledge about concepts using for concurrent systems [Ben-Ari82, Hoare85, Chandy&88, Milner89, Lea99]. From a tool perspective it is also assumed that the readers are already familiar with the basic functionality from Overture (`http://www.overturetool.org`) and a UML tool such as Modelio (`www.modelio.org`).

VDM [Jones90, Dawes91, Fitzgerald&98a, Fitzgerald&08] is a formal method [Craigen&93, Hinchey&95, Woodcock&09] and these are characterized by being able to express things in an abstract fashion and having a precise semantics for the models produced in these languages. Three different dialects exists for VDM; VDM-SL, VDM++ and VDM-RT [Larsen&10]. Some of these methods include methodological steps of getting from a very abstract model to a more concrete model. This process is typically referred to as refinement [Jones90, Morgan90, Woodcock&96, Back&98] when formal relationships are included between the different models. In this document no claims are made between the different models so no formal refinement will be included.

This document is structured such that all readers with advantage can read the introduction in Chapter 1 first. In case the reader have limited experience and knowledge about synchronization of concurrent systems it may be an advantage then to just to Chapter 4 to get more knowledge about that before proceeding with the main process guidelines for the development of real-time systems in Chapter 2. The process is followed by a major example that is developed according to the guidelines presented in Chapter 3. Understanding that requires knowledge about both VDM and general concurrency principles. All the different models for the example used in this document are also available on-line from `www.vdmbook.com` so it is possible for the reader to get hands-on experience with them.

In Chapter 5 periodic threads and statically schedulable systems are treated. Chapter 6 provides insight into the scheduling principles that can be used for the execution of concurrent and for distributed VDM-RT models in Overture. In Chapter 7 it is presented how post-execution analysis can be made of timed logfiles produced by Overture during execution of a scenario (primarily using the RealTime Log Viewer). Finally, Chapter 8 rounds off the document with a postscript indicating what the reader should have obtained at this point.

Chapter 1

Introduction

This document describes the envisaged process by which reactive distributed embedded real-time systems could be developed using Overture. In particular, the document describes how key features of real-time systems can be modelled and analyzed using tool support. In this document the main focus is on the different activities performed during the analysis and design phases of the development process. Thus, the implementation phase of the traditional waterfall life cycle [Royce70] is not dealt with in detail in this document. However, the use of Overture does not require the use of the waterfall life cycle and thus parts of the guidelines can be selected on a needs basis.

1.1 Characteristics of Reactive Real-Time Systems

Reactive real-time systems possess a number of unique characteristics which means that conventional (formal and informal) approaches to modelling and analysis are inadequate. That is, in addition to conventional functional correctness (correctness of computed values), other factors can influence whether the software is defined in such a way that the overall system performs correctly, and are therefore challenges to the system developer. These challenges are a result of the reactive, concurrent and real-time nature of such systems, so the challenges intrinsic to each of these kinds of system are described. Often it is simply not possible to check whether a system design will be adequate before it is fully implemented and operating in its real environment. For a number of critical systems this is unacceptably late and this document describes how one can get a higher level of confidence in the correctness of a design using Overture.

1.1.1 Challenges of Reactive Systems

Reactive systems are often *closed loop* systems. That is, they typically repeatedly take data from sensors and compute commands for actuators based on this data. The behaviour of the actuators then influences the values read by the sensors. In order to appropriately model this feedback loop one needs to have an accurate model of the environment in order to be able to validate that the system will work correctly in its expected environment. Moreover reactive systems are typically

non-terminating, so traditional formal methods approaches dealing with total correctness (formal proof of an algorithm always satisfying certain formally defined conditions) are not appropriate.

Modelling the environment behaviour may not be convenient in a discrete event formalism such as VDM. Often a detailed description of physical world entities are more conveniently created using different bases in mathematics such as differential equations. Thus, in case the fidelity of the requirements are sufficiently high it will be optimal to have the environment (the physical system to be controlled) and the system to be developed (the controller of the physical system) be described in different formalisms which can be combined for example using co-simulation [Broenink&10, Fitzgerald&13b]. However in this document the environment will be modelled using VDM for simplicity reasons. We will also discuss under which circumstances a multi-disciplinary models will be advantagous. The DESTECS (Design Support and Tooling for Embedded Control Software) project (`http://www.destecs.org`) was targeted explicitly at this combination and thus complementary methodology documentation came out of that project (and its tool support is now called **Crescendo**.

Reactive systems need to be able to cope with *random events* that might occur in the environment. This means that the modelling of the environment need to incorporate sporadic events. The challenge here is also to be able to handle all possible ways in which the environment can behave, and appropriately document the assumptions made about this behaviour. For critical applications the handling of erroneous situations typically take up the majority of the effort in the development of an appropriate controller.

1.1.2 Challenges of Concurrency

The correctness of a concurrent system can often be dependent on the kind of *scheduling* algorithm used, and the manner in which the scheduling algorithm is used. In order to be able to predict whether a particular design will work correctly for different scenarios one needs to take scheduling into account. In addition to scheduling the mastering of synchronisation between different threads adds to the complexity of concurrency.

1.1.3 Challenges of Real-Time

Real-time systems sometimes need to meet *hard deadlines*. That is, failure to meet a deadline could lead to the system's mission being compromised. Any kind of analysis which can indicate scenarios where such deadlines cannot be met is extremely valuable. Typically, the performance of a small portion of the system can critically affect the ability of the system to meet its deadlines. Such portions are referred to as *bottlenecks*. The earlier potential bottlenecks can be pinpointed the better.

Real-time systems often need to meet *soft deadlines*. That is, deadlines which, if met late, will not cause system failure, but persistent lateness could cause degraded performance. Detection of this is similar to the hard deadlines above; the value of this is necessarily smaller but still important.

Real-Time systems need to be able to cope with periodic events which do not occur perfectly periodically. This is called jitter and the acceptable level for this is dependent upon the different

design decisions made.

1.1.4 Challenges of Distribution

From a system perspective it is often advantageous to structure an embedded system on a number of processors. A part of the distribution of functionality to be performed by such a system is more or less determined by the physical structure of the system and its interaction with the environment. However, for a significant portion of the desired functionality the system architect have a challenging task determining how to allocate different parts of the functionality to different computation units. It requires a lot of skill to become convinced about the chosen architecture living up to different kinds of timing requirements with a given collection of processors with physical characteristics for example in the form of speed and memory.

In order to find an optimal system architecture it is an advantage to experiment with commonly used scenarios with different combinations of different processors and different allocations of functionality to these processors. The development guidelines presented in this document aims to provide an ability for carrying out such experiments early in the life-cycle without having to invest in the real hardware with different capabilities for processor speeds. Naturally it is still just a model but it is taking the speed of the processors and the bus capacities into account so it should at least be able to provide pretty good guidance about an optimal system architecture. The things that are left out of the modelling here are memory and optimisation techniques such as caching and pipelining.

1.2 Overview of VDM and its tool support

In this section we give a brief description of those features of VDM dialects specifically suited to modelling real-time systems and deployment to multiple processors. A more substantial overview of the VDM languages can be found in [LangManPP].

VDM++ is a model-based object-oriented specification language. It permits description of concurrent models by using threads. Real-time behaviour of models can be analyzed dynamically. A VDM++ model is organized as a collection of classes. A class may contain values, types, instance variables, functions and operations. Note that functions are not allowed to read or write instance variables, whereas operations are.

Overture is a tool suite which support amongst other things, static analysis of models (syntax and type checking), and execution of models with dynamic type analysis (invariant checking, pre- and post-condition checking). In addition Overture and VDMTools have features enabling users to move back and forth between a graphical UML view and a textual detailed VDM++ view. As described in [VDM++MethodGuide], this allows the complementary benefits of UML and VDM to be exploited.

A variety of scheduling algorithms are supported by Overture, and during model execution, information on the real-time behaviour of the model is accumulated for post-execution analysis. More details of how Overture facilitates timing analysis is given in Section 1.3. For post-execution

analysis there is also a feature in Overture called RealTime Log Viewer that may be used. More details will be provided about that in Chapter 7.

We now describe four key features of VDM-RT used when modelling real-time systems:

1. threads,
2. duration and cycles statements,
3. system and environment and
4. deployments.

1.2.1 Threads

A thread is an entity within a class. It is used to model independent behaviour. That is, a thread represents activity within a system, whereas a class without a thread is more akin to a server, passively responding to requests. Thus when a class containing a thread definition is instantiated, a thread is created. However this thread may only be scheduled when it is explicitly started using a VDM++ **start** statement. Consider the following example where a class `A` has a thread that is started inside an operation called `op` in a class called `B`:

```
class A

instance variables
  i : nat := 0

thread
  while i < 10 do
    i := i + 1

end A
```

```
class B

operations

op : () ==> ()
op() ==
( dcl a : A := new A();
  start(a)
)
end B
```

In this example, the thread for object `a` will not be available for scheduling until the statement **start**`(a)` has been executed.

Threads may be either procedural or periodic. A procedural thread is simply a statement (as the while loop in the above example), which is executed to completion subject to scheduling and descheduling. A periodic thread has the form

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

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

the periodic occurrence should not start right when the periodic thread is started it is possible to use the `offset` parameter to describe that. Examples of all of this will follow.

Being a bit more precise we can consider Figure 1.1 and state that:

- **period** is a non-negative, non-zero value that describes the length of the time interval between two adjacent events in a strictly periodic event stream (where $jitter = 0$)
- **jitter** is a non-negative value that describes the amount of time variance that is allowed around a single event. We assume that the interval is balanced $[-j, j]$. Note that jitter is allowed to be bigger than the period to characterize so-called event bursts.
- **delay** is a non-negative value smaller than the period which is used to denote the minimum inter arrival distance between two adjacent events.
- **offset** is a non-negative value which is used to denote the absolute time value at which the first period of the event stream starts. Note that the first event occurs in the interval [`offset, offset + jitter`].

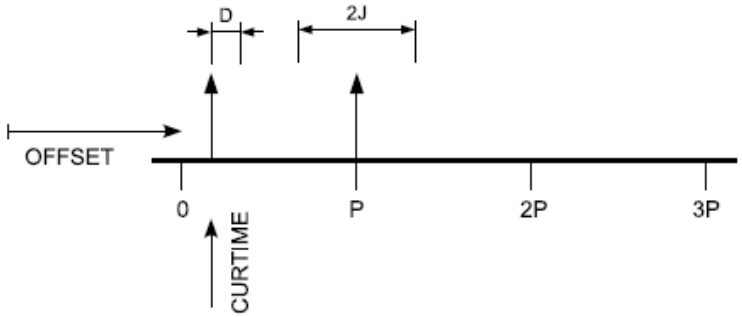

Figure 1.1: Example of a periodic event stream with a (p, j, d, o)-tuple

Communication between threads is based on shared objects. Thus a synchronization mechanism is necessary to ensure integrity of shared objects. This mechanism is described in Chapter 4.

1.2.2 Duration and Cycles Statements

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

```
duration (t) statement
```

This means that *statement* is estimated to take *t* nanoseconds to execute. This information is used for accumulation of real-time behaviour; details of this accumulation, and the manner in which time is to be interpreted are described in Section 1.3 below. Otherwise the duration statement has the exact functional effect of its body *statement*.

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

```
cycles (instruction cycles) statement
```

This means that *statement* is estimated to take *instruction cycles* to execute on any platform. Thus, if a processor with double processor speed is chosen it will take half the time. This information is used for accumulation of real-time behaviour; details of this accumulation, and the manner in which time is to be interpreted are described in Section 1.3 below. Otherwise the cycles statement has the exact functional effect of its body *statement*.

1.2.3 The System and the Environment

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

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

The instances of CPU and BUS must be made as instance variables and the definition must use constructors. The constructor for the CPU class takes two parameters: the first one indicate the primary scheduling policy used for the CPU whereas the second parameter provides the capacity of the CPU (indicated as number of instructions Per Second or Hz - NB. the step size of time is 1 nanosecond). The constructor for the BUS class takes three parameters. The first one indicates the kind of bus, the second one the capacity of the bus (its band width in bytes per second) and finally the third parameter gives a set of CPU instances connected together by the given BUS instance.

The currently supported primary scheduling policies for the CPU are:

<FP>: Fixed Priority. If this scheduling policy is used the cpu scheduler will examine the threads with the highest priority first to see if they are ready to be executed or they are blocked for some reason (a synchronisation constraint or by an operation call on a different cpu).

<FCFS>: First Come First Served. If this scheduling policy is used the cpu scheduler will always swap between threads in a round-robin fashion starting with the first one created. Each thread will be allowed to continue its execution for a time limit (decided by the user) before the next thread is examined. In this way fairness will be ensured at normal cpus but for threads running at the virtual cpu (where time is not incremeneted based on instruction execution) no fairness is ensured since ian infinite loop here may take over the entire execution.

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

<FCFS>: First Come First Served

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

1.2.4 Deployments

The CPU class has member operations called deploy and setPriority. The deploy operation takes one parameter which must be an object that is declared as a static instance variable inside the system. The semantics of the deploy operation is that execution of all functionality inside this object will take place on the CPU that it has been deployed to. The setPriority operation takes two parameters where the first must be the name of a public operation that has been deployed to the CPU and the second parameter is a natural number. The semantics of the setPriority operation is that the given operation is assigned the given priority (the second parameter). This will be used when fixed priority scheduling is used on the given CPU. For operations that are used on a CPU using fixed priority scheduling that has **not** been assigned a value using the setPriority operation a default priority of 1 is assigned to it. The same holds for normal theads in active classes.

1.3 Timing Analysis

The objective when performing timing analysis using Overture is to identify potential performance bottlenecks. That is, to identify those portions of the model that could cause scheduling problems and/or failure to meet deadlines. This allows the feasibility of a particular *dynamic architecture* to be examined. A dynamic architecture is a design which has been decomposed into a number of processes or threads onto a number of processors.

Note that checking timing assertions (desirable timing properties expressed formally) during run-time is neither an objective of the intended timing analysis, nor feasible within the framework described. Moreover it is not an objective to *formally verify* correct behaviour with respect to timing requirements.

The above objective is dependent upon the target execution architecture, and also the real-time kernel which will be used (as this dictates the scheduling policy to be used). To support the above objective Overture aims to simulate the target execution environment. The target environment is the system itself allocated to different processors and connected with BUS'es as well as the environment that itself may be considered as a system. That is, it approximates the timing properties of the target processors, the BUS'es connecting them and emulates the scheduling policy to be used by the real-time kernel on each processor.

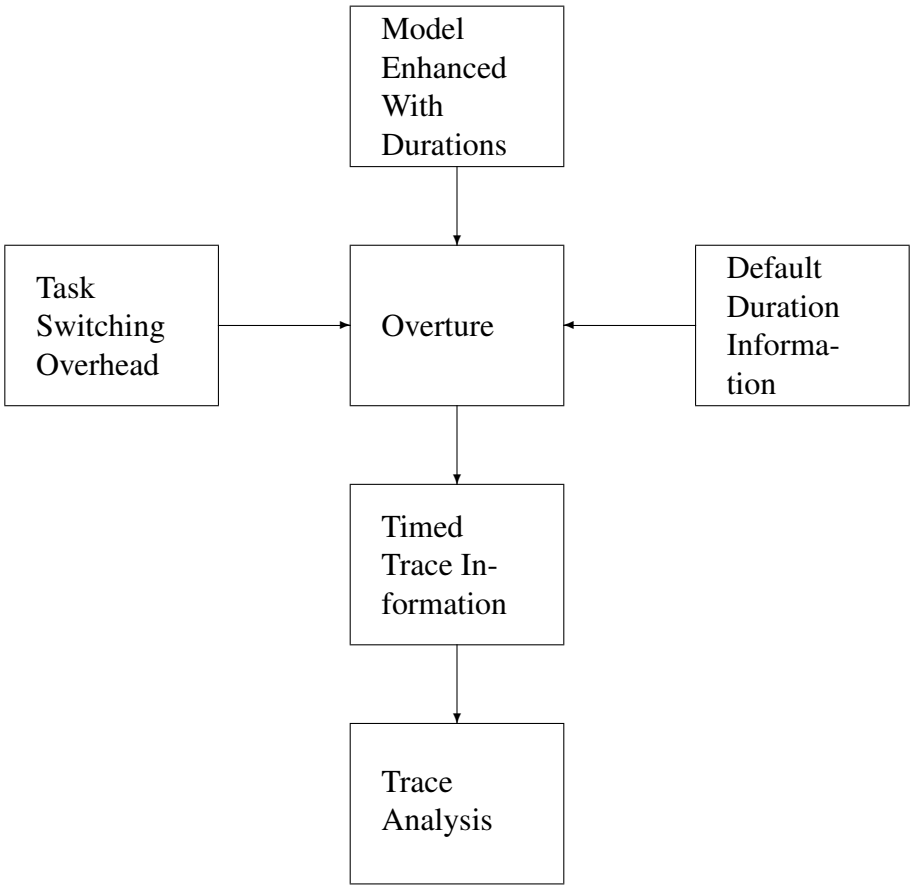

Figure 1.2: Approach to Timing Analysis

With respect to timing behaviour, Overture *simulates* the time of the target processors. That is, during execution the interpreter maintains an internal variable which corresponds to the clock of the target processors, i.e. the clock of the target processors will be simulated. The interpreter will adopt the same scheduling algorithm as that intended for the different distributed processors in the final system. During execution of the model a number of events will occur in parallel for the different processors:

- Swapping in and out of threads;
- Operation requests, activations and completions;
- Messages being communicated between processes on busses.

We call such events, *trace events*. During execution all trace events are recorded, time stamped with the simulated time at which they occurred.

Four sources of timing information are used during execution:

Model enhanced with cycles and durations: When executing a portion of the VDM model which falls under the scope of a duration or a cycles statement, the internal clock is incremented by the given duration/number of clock cycles at the completion of the statement. Note that the durations are absolute in time whereas the cycles are relative to the speed of the processor on which it is executed.

Task switching overhead: A task switching overhead can be defined, to correspond to the task switching overhead for the intended real-time kernel.

Default cycles information: For those portions of the model not in the scope of an explicit duration or cycles statement, conventional worst-case analysis is used. However it is parametrized in terms of the time taken to perform elementary assembly instructions without taking possibly optimisation techniques such as cache and pipelining into account. The user can then define the time for these assembly instructions, for the intended target processor. We call this mapping from assembly instructions to execution time on the target processor, the *default cycles information*.

Capacity of `CPU`'s and `BUS`'es: If a distributed system is modelled the capacity of the `CPU`'s and the `BUS`'es used in the interpretation of scenarios of the system

An overview of the approach is shown in Figure 1.2. Overture takes the three sources of timing information listed above, and uses this information when executing a model. During execution a timed trace file is created, containing time stamped trace events listed in the order of occurrence. Since this is just a plain text file, it may be analyzed separately.

1.4 Structure of document

After this short introduction to reactive systems, VDM and timing analysis this document proceeds in Chapter 2 with a relatively short description of the envisaged development process for real-time systems using Overture. As mentioned above, this document focuses on the different activities in the analysis and design phases in order to be able to get feedback on the timing properties of suggested designs before they are carried into the final implementation. After this general presentation of the design process a substantial example development following this process is presented in Chapter 3. Chapter 2 makes forward references to the appropriate parts in Chapter 3. The example used for illustrating the development process in Chapter 3 is a missile counter measures system.

Then a series of chapters follows with more reference material about how different aspects of reactive systems can be modelled using the Overture technology for real-time systems. Chapter 4 explains how synchronization is ensured in VDM++. In Chapter 5 modelling of periodic systems and events is explained. In Chapter 6 the scheduling policy supported by the Overture technology is presented. Finally in Chapter 7 it is illustrated what kind of post-execution trace analysis can be performed using Overture including calibration of the timing results. In Chapter 3 there are a number of forward references to some of the reference chapters. Readers who are unfamiliar with

the kind of material may therefore benefit from jumping to the reference explanations when such references are made.

The Appendices include a glossary (Appendix A), a few design patterns useful for real-time systems (Appendix B) and finally a full listing of all the examples (Appendix C).

Chapter 2

Development Process For Real-Time Systems

In this chapter we give an overview of the proposed development process for real-time systems. As mentioned in Chapter 1 the focus of the development process presented here is on the activities in the system analysis and design phases.

In Figure 2.1 an overview of the different phases we touch upon in this chapter is presented. This is the traditional V-life cycle used by many industrial organizations as an approximation of what goes on in reality (with iterations and feedback between phases). This structuring in phases may be used either with a traditional waterfall process model or for each iteration using Boehm's spiral model [Sommerville82, Boehm88]. The difference is mainly that in the spiral process model the artifacts with the highest risk will be analyzed first, and this would have the consequence that in the process we describe below one would abstract away from parts which does not have any impact on the analysis needed to mitigate the highest risks. The green arrows in the figure indicate the primary flow of information (and in this case VDM models) whereas the purple (vertical) lines indicate that the tests conducted at the development phases (using models) may be reused at the corresponding acceptance test level.

The development process will normally begin with analyzing the informal requirements and capturing these to form a design-independent specification of the system to be developed. Based on this description one needs to structure the system into a static architecture and create a sequential VDM++ design model of the system. 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. At this stage it may prove necessary to revisit the concurrent design model, as it may be that design decisions made at that stage 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). From the concurrent and distributed real-time VDM-RT design model an implementation may be developed. Testing of the final implementation (and the different design-oriented models) may be able to use the most abstract model as a test oracle. We return to the different test phases in Section 2.6.

When developing a model of a real-time system, it is typically difficult to separate the system from the environment in which it is executing, and with which it interacts. Therefore it is often the

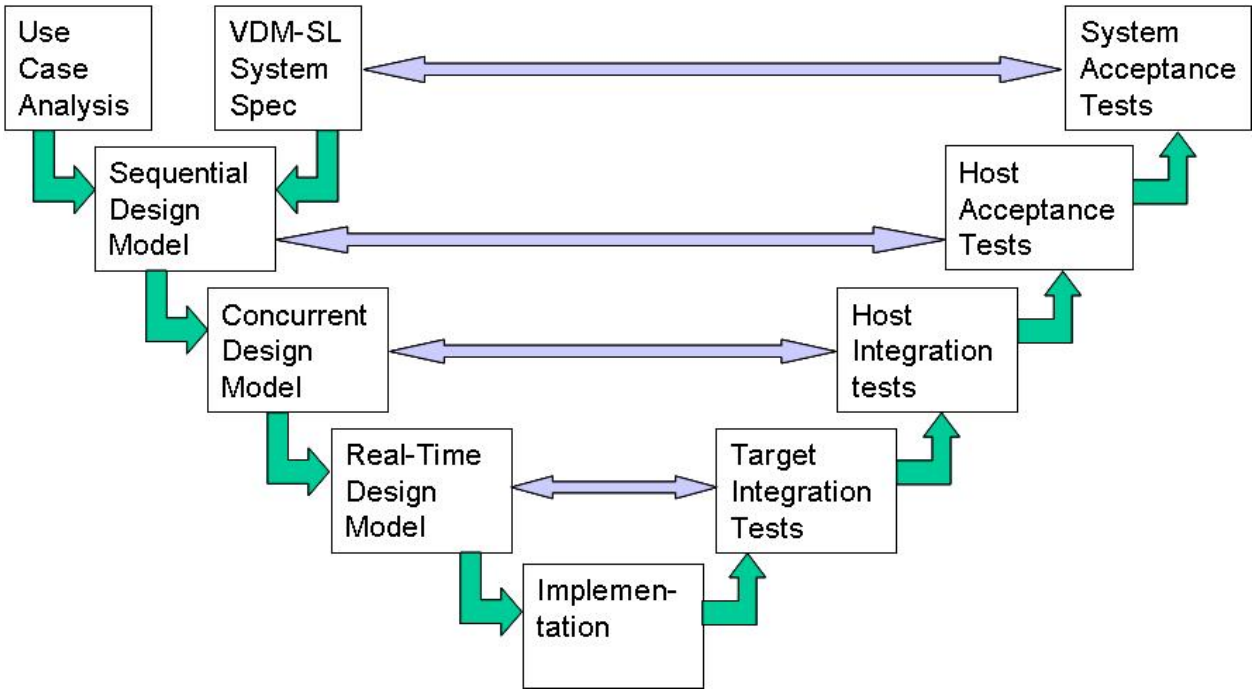

Figure 2.1: Overview of Development Process

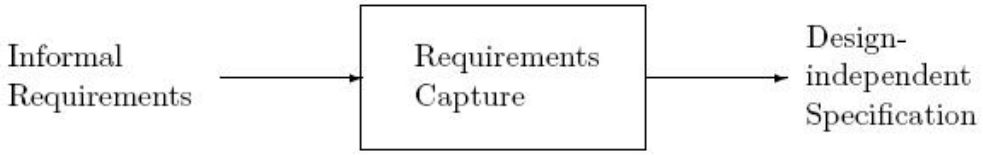

Figure 2.2: Overview of the input/output relationship for requirements capture

case that all the relevant parts of the environment are also modelled.

2.1 Requirements Capture

The first phase of a system development process is to capture the requirements of the new system. This phase is also called the system analysis phase or the specification phase depending upon the different company standards. We recommend that this stage be performed in either UML or by using VDM-SL. For both approaches, the starting point is the informal requirements, and the end point is a specification of the requirements to the system, which is independent of any design concerns, as shown in Figure 2.2. In the Model Driven Architecture (MDA) [MDA] terminology this is called a Platform Independent Model (PIM).

Both approaches are described in the sequel. In Section 2.1.1 a conventional UML [UML20]

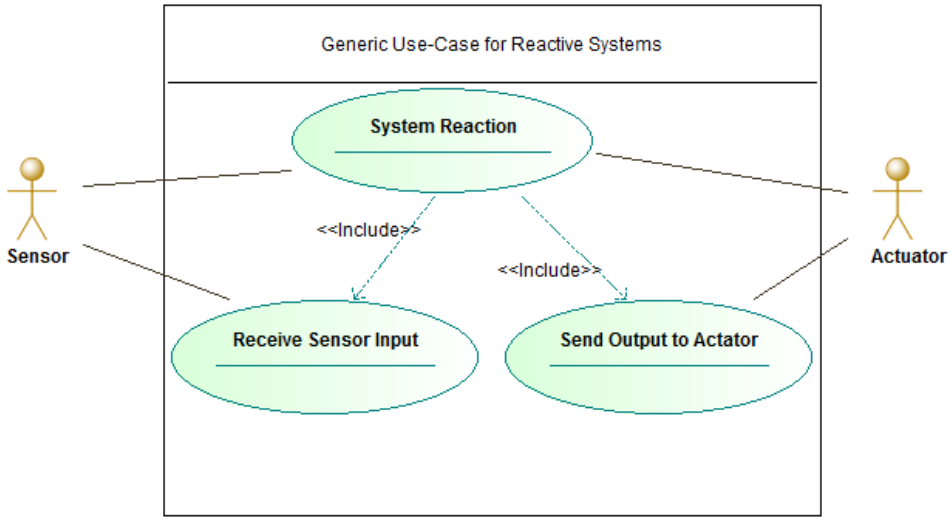

Figure 2.3: General use case for embedded systems

approach is described, in which use-cases are created and discussed with clients and users. This approach is illustrated with the counter measures example in Section 3.2.

Alternatively a flat VDM-SL model, and a Graphical User Interface (GUI) connected to this model to allow users and clients to interact with the animated model could be created (in a similar manner to that described in [CashPoint]). This approach is described in Section 2.1.2 and illustrated with the counter measures example in Section 3.3.

2.1.1 Capturing Requirements with UML Use Cases

In this Section it is described how requirements may be captured using UML use case diagrams [UML20]. The presentation assumes that the tool Rational Rose [Rose&00] or the tool Enterprice Architect is used, but other UML tools have similar functionality. An example of this analysis is given in Section 3.3

Find the actors and use cases

- Identify the frontiers of the system and their principal functionality.
- Identify the actors in the system. The actors may be equipment, users or the system environment.
- Identify use cases.
- Summarize the role of each actor and the goal of each use case.
- Arrange the actors and use cases into related packages.

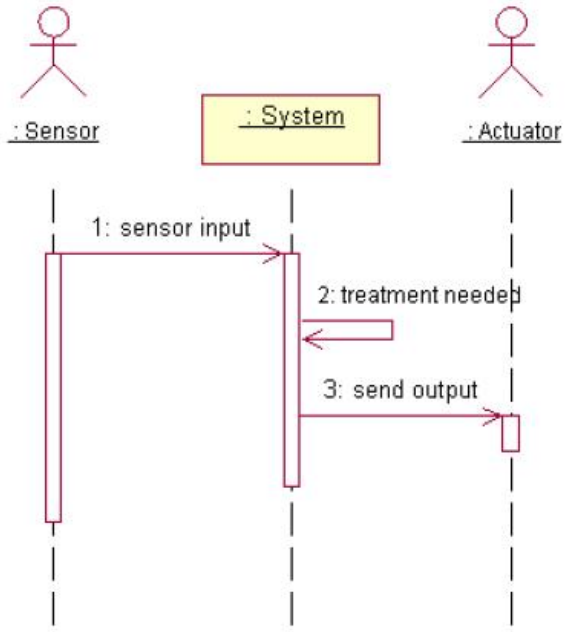

Figure 2.4: General Sequence Diagram for an Embedded System

- Describe the use cases using use case diagrams (in Figure 2.3 such an example is shown).
- Describe the states of the system using a state transition diagram.

Structure the use case model

- Define the relation `<<includes>>` between use cases (e.g. in Figure 2.3).
- Define the relation `<<extend>>` between use cases.
- Define the generalization relation between use cases.
- Define the generalization relation between actors.

These stereotypes are defined in the UML 2.0 standard [UML20].

Specify the use cases in detail

For each use case

- Detail the interaction between the actors and the system.

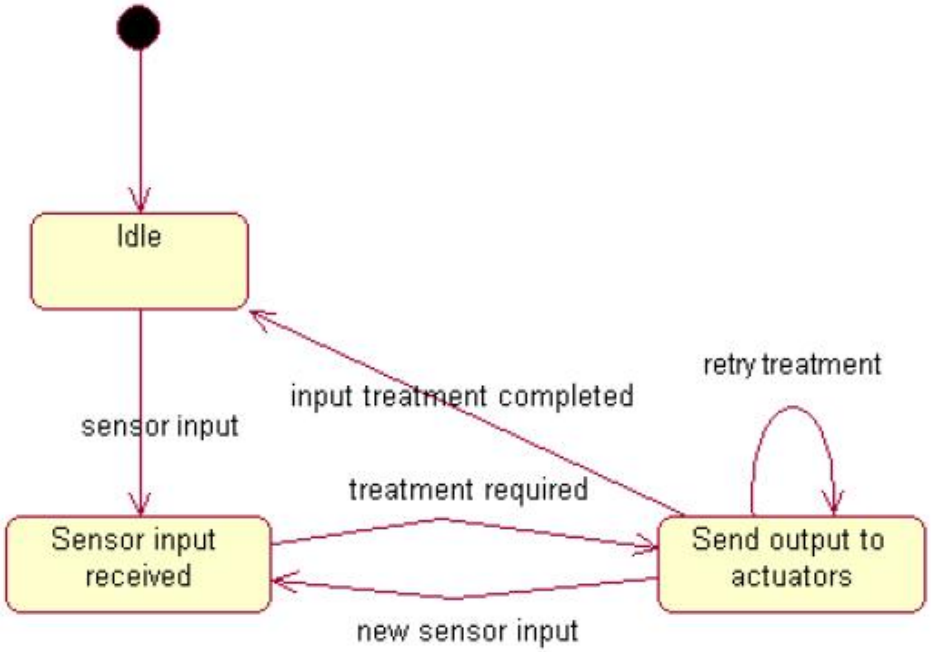

Figure 2.5: General Activity Diagram for an Embedded System

- Structure the interactions between the actors and the system. Note that normal sequences of actions should be distinguished from abnormal sequences of actions (e.g. exceptional cases, degraded cases, failure cases etc).

- A use case could perhaps be associated with a user interface that is already defined. This aspect should not be developed with particular concern to the human-computer interface, but it is useful to represent this interface with the description facilities of the use case.

- If useful, all the exit scenarios of a use case can be specified. The difference between a use case and an associated scenario is that a scenario is an instance of a use case.

- Illustrate each use case or scenario:

  Using a sequence diagram: At this stage the system in question is not yet divided into classes so such sequence diagrams will only show the flow of control between the actors and the system. In version 2.0 of UML there are also possibilities to include information about alternative flow of control primitives [UML20]. Typically a diagram should fit on one sheet of A4/letter paper. An example of this is shown in Figure 2.4.

  Using a collaboration diagram as an alternative view of the sequence diagram. This allows representation of a simple scenario.

Using an activity diagram if necessary (or multiple activity diagrams). An example of this is shown in Figure 2.5.

Using a state diagram if the complexity of the use case justifies it.

2.1.2 Capturing Requirements Using VDM-SL

As an alternative to the use case approach described above in Section 2.1.1, the informal requirements can be captured in a design-independent way using VDM-SL. In the first instance this follows Chapter 2 of [Fitzgerald&98b, Fitzgerald&05, Fitzgerald&09]. At this stage, time can be included as a state variable and can be driven as part of the model. In this way the functional, timing, and time-dependent functional requirements can be captured within one model.

The general strategy used to model a real-time system is to model the system as one top-level operation where the input is considered as a sequence of events which comes into the system to be modelled. The output from the top-level operation will then be those events which are sent out from the system. If time is essential, all input and output events must be tagged with the time at which the event appeared (i.e. an extra field in the event record values). In this way time is modelled explicitly in such a VDM-SL model.

The general structure of the top-level function in such a VDM-SL model would look like (this is without taking the closed loop complications into account at the top-level):

```
operations

  PerformSystemReaction: seq of SensorInput ==>
                         seq of ActuatorCommand
  PerformSystemReaction(inputseq) ==
   if inputseq = []
   then []
   else SensorTreatment(hd inputseq) ^
        PerformSystemReaction(tl inputseq)
```

The types `SensorInput` and `ActuatorCommand` can then either include the time explicitly as an attribute or a fixed stepping length between the inputs can be modelled such that the time is represented implicitly. When the treatment of a sensor input may take longer than the next sensor input arrives, it is likely that this will cause an interruption of the existing treatment. In that case the functional description using the functions `SensorTreatment` and `PerformSystemReaction` used above are more complicated. A full example of such a VDM-SL model is given in Section 3.3. Naturally the feedback loop where the way the environment behaves depending upon the system reaction cannot be taken appropriately into account with this approach. Thus, if a sequence of sensor input does not fit with the order of the system output sequence. It simply means that the given input sequence does not match reality. Despite this such a high level executable model may be valuable to act as an oracle against real final implementation scenarios.

In order to accomodate for a proper feedback loop one simply needs to introduce an accumulating parameter with the actuator commands to be issued unless new sensor inputs are detected. That accumulating parameter can then be manipulated accordingly when new sensor input arrives.

This approach is actually followed in Section 3.3.

Note that it is possible to construct a GUI to animate the VDM-SL model to allow users, domain experts and clients to see how the requirements have been captured. This animation technique allows visualization of a number of scenarios where the input/output relations can be inspected.

2.1.3 Validating Requirements Capturing

Traditionally in the life-cycle only test planning would take place in the early phases of a systems development [Sommerville82]. One of the advantages of the development process described here is, that it makes it possible to carry out systematic testing, and thus get feedback on such test plans, much earlier in the life cycle. If the flat VDM-SL approach has been used for capturing the requirements, then the GUI used for interaction with the client and users should be reused, and the model should be animated to the satisfaction of the client and users. At the final acceptance test of the completed system the regression test environment should be reused as much as possible by changing the script from execution of the abstract VDM-SL model to execution of the final implemented system as a kind of proof of concept prototype.

2.1.4 Criteria for Completion

This stage is complete when:

1. The architecturally significant use cases have been completed and they have been manually validation for example using inspection or

2. The abstract VDM-SL model has been completed and validated.

2.2 Sequential Design Model

A sequential design model must describe both the data that is to be computed, and how it is to be structured into static classes, without making any commitment to a specific dynamic architecture.

The first stage in creating a sequential model is to decide on a static architecture. A static architecture is an arrangement of system behaviour into classes/objects. There already exist a number of classical books about deriving a class structure [Rumbaugh&91, Meyer88, Booch&97, Douglass99] so we will not include that discussion in this document. This document will only provide a few guidelines about how the classes can be identified and discuss to what extent a VDM-SL model can be reused when one produces VDM++ skeletons for each of the identified classes in a system.

The approach to developing a static architecture depends on whether the UML approach or the VDM-SL approach was used to capture requirements. These are therefore treated separately below. Note that whichever requirements capture approach was used, the result of the current phase will always be VDM++ class skeletons.

	Actor	Entity Class	Boundary Class	Control Class
Actor	N/A	No	Yes	No
Entity Class	No	No[1]	No	Yes
Boundary Class	Yes	No	No	Yes
Control Class	No	Yes	Yes	Yes

Figure 2.6: Recommendations for Associations Between Classes

2.2.1 If UML Was Used For Requirements Capture

If UML was used for requirements capture, the uses cases are analyzed to identify a number of classes, and from these classes VDM++ class skeletons are produced.

Identify the classes

This activity consists of identifying concepts, and more generally, abstractions from the specification resulting from the requirements capturing process. Also, relationships between these classes should be defined. Entire books are written about this subject so here only a short introduction is provided using terminology from the literature [Kruchten00]. Note that at this stage it is more important to identify the classes, than to add detail to particular classes. There should be classes both for the system itself and also for the artifacts from the environment in order to be able to capture the feedback from the environment. However it may be possible to define some attributes and relations on the basis of dependencies identified by use cases. From these classes, a first class diagram may be created.

The following stereotypes should be used for each class:

Entity Class Indicates that the class contains persistent data.

Control Class Indicates that the class controls, sequences and coordinates activity in the system.

Boundary Class Indicates that the class is an interface to an actor (a part of the environment to the system).

A number of recommendations apply to the use of associations between classes. These can be found in Table 2.6 where "No" indicates that they should not be combined and "Yes" indicates that they can be combined.

Producing VDM++ Class skeletons

When the definition of classes is finished, the next stage is to generate skeleton VDM++ classes for the model. Class skeletons can automatically be generated using UML class diagrams and the UML-VDM++ link feature from VDMTools or the Overture linkage to Modelio. Each such file contains the VDM++ skeleton of the class.

[1] Unless there exists an aggregation or composition between the two classes.

2.2.2 When VDM-SL was used for Requirements Capturing

When the informal requirements are captured in a precise way using VDM-SL it is often possible to conceptually reuse most of the VDM-SL operations inside the VDM++ classes where structuring and design decisions are taken into account. However, for real-time systems the nature of the VDM-SL model will traditionally be very different from the design which is needed in the system architecture so less reuse should be expected for this kind of systems. Still, the use of VDM-SL for requirements capturing may still be valuable for such systems.

Having defined the formal functional requirements in the VDM-SL model, these requirements should be mapped to a static architecture. That is, skeleton VDM++ classes should by synthesized from the VDM-SL model. This is not an automatic process and we do not claim to be original or better than anybody else with guidelines about how one most conveniently divides the system into components or classes. The following guidelines should be used during this synthesis:

- Record types in the VDM-SL model typically become classes in the static architecture.
- Activities which are functionally independent will typically be encapsulated in separate classes in the static architecture.

A guideline which is always good to follow to divide into classes and relationships between classes, is that one should model both the system in question and its environment. Typically each sensor and actuator will get their own classes. From a deployment perspective instances of such classes will also typically be allocated to their own `CPU`'s. These would be the actors from the use case diagram. Traditionally it is often a good idea to have a `World` class which controls the overall interaction between the environment and the classes representing the system in question. This class can then be used for setting up the appropriate connections between the different objects and testing the interaction between them.

An example of synthesizing such class skeletons is given in Section 3.4.

2.2.3 Class Descriptions in VDM++

Class skeletons should evolve into complete specifications. This involves completing operation bodies for those operations already referred to in the previous stages, and adding any auxiliary functions and operations.

Where possible invariants on types and instance variables should be identified and specified. Pre-conditions should be specified for all operations and functions that are non-total, and post-conditions should be specified wherever it is meaningful to do so (i.e. where it does not lead to restatement of the function or operation body, if present).

An example of completed class descriptions in VDM++ is given in Section 3.5.

2.2.4 Modelling of the Environment

It is often difficult to model the behaviour of a reactive real-time system without reference to the environment in which it is executed. Thus it is often convenient and useful to create classes (and/or

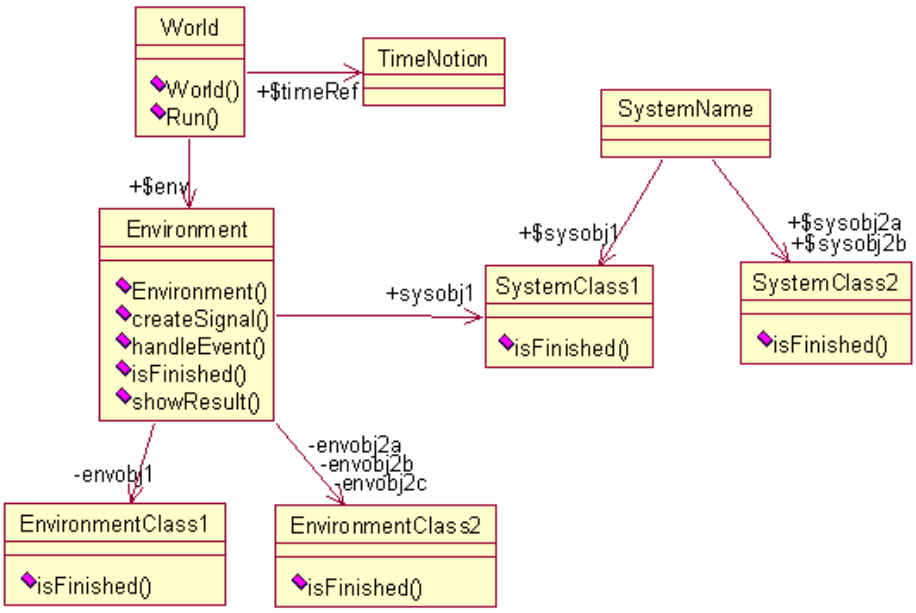

Figure 2.7: General Class Diagram for an Embedded System

threads in a concurrent model) which represent this environment. These classes may then be used to inhabit the model with test data.

2.2.5 Typical Design Structure

Figure 2.7 illustrates the general static class structure recommended for reactive embedded systems in the guidelines in this document. It is recommended to always have a class responsible for the `Environment` and a class responsible for the composition of the components using inside the system called `SystemName`. It is also recommended always to have a class called `World` that contain a constructor responsible for setting up both the environment and the system components enabling them to carry out a scenario.

The scenario is then usually invoked by an operation called `Run`. Thus, typically one can made a call like `new World().Run()` where the details of the scenario are either given as a parameter or stored in a file that will be read by the standard `IO` class. Since reactive systems will react to stimuli from the environment `IO` is typically used inside the `Environment` class or one of the other classes representing parts of the environment (`EnvironmentClass1` and `EnvironmentClass2` in Figure 2.7).

In addition to a constructor the `Environment` is recommended to contain operations to provide input for the system (`createSignal`) and operations to receive feedback from the system (`handleEvent`). It is also recommended to provide an operation called `isFinished` for all classes that play an active role in a scenario (this operation shall indicate when the processing is

finished locally). This holds for both environment and system classes. As will be clear after the example is presented in Chapter 3 the signature for `isFinished` will in the sequential model typically yield a Boolean result, but that will change in the concurrent and distributed models. For the `Environment` class it is also necessary to provide some kind of operation that can show the result of running a given scenario (`showResult`).

It is recommended to use static public instances from the `World` and the overall system class `SystemName` in order for them to be accessible throughout the model without having to pass the object references around. This also includes some kind of notion of time if time is of importance for the reactive system (which it usually is). In the sequential model it is recommend to use a basic `Timer` class here, but in the concurrent model a stronger notion of time and synchronisation as will be seen in Section 2.3.5.

Finally it is worth mentioning that it is recommended to let the flow of control be steered from the `Environment`. At the sequential level this is typically done using a loop until both the environment and the system are finished (using different `isFinished` operations. Inside the loop `Step` operations are used for stepping over time and passing the control around to the different components of the system.

2.2.6 Validation of the Model

Whenever possible the model should be executed to allow validation. If the use cases approach has been used in the requirement capturing process, then execution should ensure that model execution satisfies all of the use cases identified.

Note that even if the whole model is not executable, portions of it may be, and therefore these portions should be validated in the manner described above.

In addition to the animation technique for validation of the model described above a more systematic traditional testing approach should also be used to validate the model. Thus, test cases should be defined and an automatic test script should be made such that the test cases can be used in a regression fashion during this phase and reused in subsequent phases. The VDMUnit test framework from [Fitzgerald&05] (chapter 9) may be used with advantage here (include the `VDMUnit` standard library).

2.2.7 Criteria for Completion

This stage is complete when:

1. The VDM++ model is syntax and type correct;
2. The UML class diagram and VDM++ model are synchronized;
3. The model has been validated;
4. XX% test coverage for executable portions of the model (uncovered parts should be justified).

Here "XX" is a figure that would be determined by the standards used by each individual company or organization.

2.3 Concurrent VDM++ Design Model

The objective in developing a concurrent VDM++ design model is to take the first steps towards a particular dynamic architecture, without worrying in the first instance about real-time behaviour. An example of such a model is given in Section 3.6.

2.3.1 Identification of Threads

The first step in developing a concurrent model is to identify which computations can be performed independently of each other. These computations may then be separated into independent threads. Often, this separation will be forced by hardware constraints and/or pre-defined architectural requirements. While it is worthwhile identifying as many independent threads as possible, in general the number of threads in the system should be minimized. This is because (again in general) threads increase complexity of the model and make model validation more difficult.

2.3.2 Communication

After identification of threads, it must be decided which threads communicate with each other, and what values are passed. Accordingly object sharing between threads should be specified. For classes representing shared objects, appropriate synchronization should be specified.

2.3.3 Synchronization Points

In addition to synchronized object sharing it may be necessary to introduce explicit synchronization points. That is, ensure that a particular thread does not proceed beyond a specified point, until another thread has reached an appropriate state. This can be important to ensure correct sequencing amongst the different threads, or else to ensure freshness of data. In general if an operation both reads from and writes to instance variables it will be advantagous to make it mutexed with itself.

2.3.4 Validation of the Model

The model should be executed using the same scheduling policy as that used by the target real-time kernels used for the processors in the target environment. Upon execution the model should be free of deadlocks (note that there is not yet a formal analysis conducted in Overture for this, so it is limited to the scenarios used as test cases). Moreover the model should functionally yield the same results as the sequential model, perhaps modulo some adjustments with the formulation of the values and the time at which they appear.

2.3.5 Typical Design Structure

The general structure from Figure 2.7 is also recommended for concurrent models. The main changes from the sequential model are:

- Flow of control is changed such that instead of it residing with the `Environment` it will be distributed to all the active parties and thus the body of the `Step` operations are typically turned into threads.
- Synchronization between the different threads are specified using permission predicates and mutex constraints.
- The signature for the `isFinished` operations is changed such that no value is returned. Instead the Boolean expression is typically used as permission predicates. This is a way to block the threads requesting this operation until the corresponding instance indeed is finished with its business.
- It is recommended to replace the simple `Timer` class with a `BaseThread` and a `TimeStamp` class from Appendix B.2 in order to easily synchronize the steps taken now when the flow of control is distributed to multiple threads.

2.3.6 Criteria for Completion

1. The VDM++ model is syntax and type correct.
2. The UML class diagram and VDM++ model are synchronized.
3. The model has been validated.
4. The model displays no deadlocks;
5. XX% test coverage for executable portions of the model (uncovered parts should be justified).

Here "XX" is a figure that would be determined by the standards used by each individual company or organization.

2.4 Concurrent Real-Time and Distributed VDM-RT Design Model

At this stage real-time information is added to the model. In addition if the system in question is to be distributed over multiple processors allocation of functionality to such processors are conducted. The timing analysis described in Section 1.3 is then performed. Following such analysis, it may be concluded that the proposed dynamic architecture is in fact infeasible. Therefore it might be

necessary to revisit Section 2.3 and revise the dynamic architecture or revise the deployment of the functionality to CPU's or their capabilities. An example of a concurrent real-time model is given in Section 3.7.

2.4.1 Duration and Cycle Statements

For those parts of the model where knowledge of real-time behaviour exists (e.g. components which are being reused) duration statements should be used to give precise estimates of fixed execution times. The cycle statements should be used to give precise estimates of the execution time relative to a processor (in the form of the number of expected clock cycles).

For those parts of the model which are effectively modelling the environment. Depending upon the accuracy needed for the VDM-RT model of the system under design the model of the environment should be more elaborate. Here is is possible to have different instances in the environment on each their own virtual processor or even be modelled using its own formalism and validated using a co-simulation interface. However, it is also possible for example to model the environment to constrain the environment instances to be deployed on the same processor.

For modelling closed loop systems, a duration statement should be used to force a delay in the time between sending a command to an actuator, and seeing its effect at a sensor.

2.4.2 Task Switching Overhead

The task switching overhead for the target real-time kernel should be ascertained. This value should be used as the task switching overhead in Overture and VDMTools during execution of the model.

2.4.3 Typical Design Structure

The general structure from Figure 2.7 is still recommended for the real-time distributed models. The main changes from the concurrent model are:

- The `SystemName` class is now changed to become a **`system`** where additional instance variables are introduced for all the `CPU`'s and `BUS`'es that one would like to distribute the functionality to. In addition a constructor is introduced here where the actual deployment of the static instance variables to `CPU`'s take place. In addition it is possible to define the priority of different operations in this constructor in case priority-based scheduling is used on some of the `CPU`'s.

- Optionally a system can be created of the `Environment` class as well. In case this is not done VDMTools will simply execute all functionality on a virtual `CPU` for each instance created in the `World` class. Thus, it may be valuable making a system for the `Environment` in case that it is essential that the environment functionality have dependencies upon each other and thus when one instance is executing another instance cannot also be executing in a true parallel fashion.

- A number of the operations that simply need to start execution on a different thread are made asynchronous using the **`async`** keyword. Note that in essence if such two instances are deployed at different `CPU`'s behind the back of the user they will communicate over a `BUS` and here the notion of synchronism is essential.
- Some of the threads (typically those that previously had `Step` functionality) are turned into periodic threads.
- The explicit notion of time (at the concurrent level using the `BaseThread` and `TimeStamp` classes) is removed. Now time is implicit and it is possible to make use of the keyword **`time`** to refer to the current time on a given `CPU`.

2.4.4 Validation of the Model

The model should be executed using as many different scenarios as are necessary to satisfy the following two criteria:

1. To achieve the required test coverage as dictated by the completion criteria for this stage;
2. To cover all use cases identified during requirements capture (if the UML was used to capture the requirements).

This execution can be used to check correctness of computed values in these scenarios, and the absence of deadlocks. Note that introduction of real-time information could in itself introduce deadlocks to the model, as it could cause the scheduler to make different decisions to those made during execution of the untimed model.

2.4.5 Timing Analysis

Execution of the model will create a time trace file which may be analyzed subsequently (examples of this are shown in Chapter 7). Analysis should establish the following:

- No periodic threads miss their deadlines.
- All real-time response requirements are satisfied (all hard real-time deadlines are met with the used scenarios).

Here the RealTime Log Viewer feature from Overture is particular valuable in the automatic analysis of these timed traces. It is able to automatically to provide:

- A graphical overview of the physical architecture with the `CPU`'s and the `BUS`'es.
- Show an overview of the overall execution and communication between the `CPU`'s.
- Show a detailed overview of the instances and the execution and communication between these at a single `CPU` in a detailed fashion.

2.4.6 Criteria for Completion

1. The model is syntax and type correct.
2. The UML class diagram and VDM-RT model are synchronized.
3. The model is schedulable.
4. The model displays no deadlocks.
5. The model is functionally correct for all executed scenarios.
6. The model includes a deployment of functionality to a physical architecture.
7. All periodic threads make their deadlines with the chosen physical architecture for the tests conducted.
8. Any real-time response requirements are satisfied for the tests conducted.
9. XX% test coverage for executable portions of the model (uncovered parts should be justified).

Here "XX" is a figure that would be determined by the standards used by each individual company or organization.

2.5 Implementation

The approach to implementation depends upon the target implementation language and constraints on program structure and performance. If C++ is the implementation language, and dynamic memory allocation is permitted, then use of the VDMTools C++ Code Generator is possible. This implies a massive reduction in effort, since the implementation is obtained by one mouse click. Similarly if Java is the implementation language, the VDMTools Java Code Generator can be used. However, such automation should not be relied upon for hard-real-time systems and the distribution of the model os **not** taken into account in this code generation.

In other circumstances the implementation must be written by hand. However this tends to be quite straightforward due to the quantity and depth of information acquired during the modelling stages. Typically a number of rules can be applied reasonably mechanically, to translate the VDM-RT model into code.

2.6 The Different Test Phases

As identified in Figure 2.1 there are a number of different phases with different levels of testing conducted on the system being developed. With conventional development technology it is normally not possible to validate whether a system under development will be able to meet its

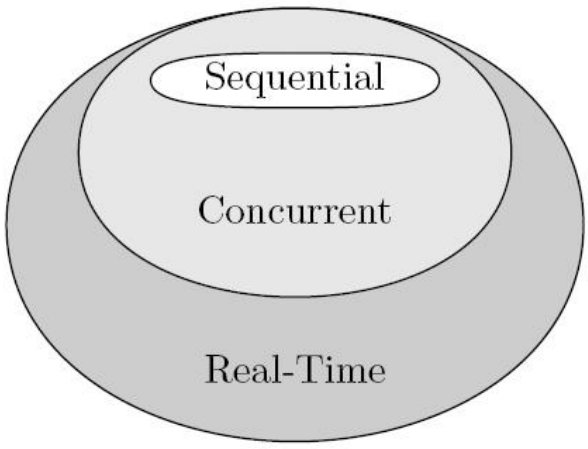

Figure 2.8: Relationship Between VDM Models

deadlines before the target integration test phase. If these tests show that it is not possible to meet the deadlines, there is therefore a significant cost involved in redesigning the system and its corresponding implementation to improve this situation. The approach described in this document aims to find potential bottlenecks in the system even before the final implementation is made. The aim is to be able to sit on the host computer (where the development takes place) and simulate the timing behaviour of the concurrent real-time VDM-RT design model with information about the timing behaviour of the intended possibly distributed hardware platform.

With the approach described in this document the test cases used in the different test phases are already executed when the design is made and then reused with the final implementation subsequently. This is a major change compared to the conventional way of development where feedback about the timing properties of a particular design only comes very late in the development process. Note however that each of the different models gets closer and closer to reality, where the most abstract models have a somewhat idealised view of the system and its environment, the later models takes more and more details into account.

2.7 Discussion

In this chapter we have described the evolution of one VDM-SL model and three object oriented VDM models: a sequential model, a concurrent model and a concurrent distributed real-time model. There is no formal relationship between them in the sense that no formal relationship has been established between any of the models. This is consistent with the pragmatic software engineering approach advocated in this process, in contrast to a pure formal development. Nonetheless, since the process involves adding detail to each model, to obtain the next model, in some sense we can consider each model a sub-model of the next, as shown in Figure 2.8. That is to say, the

concurrent model is an extension of the sequential one, and the real-time and distributed model is an extension of the concurrent one. Each of the models get more and more concrete and complex, but gradually closer and closer to reality.

Chapter 3

Example Development

In this chapter an example development process is presented. This development goes through each of the steps from Chapter 2. The system developed is a simplified version of an actual real-time system. The chapter begins by presenting the informal requirements for the system and then the different models developed are presented.

3.1 The Counter Measures System

The application to be modelled in VDM is the controller for a missile counter measures system. This takes information from sensors concerning threats and sends commands to hardware which releases flares intended to distract the threat sensed. The overall high-level architecture is shown in Figure 3.1.

Flares are released in a timed sequence, the number of flares released and the delay between releases depending on the threat and its angle of incidence with the missile. Different places around an aircraft different flare dispensers or magazines are located dealing with threats arriving from different angles. The threat sensors relay the ID of the threat to the controller. For each different kind of ID and the angle of the missile the controller must then derive a plan for how to deal with the given threat by firing a sequence of flares with a given pattern with a given flare dispenser (a magazine) dealing with the given angle. Such a pattern contains the number of flares to be fired and the delay between each firing. The task communicates the stated number of firings to the flare release hardware with the specified delay between each communication. For the purposes of this document it is assumed that there are only two kinds of physical flares.

An example firing sequence is shown in Figure 3.2. Flare release commands are represented by the vertical arrows. Five actions are depicted in this figure. For simplicity we will assume that only three kinds of missiles are known to the system: A, B and C where they have increasing priority in this order. Similarly we simplify the example by supposing that there are only two kinds of physicals flares, together with the special "Do Nothing" flare, which requires that nothing is released for a specified duration. For the purposes of this example we assume the counter measures system for each flare dispenser to respond to the different missile types as shown by the firing sequences in Figure 3.3.

Figure 3.1: Context Diagram for the Counter Measures System

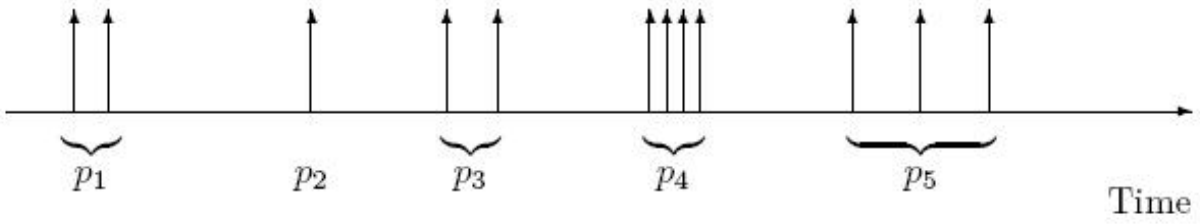

Figure 3.2: Example Firing Sequence

The following requirements apply to this system:

1. If while computing the firing sequence with a flare dispenser for a given threat from an angle treated by the same flare dispenser, another threat is sensed (in the same angle area), the system should check the priority of the more recent threat and, if greater than the previous one, should abort computation of the current firing sequence. Computation of the new firing sequence should then take place.

2. If different threats are sensed with angles that are treated by different flare dispensers the corresponding firing sequences shall be performed in parallel.

3. The controller should be capable of sending the first flare release command within 250 milliseconds of receiving threat information from the sensor.

4. The controller should be able to abort a firing sequence within 130 milliseconds.

The system to be developed can be composed of a number of different hardware components (sensors and flare dispensers) organised in a physical architecture to be determined.

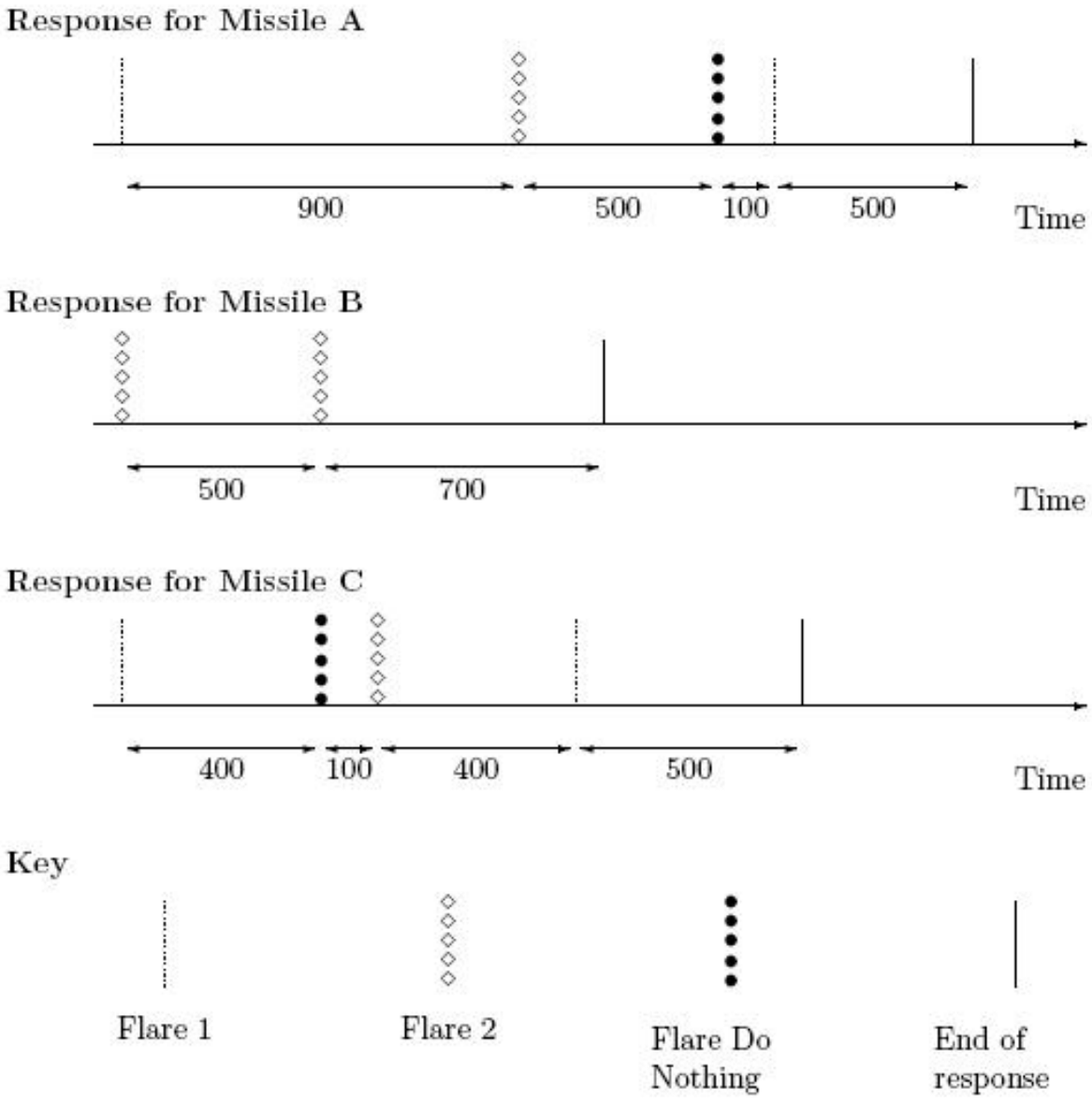

Figure 3.3: Example Missile Responses Used in the Models

3.2 UML Use Cases for Requirements Capture

If use cases are used for capturing the requirements for the counter measures system we produce a use case diagram like the one shown in Figure 3.4. For each of the ovals (the graphical representation of a use case) a textual description of the use case must be made. This is typically done in an itemized fashion. In the three subsections below we show such a description for each of the different use cases.

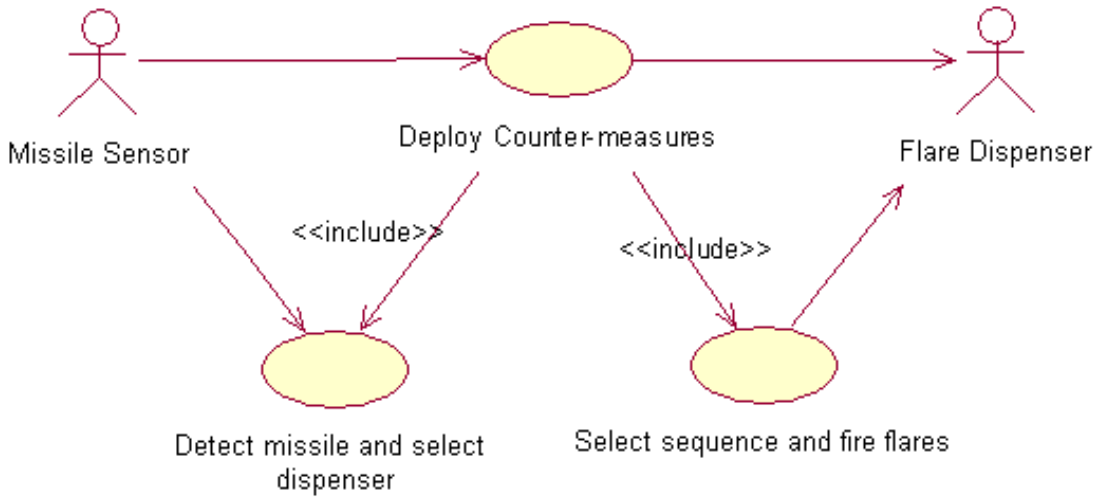

Figure 3.4: Use Case Diagram for the Counter Measures System

3.2.1 Deploy Counter-measures

Primary Actor(s): Missile Sensor and Flare Dispenser

Secondary Actor(s): None

Intent: This use case is responsible for taking the given threat identification and fire flares accordingly.

Assumptions: All potential threats are identified.

Known Limitations: If threats come too close to each other the system may not be able to manage to deal with all of them.

Includes: "Detect Missile and Select Dispenser" and "Select Sequence and Fire Flares"

Pre-conditions: A threat has arrived and has been detected by a missile sensor.

Course of action: When a new threatening missile has been detected by a missile sensor the appropriate flare dispenser should be determined upon the angle of attack. Then, it should be determined if another threat is already being handled by the flare dispenser. In that case the priority of the thread identification must be compared and if the new threat has a higher priority its sequence of flares must be determined and that flare dispenser must interrupt what it is currently doing and start firing this new sequence. If the priority is equal to or lower the threat is simply ignored. If no threats have been detected previously the corresponding sequence of flares must be determined and subsequently fired using that flare dispenser. This is illustrated in the state diagram shown in Figure 3.5.

Post-conditions: When the firing of the flares is complete the threat should be distracted and no longer be targeting the system guarded by the counter measures system.

3.2.2 Detect Missile and Select Dispenser

Primary Actor(s): Missile Sensor

Secondary Actor(s): None

Intent: This use case is responsible for taking the given threat identification and its angle of attack and based on this determine what flare dispenser to select.

Assumptions: All potential threats are identified.

Known Limitations: If threats come too close to each other the system may not be able to manage to deal with all of them.

Includes: None.

Pre-conditions: A threat has arrived and been detected by the missile sensor.

Course of action: Whenever a thread identification and its angle of attack is received the responding flare dispenser must be selected and the sequence of flares fires must follow the identification from Figure 3.3.

Post-conditions: The correct flare dispenser and the flare sequence to be fired has been identified.

3.2.3 Select Sequence and Fire Flares

Primary Actor(s): Flare Dispenser

Secondary Actor(s): None

Intent: This use case is responsible for performing the actual firing of a sequence of flares according to the timing requirements described in Section 3.1.

Assumptions: All potential threats are identified.

Known Limitations: There is probably a physical limit concerning how close the different flares can be fired after each other.

Includes: None

Pre-conditions: A sequence of flares have been identified.

Course of action: An internal timer is started and the different flares are fired by the flare dispenser at the times identified by the firing sequence.

Post-conditions: All necessary flares have been released.

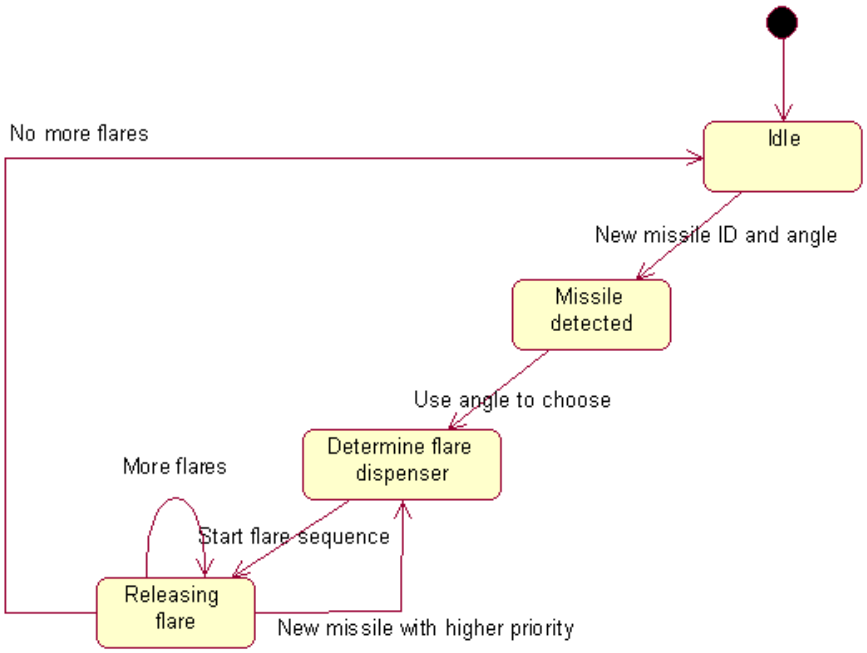

Figure 3.5: State Diagram for Deploying Counter Measures

3.2.4 Summary

As it can be seen from the use case descriptions given above the key functionality of the counter measures system is identified by the use cases. It is an abstract way of looking at the system, independent of the different design issues which must be taken into account later in the development process. The use cases can be a first way to communicate that one has captured the essential usage of the system to other domain experts. The use case diagram can be a nice way to get an overview of how the different use cases hang together, in particular when more complex systems are to be modelled. However, any validation of the use cases can only take place in the form of manual reviews. There is no way in which any kind of automated validation can be performed when we deal with natural language formulations like the ones shown above.

3.3 VDM-SL for Requirements Capture

In this section the requirements to the counter measures system are captured in a precise way using VDM-SL. The intention is to end up with an abstract specification independent of any design issue. The development of the model follows the general principles from Section 2.2.1. The underlying modelling decisions made in this first model is to inspect the first missile and determine what the appropriate response should be if that was the only threat received. In case there are more than

one missile arriving in an area before the processing of the current missile the response is altered in case the priority of the new missile is higher than the current one.

3.3.1 Definition of Types

The counter measures system takes `MissileInput`'s as input. This is a sequence of values of pairs of `MissileType` and `Angle`. `MissileType`'s are constant values representing the different possible missiles that could be detected. In this case we have three different missiles, for testing purposes, together with the special value `<None>` representing the absence of any missile. `Angle` for simplicity is modelled as a number from 0 to 360 degrees (corresponding to one possible model of a two dimensional coordinate system).

```
types

MissileInputs = seq of MissileInput;

MissileInput = MissileType * Angle;

MissileType = <MissileA> | <MissileB> | <MissileC> | <None>;

Angle = nat
inv num == num <= 360;
```

The type `Output` represents a mapping from identifiers for magazines (`MagId`) to a sequence of OutputSteps, where an `OutputStep` is a pair consisting of the type of flare to be released (`FlareType`) and the time at which it was released (`AbsTime` modelled in milliseconds).

```
Output = map MagId to seq of OutputStep;

OutputStep = FlareType * AbsTime;

AbsTime = nat;
```

Note that it is assumed that there only are two kinds of physical flares (`FlareOne` and `FlareTwo`), and also the action of doing nothing can be critical for reacting to a threat, so it is considered as a kind of flare. All of these are additionally tagged according to the type of missile which they are responding to, in order to allow easy validation of generated results.

```
FlareType = <FlareOneA> | <FlareTwoA> | <FlareOneB> |
            <FlareTwoB> | <FlareOneC> | <FlareTwoC> |
            <DoNothingA> | <DoNothingB> | <DoNothingC>;
```

A `Plan` is used during construction of the output. This consists of a `FlareType` to be released, and the amount of time to wait after its release before the next one should be released, `Delay`.

```
Plan = seq of (FlareType * Delay);

Delay = nat;
```

3.3.2 Value Definitions

Information concerning how to handle different missiles is stored in `responseDB` (a mapping from missiles to sequences of responses represented in a `Plan` as identified by Figure 3.3).

```
values

responseDB : map MissileType to Plan =
{<MissileA> |-> [mk_(<FlareOneA>,900), mk_(<FlareTwoA>,500),
                 mk_(<DoNothingA>,100), mk_(<FlareOneA>,500)],
 <MissileB> |-> [mk_(<FlareTwoB>,500), mk_(<FlareTwoB>,700)],
 <MissileC> |-> [mk_(<FlareOneC>,400), mk_(<DoNothingC>,100),
                 mk_(<FlareTwoC>,400), mk_(<FlareOneC>,500)]
};
```

The relative priority between missiles is represented using `missilePriority`, which maps each missile to a numeric value, where greater numeric value indicates higher priority.

```
missilePriority : map MissileType to nat
                = {<None>     |-> 0,
                   <MissileA> |-> 1,
                   <MissileB> |-> 2,
                   <MissileC> |-> 3};
```

Each value in the input is separated by 100 milliseconds. Larger gaps between missile arrivals are indicated by the value `<None>` in the input. We use the symbolic constant `stepLength` to represent this temporal separation.

```
stepLength : Time = 100
```

3.3.3 The Counter Measures Functionality

The top-level function is called `CounterMeasures`. This takes `MissileInputs` as input and returns `Output`. It consists simply of a call to the auxiliary function `CM`.

```
functions

CounterMeasures: MissileInputs -> Output
CounterMeasures(missileInputs) ==
  CM(missileInputs,{|->},{|->},0);
```

The recursive version of the counter measures function, `CM`, takes four parameters. They can be explained as:

`missileInputs`: This parameter contains the missile input which has not yet been considered in the analysis of which flares should be fired. Recursion is done over this parameter such that in each recursive call this sequence will be one smaller.

`outputSoFar`: This parameter contains a mapping from the magazine identifiers to the flare sequence expected to be fired (and their expected firing time) given the missile inputs taken

into account so far. This is the accumulating parameter which at the end will contain the final result.

lastMissile: This parameter contains mapping from the magazine identifier to the last missile which has had effect on the output so far relative to the `MagId`. The priority of this missile is important in relation to the next missile arriving.

curTime: This parameter specifies the time at which this missile has been detected (a multiple of `stepLength`).

If no missiles are left to take into account for counter measures the `outputSoFar` can be used directly. Otherwise the priority of the next arriving missile must be compared to the `lastMissile`. If the priority of the new arriving missile is higher than the last missile the existing plan for output must be interrupted and the response for the new missile must be incorporated instead. If on the other hand the priority is lower, the current missile can be ignored.

```
CM: MissileInputs * Output * map MagId to [MissileType] *
    nat -> Output
CM( missileInputs, outputSoFar, lastMissile, curTime) ==
  if missileInputs = []
  then outputSoFar
  else let mk_(curMis,angle) = hd missileInputs,
           magid = Angle2MagId(angle)
       in
         if magid not in set dom lastMissile or
            (magid in set dom lastMissile and
             missilePriority(curMis) >
             missilePriority(lastMissile(magid)))
         then let newOutput =
                     InterruptPlan(curTime,outputSoFar,
                                   responseDB(curMis),
                                   magid)
              in CM(tl missileInputs, newOutput,
                    lastMissile ++ {magid |-> curMis},
                    curTime + stepLength)
         else CM(tl missileInputs, outputSoFar,
                 lastMissile,curTime + stepLength)
measure CMLen;
```

Note that the `CMLen` function used in the **measure** part here simply is used to indicate how this recursive definition is guranteed to terminate.

```
CMLen: MissileInputs * Output * map MagId to [MissileType] * nat -> nat
CMLen(list,-,-,-) == len list;
```

The function `InterruptPlan` is used to modify the previously expected output so that the response for a higher priority missile can be incorporated into the output. This means that the output before `curTime` is unchanged, whereas the output on or after `curTime` is taken from the

given `Plan`[1]. Thus, conversion needs to take place here; this is performed by `MakeOutputFromPlan`.

```
InterruptPlan: nat * Output * Plan * MagId -> Output
InterruptPlan(curTime,expOutput,plan,magid) ==
  {magid |-> (if magid in set dom expOutput
              then LeavePrefixUnchanged(expOutput(magid),
                                        curTime)
              else []) ^
              MakeOutputFromPlan(curTime, plan)}
  munion
  ({magid} <-: expOutput);
```

`LeavePrefixUnchanged` ensures that the output before the current time is not affected by the latest missile arrival.

```
LeavePrefixUnchanged: seq of OutputStep * nat ->
                      seq of OutputStep
LeavePrefixUnchanged(output_l, curTime) ==
  [output_l(i) | i in set inds output_l
               & let mk_(-,t) = output_l(i) in t <= curTime];
```

`MakeOutputFromPlan` converts a sequence of responses (`response`) which began at time `curTime` and converts it into an `Output` value. It converts the responses into an output in which the first flare was released at time 0. This output is then offset by the current time, to produce the desired output.

```
MakeOutputFromPlan : nat * seq of Response -> seq of OutputStep
MakeOutputFromPlan(curTime, response) ==
  let output = OutputAtTimeZero(response) in
    [let mk_(flare,t) = output(i)
     in
       mk_(flare,t+curTime)
    | i in set inds output];
```

The function `OutputAtTimeZero` takes a response and converts it into an output whose first flare is released at time zero. Thereafter the delay between flare releases corresponds to the delay specified by the response.

```
OutputAtTimeZero : seq of Response -> seq of OutputStep
OutputAtTimeZero(response) ==
  let absTimes = RelativeToAbsoluteTimes(response) in
    let mk_(firstFlare,-) = hd absTimes in
      [mk_(firstFlare,0)] ^
      [ let mk_(-,t) = absTimes(i-1),
            mk_(f,-) = absTimes(i)
        in
          mk_(f,t)
      | i in set {2,...,len absTimes}];
```

[1]Note that `Output` is in terms of absolute time, whereas `Plan` is in terms of relative time.

The function `RelativeToAbsoluteTimes` performs conversion from relative delays into absolute times. This recursively offsets later flare releases in the response by the delay of the first flare release.

```
RelativeToAbsoluteTimes : seq of Response ->
                          seq of (FlareType * nat)
RelativeToAbsoluteTimes(ts) ==
  if ts = []
  then []
  else let mk_(f,t) = hd ts,
           ns = RelativeToAbsoluteTimes(tl ts) in
         [mk_(f,t)] ^ [ let mk_(nf, nt) = ns(i)
                        in mk_(nf, nt + t)
                      | i in set inds ns]
measure RespLen;

RespLen: seq of Response -> nat
RespLen(l) ==
  len l;
```

The `Angle2MagId` function makes a conversion from the input angle of the missile to the magazine to cope with that missile. In this early high-level model this function have simply hard-coded 4 different magazines each coping with 90 degrees for test purposes.

```
Angle2MagId: Angle -> MagId
Angle2MagId(angle) ==
  if angle < 90
  then mk_token("Magazine 1")
  elseif angle < 180
  then mk_token("Magazine 2")
  elseif angle < 270
  then mk_token("Magazine 3")
  else mk_token("Magazine 4");
```

3.3.4 Validation of the Model

In order to gain confidence in the functionality of the model being appropriate it is necessary to validate it somehow. Since this model is made in VDM-SL it is fortunately possible to validate it using the VDM-SL version of Overture. Once the model have been syntax and type checked it is possible to test that the main function called `CounterMeasures` behave in the intended fashion. Here the strategy is as always with testing to start from simple values and gratually produce more and more conplex scenarios. In this case one could for example start with testing the behaviour with each of the missiles arriving as the only one.

In Appendix C.1 three value definitions are presented that has been used to test the intended behaviour with more complex scenarios. These three turned out to be enough to cover all parts of the model which can be shown using the test coverage feature from Overture. This first model in VDM-SL is abstracting away from different timing delays but even though it makes sense to

inspect the specific timing requirements for the system (listed as item 3 and 4 on page 40). Both of these timing requirements are also validated with the given scenarios.

3.3.5 Summary

In Section 3.3 a very abstract model of the counter measures system has been presented. Note how this model follows the general principles for specifying a real-time system using VDM-SL presented in Section 2.1.2. This model is entirely independent of any design issues which need to be taken into account later to break the system down into its static components (classes) and its dynamic architecture (threads). The main virtue of this model is that it precisely characterizes the requirements of the counter measures system without any kind of design details.

This model was tested with a number of test cases. A significant portion of time was invested in ensuring that these test cases covered a wide range of scenarios, and that the model delivered the expected results. However there are two returns on this investment: first, the test cases can be reused during system acceptance testing; second, the VDM-SL model can be used as an oracle during the development of later models, to compare the functional behaviour of these later models.

The lesson which can be learnt from this abstract model is that we now have a common understanding of what the counter measures system is intended to do, which can be used as an oracle when the final implementation is completed. Note however, that it must be kept in mind that this is an idealised version of the counter measures system.

3.4 VDM++ Class Skeletons

In this section it is considered how a first decomposition of the counter measures system into different classes can be achieved, as discussed in Section 2.2.1 and Section 2.2.5. The main guideline we have used to derive the structuring of the system into classes is that it is necessary to include the environment in the modelling. This means that we must have classes which simulate the sensors and actuators of the system.

In Section 3.3 the main activities displayed are detection of missiles and releasing flares. In addition we might expect to have a class which simulates the hardware sensors which alert the system to arriving missiles. Hence we have four candidate classes, which we respectively call `MissileDetector`, `FlareController`, `FlareDispenser` and `Sensor`.

As discussed in Section 2.1.2 neither the use cases presented in Section 3.2 nor the VDM-SL specification presented in Section 3.3 provide much help in this structuring of the system. Neither is it easy to reuse all parts of the VDM-SL model because it is a real-time system.

3.5 Sequential VDM++ Design Model

The sequential model has the following classes:

CM: This is the overall system class (a `SystemName` class) that creates static public instances for all the system components.

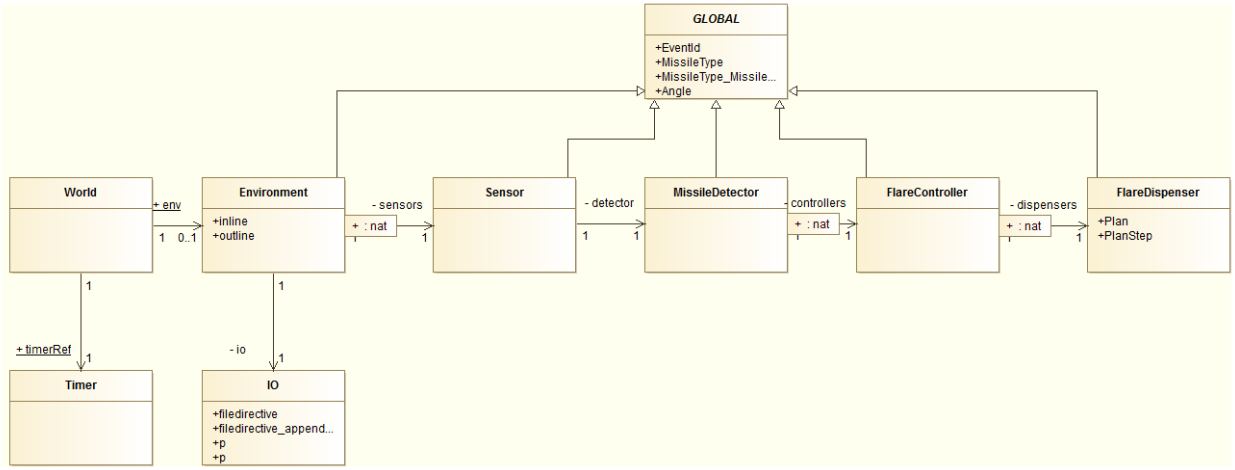

Figure 3.6: Class Diagram for the Sequential Counter Measures Model

`World`: The main class, used to combine the system classes and the environment and allow execution of scenarios.

`Environment`: This is used for modelling the environment (in this case the sensors providing input for the system).

`Sensor`: A class for modelling the hardware used to sense the arrival of missiles with a given angle.

`MissileDetector`: A class which takes information from the `Sensor` and passes it to one of the `FlareController`'s.

`FlareController`: A class which controls outputs of flares for a given detected missile using a number of flare dispensers.

`FlareDispener`: A class which master the actual firing of flares depending upon the type of the missile.

`Timer`: A timer class used to step time throughout the sequential VDM++ model.

`IO`: A VDM++ standard library class.

`GLOBAL`: This is a superclass providing a number of general definitions used by a number of the system and environment classes.

An overview of the relationship between the different classes can be seen in Figure 3.6.

3.5.1 The Counter Measures Class

This is the overall system class called `CM` corresponding to the `SystemName` in Section 2.4.3. In Section 3.3 four different magazines (or flare dispensers) was used covering 90 degrees each. In reality it is possible to have different numbers of sensors and actuators. In principle it is also possible to have overlapping sensors and actuators redundantly in order to increase fault tolerance for the overall system. In order to illustrate how this can be done using the VDM++ framework described in this document a design is presented with

- four sensors covering 90 degrees of angle each;
- one missile detector;
- three flare controllers covering 120 degrees of angle each controlling four flare dispensers;
- twelve flare dispensers coping with 30 degrees of each.

In this way each flare controller will have four flare dispensers to control each. As it will be clear below this is a rather complex system, but it is modelled in a fashion that makes it easy to re-configure it to investigate alternative compositions of sensors, flare controllers and flare dispensers.

In the `CM` class this is documented as:

```
class CM
instance variables

public static
detector : MissileDetector := new MissileDetector();
public static sensor0 : Sensor := new Sensor(detector,0);
public static sensor1 : Sensor := new Sensor(detector,90);
public static sensor2 : Sensor := new Sensor(detector,180);
public static sensor3 : Sensor := new Sensor(detector,270);
public static
controller0 : FlareController := new FlareController(0);
public static
controller1 : FlareController := new FlareController(120);
public static
controller2 : FlareController := new FlareController(240);
public static
dispenser0 : FlareDispenser := new FlareDispenser(0);
public static
dispenser1 : FlareDispenser := new FlareDispenser(30);
public static
dispenser2 : FlareDispenser := new FlareDispenser(60);
public static
dispenser3 : FlareDispenser := new FlareDispenser(90);
public static
dispenser4 : FlareDispenser := new FlareDispenser(0);
public static
dispenser5 : FlareDispenser := new FlareDispenser(30);
public static
dispenser6 : FlareDispenser := new FlareDispenser(60);
public static
dispenser7 : FlareDispenser := new FlareDispenser(90);
public static
dispenser8 : FlareDispenser := new FlareDispenser(0);
public static
dispenser9 : FlareDispenser := new FlareDispenser(30);
public static
dispenser10 : FlareDispenser := new FlareDispenser(60);
public static
dispenser11 : FlareDispenser := new FlareDispenser(90);

end CM
```

3.5.2 The World Class

The `World` class consists of two static public instance variables and two operations. The two instance variables refer to the `environment` and the `timerRef` to be used by both the environment and system classes.

```
class World

instance variables

public static
env : [Environment] := nil;
public static timerRef : Timer := Timer`GetInstance();
```

The constructor `World` is used to set up the object topology (adding sensors, controller and dispensers to the environment, the detector and different controllers respectively). In principle the `new Environment` could be made in the **`instance variables`** section but that could introduce infinite recursion in the initialisation of Overture or VDMTools.

```
operations

public World: () ==> World
World () ==
 (  -- set-up the sensors
    env := new Environment("scenario.txt");
    env.addSensor(CM`sensor0);
    env.addSensor(CM`sensor1);
    env.addSensor(CM`sensor2);
    env.addSensor(CM`sensor3);

    -- add the first controller with four dispensers
    CM`controller0.addDispenser(CM`dispenser0);
    CM`controller0.addDispenser(CM`dispenser1);
    CM`controller0.addDispenser(CM`dispenser2);
    CM`controller0.addDispenser(CM`dispenser3);
    CM`detector.addController(CM`controller0);

    -- add the second controller with four dispensers
    CM`controller1.addDispenser(CM`dispenser4);
    CM`controller1.addDispenser(CM`dispenser5);
    CM`controller1.addDispenser(CM`dispenser6);
    CM`controller1.addDispenser(CM`dispenser7);
    CM`detector.addController(CM`controller1);

    -- add the third controller with four dispensers
    CM`controller2.addDispenser(CM`dispenser8);
    CM`controller2.addDispenser(CM`dispenser9);
    CM`controller2.addDispenser(CM`dispenser10);
    CM`controller2.addDispenser(CM`dispenser11);
    CM`detector.addController(CM`controller2);
   );
```

The `Run` operation is used to execute the model and this is done by handing over control to the environment.

```
public Run: () ==> ()
Run () ==
```

```
  env.Run()

end World
```

3.5.3 The Global Class

The GLOBAL class is responsible for providing a common place to store global definitions of relevance for the rest of the model.

This includes value definitions indicating the number of degrees that can be observe (the viewing angle is termed "aperture"):

```
class GLOBAL

values

public SENSOR_APERTURE = 90;
public FLARE_APERTURE = 120;
public DISPENSER_APERTURE = 30
```

It also includes the MissileType and FlareType that was presented in Section 3.3.1 and in addition a type for identifying the incoming events EventId.

```
types

public
MissileType = <MissileA> | <MissileB> | <MissileC> | <None>;

public FlareType =
    <FlareOneA> | <FlareTwoA> | <DoNothingA> |
    <FlareOneB> | <FlareTwoB> | <DoNothingB> |
    <FlareOneC> | <FlareTwoC> | <DoNothingC>;

public Angle = nat
inv num == num <= 360;

public EventId = nat;

public Time = nat
```

The canObserve operation is used to check whether an aperture (a sensor, a flare controller or a flare dispenser) can observe a missile coming at the pangle angle. Each subclass of GLOBAL must supply an operation called getAperture that will yield a pair of degrees indicating the left hand side that the aperture can observe and the number of degrees it can observe.

```
operations

public canObserve: Angle * Angle * Angle ==> bool
canObserve (pangle, pleft, psize) ==
  def pright = (pleft + psize) mod 360 in
```

```
    if pright < pleft
    -- check between [0,pright> and [pleft,360>
    then return (pangle < pright or pangle >= pleft)
    -- check between [pleft, pright>
    else return (pangle >= pleft and pangle < pright);

public getAperture: () ==> Angle * Angle
getAperture () == is subclass responsibility;

end GLOBAL
```

3.5.4 The Environment Class

The `Environment` class is responsible for the interaction with all the system classes to and from the environment classes. It is defined as a subclass of `GLOBAL`. The input and output are sequences of lines that have types defined for them. The types `inline` and `outline` are tuples ending with the time at which the event have appeared (in `outline` the last two indicate the time which the event has been received and dealt with respectively).

```
class Environment is subclass of GLOBAL

types

public inline  = EventId * MissileType * Angle * Time;
public outline = EventId * FlareType * Angle * Time * Time;
```

Instance variables exist for performing `io` in addition to the `inlines` and `outlines`.

```
-- access to the standard IO library
io : IO := new IO();

inlines : seq of inline := [];

outlines : seq of outline := [];
```

The mappings `ranges` and `sensors` are used for keeping track of the sensors and the angle ranges they are able to observe between.

```
ranges : map nat to (Angle * Angle) := {|->};
sensors : map nat to Sensor := {|->};
inv dom ranges = dom sensors;
```

Finally `evid` and `busy` are used to keep track of the last received event and whether the environment is busy or not respectively.

```
evid : [EventId] := nil;

busy : bool := true;
```

The constructor reads a given scenario and the file name is used as parameter.

```
operations

public Environment: seq of char ==> Environment
Environment (fname) ==
 def mk_ (-,input) = io.freadval[seq of inline](fname) in
   inlines := input;
```

Sensors must be added to the `Environment` using the `addSensor` operation. Note that in order to ensure the state invariant between `ranges` and `sensors` an `atomic` statement is used.

```
public addSensor: Sensor ==> ()
addSensor (psens) ==
  ( dcl id : nat := card dom ranges + 1;
    atomic (
     ranges := ranges munion {id |-> psens.getAperture()};
     sensors := sensors munion {id |-> psens}
    )
  );
```

The `Run` operation create new signals from the environment and steps to the corresponding system reaction until both the system and the environment are finished with their execution.

```
public Run: () ==> ()
Run () ==
 (while not (isFinished() and CM`detector.isFinished()) do
    (createSignal();
     CM`detector.Step();
     World`timerRef.StepTime();
    );
 showResult()
 );
```

The `createSignal` operation is used for extracting input and directing it to a sensor that can observe the given input angle `pa`.

```
private createSignal: () ==> ()
createSignal () ==
  ( if len inlines > 0
    then (dcl curtime : Time := World`timerRef.GetTime(),
              done : bool := false;
          while not done do
            def mk_ (eventid, pmt, pa, pt) = hd inlines in
              if pt <= curtime
              then (for all id in set dom ranges do
                      def mk_(papplhs,pappsize) = ranges(id) in
                        if canObserve(pa,papplhs,pappsize)
                        then sensors(id).trip(eventid,pmt,pa);
                    inlines := tl inlines;
                    done := len inlines = 0)
              else done := true)
    else busy := false);
```

The handleEvent operation is used when the CM classes have coped with an input event and produced an outgoing event that must be stored.

```
public
handleEvent: EventId * FlareType * Angle * Time * Time ==> ()
handleEvent (evid,pfltp,angle,pt1,pt2) ==
  (outlines := outlines ^ [mk_ (evid,pfltp, angle,pt1, pt2)] );
```

The showResult operation is used to write out the overall result.

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

Finally the isFinished operation is used to check if the Environment is finished with its execution.

```
public isFinished : () ==> bool
isFinished () ==
  return inlines = [] and not busy;

end Environment
```

3.5.5 The Sensor Class

The Sensor class is used to model the sensor hardware from the environment. It contains an instance variable aperture that models the left hand-side of the viewing angle of the sensor.

```
class Sensor is subclass of GLOBAL

instance variables

-- the missile detector this sensor is connected to
private detector : MissileDetector;

-- the left hand-side of the viewing angle of the sensor
private aperture : Angle;
```

The constructor initialises the aperture instance variable and the getAperture operation uses this information.

```
operations

public Sensor: MissileDetector * Angle ==> Sensor
Sensor (pmd, psa) == (detector := pmd; aperture := psa);

public getAperture: () ==> GLOBAL`Angle * GLOBAL`Angle
getAperture () == return mk_ (aperture, SENSOR_APERTURE);
```

The `trip` operation is called from the `Environment` to signal an event. The sensor triggers the missile detector for further processing using the `addThreat` operation. Note that the caller must ensure that the sensor must be able to observe the given event.

```
public trip: EventId * MissileType * Angle ==> ()
trip (evid, pmt, pa) ==
  -- log and time stamp the observed threat
  CM`detector.addThreat(evid, pmt,pa,World`timerRef.GetTime())
pre canObserve(pa, aperture, SENSOR_APERTURE)

end Sensor
```

3.5.6 The Missile Detector Class

The primary task of the `MissileDetector` class is to collect all sensor data and dispatch each event to the appropriate `FlareController` instance.

In the same way as the `Environment` class instance variables are used for keeping track of the `ranges` and the `controllers`.

```
class MissileDetector is subclass of GLOBAL

instance variables

ranges : map nat to (Angle * Angle) := {|->};
controllers : map nat to FlareController := {|->};
inv dom ranges = dom controllers;
```

The `threats` instance variable collects the observations from all attached sensors and `busy` keeps track of the status of the `MissileDetector`.

```
threats : seq of (EventId * MissileType * Angle * Time) := [];

busy : bool := false
```

The `addController` operation is only used to instantiate the model

```
operations

public addController: FlareController ==> ()
addController (pctrl) ==
  (dcl nid : nat := card dom ranges + 1;
   atomic
    (ranges := ranges munion {nid |-> pctrl.getAperture()};
     controllers := controllers munion {nid |-> pctrl}
    );
  );
```

The operation `Step` is used to "step" the algorithm: it takes threats and finds the right controller to relay the threat to and calls `addThreat` on the appropriate controller instance. It also

need to ensure that all controllers will take a `Step` in their own processing.

```
public Step: () ==> ()
Step() ==
  (if threats <> []
   then def mk_ (evid,pmt, pa, pt) = getThreat() in
          for all id in set dom ranges do
            def mk_(papplhs, pappsize) = ranges(id) in
              if canObserve(pa, papplhs, pappsize)
              then controllers(id).addThreat(evid,pmt,pa,pt);
    busy := len threats > 0;
    for all id in set dom controllers do
      controllers(id).Step()
  );
```

The `addThreat` operation is used to modify the event list. In the current model events are stored first come first served, but one could imagine using a different ordering instead.

```
public addThreat: EventId * MissileType * Angle * Time ==> ()
addThreat (evid,pmt,pa,pt) ==
  (threats := threats ^ [mk_ (evid,pmt,pa,pt)];
   busy := true );
```

The `getThreat` operation is a local helper operation to modify the event list. Finally the `isFinished` operation is defined such that it is finished when all the controllers are finished.

```
private getThreat: () ==> EventId * MissileType * Angle * Time
getThreat () ==
  (dcl res : EventId * MissileType * Angle * Time
           := hd threats;
   threats := tl threats;
   return res );

public isFinished: () ==> bool
isFinished () ==
  return forall id in set dom controllers &
            controllers(id).isFinished()

end MissileDetector
```

3.5.7 The Flare Controller Class

The job of the `FlareController` class is to determine which flare dispenser to relay the incoming threat to the right flare dispenser.

Like the `Sensor` class it has `aperture` as an instance variable to keep track of the left-hand-side of its working angle. In addition, it maintains a link to all its flare dispensers using the same scheme as for `MissileDetector` and the `Environment` classes.

```
class FlareController is subclass of GLOBAL
```

```
instance variables

private aperture : Angle;

ranges : map nat to (Angle * Angle) := {|->};
dispensers : map nat to FlareDispenser := {|->};
inv dom ranges = dom dispensers;
```

Just like the MissileDetector class it includes the threats and busy instance variables.

```
-- the relevant events to be treated by this controller
threats : seq of (EventId * MissileType * Angle * Time) := [];

-- the status of the controller
busy : bool := false
```

The constructor is setting the aperture instance variable in the same fashion as the Sensor class. The addDispenser operation is similar to the addController and addSensor from the MissileDetector and the Environment classes respectively.

```
operations

public FlareController: Angle ==> FlareController
FlareController (papp) == aperture := papp;

public addDispenser: FlareDispenser ==> ()
addDispenser (pfldisp) ==
  let angle = aperture + pfldisp.GetAngle() in
    (dcl id : nat := card dom ranges + 1;
     atomic
     (ranges := ranges munion
                {id |-> mk_(angle, DISPENSER_APERTURE)};
      dispensers := dispensers munion {id |-> pfldisp});
     );
```

The operation Step is used to "step" the algorithm in the same way as the Step operation from the MissileDetector class: it takes threats and finds the right dispenser to relay the threat to and calls addThreat on the appropriate dispenser instance. It also need to ensure that all dispensers will take a Step in their own processing.

```
public Step: () ==> ()
Step() ==
  (if threats <> []
   then def mk_ (evid,pmt, pa, pt) = getThreat() in
          for all id in set dom ranges do
            def mk_(papplhs, pappsize) = ranges(id) in
              if canObserve(pa, papplhs, pappsize)
              then dispensers(id).addThreat(evid,pmt,pt);
   busy := len threats > 0;
   for all id in set dom dispensers do
```

```
    dispensers(id).Step());
```

The `getAperture` operation gets the left hand-side start point and opening angle and yield a pair of angles in the same way as `getAperture` from the `Sensor` class.

```
public getAperture: () ==> GLOBAL`Angle * GLOBAL`Angle
getAperture () == return mk_(aperture, FLARE_APERTURE);
```

Just like the `MissileDetector` class the `addThreat`, `getThreat` and `isFinshed` operations are present here.

```
public addThreat: EventId * MissileType * Angle * Time ==> ()
addThreat (evid,pmt,pa,pt) ==
  (threats := threats ^ [mk_ (evid,pmt,pa,pt)];
   busy := true );

private getThreat: () ==> EventId * MissileType * Angle * Time
getThreat () ==
  (dcl res : EventId * MissileType * Angle * Time
           := hd threats;
   threats := tl threats;
   return res );

public isFinished: () ==> bool
isFinished () ==
  return forall id in set dom dispensers &
           dispensers(id).isFinished();

end FlareController
```

3.5.8 The Flare Dispenser Class

The `FlareDispenser` class is responsible for firing the actual flares according to the plan for the given missile id. The `responseDB` value contains a mapping from `MissileType` to such a `Plan`.

```
class FlareDispenser is subclass of GLOBAL

values

responseDB : map MissileType to Plan =
  {<MissileA> |-> [mk_(<FlareOneA>,900),
                   mk_(<FlareTwoA>,500),
                   mk_(<DoNothingA>,100),
                   mk_(<FlareOneA>,500)],
   <MissileB> |-> [mk_(<FlareTwoB>,500),
                   mk_(<FlareTwoB>,700)],
   <MissileC> |-> [mk_(<FlareOneC>,400),
                   mk_(<DoNothingC>,100),
                   mk_(<FlareTwoC>,400),
```

```
                    mk_(<FlareOneC>,500)] };
```

In the same way the `missilePriority` provides information about the priority for the different types of missiles.

```
missilePriority : map MissileType to Time =
  { <None>     |-> 0,
    <MissileA> |-> 1,
    <MissileB> |-> 2,
    <MissileC> |-> 3 }
```

`Plan`'s are structured as a sequence of `PlanStep`'s.

```
types

public Plan = seq of PlanStep;

public PlanStep = FlareType * Time;
```

`FlareDispenser` contains a number of instance variables for keeping track of the current status of the flare dispenser.

```
instance variables

public curplan : Plan := [];
curprio        : nat  := 0;
busy           : bool := false;
aperture       : Angle;
eventid        : [EventId];
```

The constructor sets the `aperture` instance variable in the same way as for the constructor of the `Sensor` and `FlareController` classes. The `GetAngle` provides the `aperture` in the usual fashion.

```
operations

public FlareDispenser: nat ==> FlareDispenser
FlareDispenser(ang) ==
  aperture := ang;

public GetAngle: () ==> nat
GetAngle() ==
  return aperture;
```

The `Step` operation is similar to what is found in the `MissileDetector` and `FlareController` classes except that here actual flares are released rather than relaying the information to yet another class.

```
public Step: () ==> ()
Step() ==
  if len curplan > 0
```

```
  then (dcl curtime : Time := World`timerRef.GetTime(),
            first : PlanStep := hd curplan,
            next : Plan := tl curplan;
        let mk_(fltp, fltime) = first in
          (if fltime <= curtime
           then (releaseFlare(eventid,fltp,fltime,curtime);
                 curplan := next;
                 if len next = 0
                 then (curprio := 0;
                       busy := false ) )
           )
    );
```

The addThreat operation is used from the Flare-Controller's to insert the event of a new missile detected. Here the priority of the missile is essential to determine whether any existing treatment of another missile shall be interrupted.

```
public addThreat: EventId * MissileType * Time ==> ()
addThreat (evid, pmt, ptime) ==
  if missilePriority(pmt) > curprio
  then (dcl newplan : Plan :=  [],
            newtime : Time := ptime;
        -- construct an absolute time plan
        for mk_(fltp, fltime) in responseDB(pmt) do
          (newplan := newplan ^ [mk_ (fltp, newtime)];
           newtime := newtime + fltime );
        -- immediately release the first action
        def mk_(fltp, fltime) = hd newplan;
            t = World`timerRef.GetTime() in
          releaseFlare(evid,fltp,fltime,t);
        -- store the rest of the plan
        curplan := tl newplan;
        eventid := evid;
        curprio := missilePriority(pmt);
        busy := true )
pre pmt in set dom missilePriority and
    pmt in set dom responseDB;
```

The releaseFlare operation is simply communicating to the environment using the operation called handleEvent from the Environment class.

```
private releaseFlare: EventId * FlareType * Time * Time ==> ()
releaseFlare (evid,pfltp, pt1, pt2) ==
  World`env.handleEvent(evid,pfltp,aperture,pt1,pt2);
```

Finally the isFinished operation is defined in terms of checking whether the operation FlareDispenser is busy.

```
public isFinished: () ==> bool
isFinished () ==
  return not busy
```

```
end FlareDispenser
```

3.5.9 The Timer Class

The `Timer` class is used to control evolution of time throughout the sequential model. To get an unambiguous notion of time, it is important that only a single instance of this class exists. For this reason, the class is following the singleton pattern. The constructor of the class is `private`, so all classes in the system that needs to access time must use the operation `GetInstance` to get the singleton instance.

```
class Timer

instance variables

private static timerInstance : Timer := new Timer();

operations

private Timer: () ==> Timer
Timer() ==
  skip;

public static GetInstance: () ==> Timer
GetInstance() ==
  return timerInstance;
```

The `Timer` has one instance variable called `currentTime` representing the current time in the system.

```
class Timer

instance variables

currentTime : nat := 0;
```

Time is incremented in the system in units of the constant value `stepLength`.

```
values

stepLength : nat = 100;
```

The operation `StepTime` is used to progress time in the system.

```
public StepTime : () ==> ()
StepTime() ==
  currentTime := currentTime + stepLength;
```

The operation `GetTime` is used to read the current time.

```
public GetTime : () ==> nat
GetTime() ==
  return currentTime;

end Timer
```

3.5.10 The IO Class

The class `IO` is the VDM++ standard input/output library. It needs no further explanation.

```
class IO

--      Overture STANDARD LIBRARY: INPUT/OUTPUT
--      ---------------------------------------
--
-- Standard library for the Overture Interpreter. When the interpreter
-- evaluates the preliminary functions/operations in this file,
-- corresponding internal functions is called instead of issuing a run
-- time error. Signatures should not be changed, as well as name of
-- module (VDM-SL) or class (VDM++). Pre/post conditions is
-- fully user customisable.
-- Dont care's may NOT be used in the parameter lists.
--
-- The in/out functions  will return false if an error occurs. In this
-- case an internal error string will be set (see 'ferror').

types

public
filedirective = <start>|<append>

functions

-- Write VDM value in ASCII format to std out:
public
writeval[@p]: @p -> bool
writeval(val)==
  is not yet specified;

-- Write VDM value in ASCII format to file.
-- fdir = <start> will overwrite existing file,
-- fdir = <append> will append output to the file (created if
-- not existing).
public
fwriteval[@p]:seq1 of char * @p * filedirective -> bool
fwriteval(filename,val,fdir) ==
  is not yet specified;

-- Read VDM value in ASCII format from file
public
```

```
freadval[@p]:seq1 of char -> bool * [@p]
freadval(f) ==
  is not yet specified
  post let mk_(b,t) = RESULT in not b => t = nil;

operations

-- Write text to std out. Surrounding double quotes will be stripped,
-- backslashed characters should be interpreted.
public
echo: seq of char ==> bool
echo(text) ==
  fecho ("",text,nil);

-- Write text to file like 'echo'
public
fecho: seq of char * seq of char * [filedirective] ==> bool
fecho (filename,text,fdir) ==
  is not yet specified
  pre filename = "" <=> fdir = nil;

-- The in/out functions  will return false if an error occur. In this
-- case an internal error string will be set. 'ferror' returns this
-- string and set it to "".
public
ferror:()  ==> seq of char
ferror () ==
  is not yet specified;

-- New simplified format printing operations
-- The questionmark in the signature simply means any type
public static print: ? ==> ()
print(arg) ==
  is not yet specified;

-- New simplified format printing operations
-- The questionmark in the signature simply means any type
public static printf: seq of char * seq of ? ==> ()
printf(format, args) ==
  is not yet specified;

end IO
```

3.5.11 Validation of the Model

As for the VDM-SL model there is a need to validate the correct behaviour of the sequential VDM++ model presented above. In this case the input and output is done using the standard `IO` class. Thus one needs to make updates to the `scenario.txt` file to test the model with new possible input. Again one would start this by providing simple test cases and then gradually make

them more and more complex. In this case the validation of the correct behaviour is complicated by the `FlareDispenser`'s coping with different angles (because their angles are relative to the `FlareController` they are assigned to) will report their own angle only. We leave it as an exercise for the reader to update the model to yield the real angle (combining the angle for the `FlareController` and the `FlareDispenser`).

There is a rather complex `scenario.txt` available in the electronic version of this model on the web that is sufficient to ensure that all parts of the model have been exercised at least once just by interpreting `new World().Run()`. Again is this documented using the test coverage feature from Overture. In this sequential model in VDM++ there is still a considerable amount of abstracting away from different timing delays. However, it still makes sense to inspect the specific timing requirements for the system (listed as item 3 and 4 on page 40). Both of these timing requirements are also validated with the given scenarios.

3.5.12 Summary

In Section 3.5 a more design-oriented model of the counter measures system has been presented. This model breaks the system into its static components (classes) but it does not yet deal with the dynamic architecture (threads). The main virtue of this model is the emphasis on functionally correct behaviour using the envisaged static components. The precision of the model here enables validation using traditional testing techniques.

This model was tested with test cases conceptually identical to those used for the VDM-SL model in Section 3.3. In this way the old test cases were reused during host acceptance testing, and the VDM-SL model was used as an oracle to compare the functional behaviour of this sequential VDM++ model. Whenever one is reusing such test cases it may be necessary to include more test cases to achieve the desired test coverage for the new model.

What can be learnt from this sequential design model is that we now have a common understanding of the general structural break down of the counter measures system into its static components. In addition we have validated that the new model functionally corresponds to the behaviour described at the more abstract level in Section 3.3.

3.6 Concurrent VDM++ Design Model

In this section the focus will be on the differences that has been made to the sequential counter measures model to achieve a concurrent design model. A full listing of all the classes is present in Appendix C.3 and an overview of the new class diagram is given in Figure 3.7.

From the sequential model four of the classes will get active threads because they have independent computations that can be carried out. These are the `Environment`, the `MissileDetector`, the `FlareController` and the `FlareDispenser` classes. Note that this includes the class that had the overall control in the sequential model (`Environment`) and all the classes that had a `Step` operation.

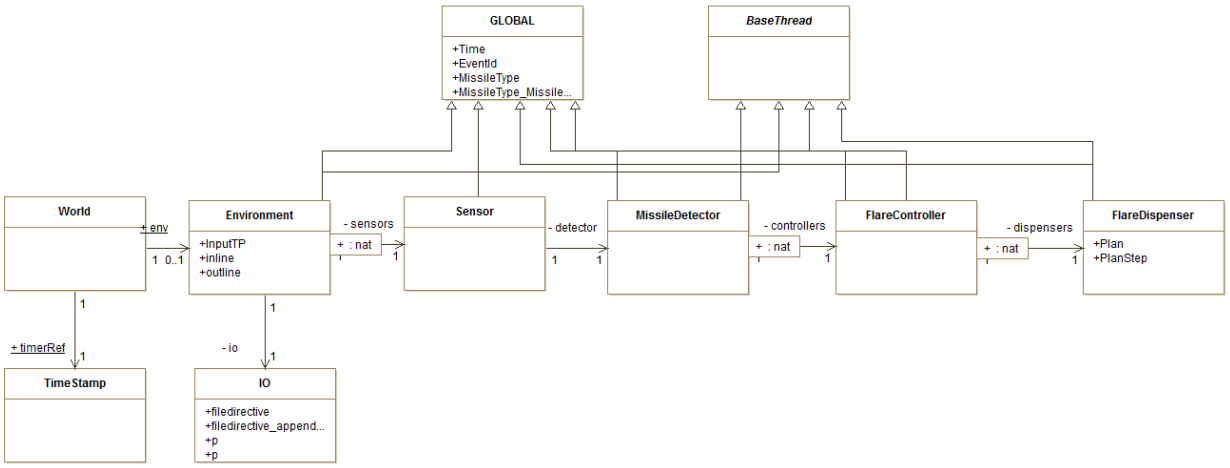

Figure 3.7: Class Diagram for the Concurrent Counter Measures Model

In addition to these changes the way timing is treated is changed so the `Timer` class is replaced with a more elaborate treatment of time. This is done using the `TimeStamp` class and the `BaseThread` class, which may be reused directly for other concurrent VDM++ models following the same principles.

Communication occurs between the sensor and the missile detector (unidirectional) and between the missile detector and the flare controller (bidirectional). To ensure correct sequencing of the communication, synchronization is used.

We now present the concurrent model on a class by class basis. The classes `GLOBAL`, `Sensor` and `IO` are unchanged from the previous model, so are not given again here.

3.6.1 Introducing the BaseThread and TimeStamp Classes

In order to ensure that the different active instances get a chance of stepping in sync with each other both a `BaseThread` and a `TimeStamp` class needs to be introduced. The `BaseThread` class is used as a superclass for all active classes that need a chance to take a step. It is able to handle both periodic as well as aperiodic threads. It is defined as:

```
class BaseThread

types

public static ThreadDef ::
  p : nat1
  isP : bool;

instance variables

protected period : nat1 := 1;
```

```
protected isPeriodic : bool := true;

protected registeredSelf : BaseThread;
protected timeStamp : TimeStamp := TimeStamp`GetInstance();

operations

protected BaseThread : BaseThread ==> BaseThread
BaseThread(t) ==
 (registeredSelf:= t;
  timeStamp.RegisterThread(registeredSelf);
  if(not timeStamp.IsInitialising())
  then start(registeredSelf);
 );
```

Note how threads get started in the constructor if we are not initialising the overall collection of classes. For this to happen correctly, all classes inheriting from `BaseThread` must call the super-constructor explicitly passing a reference to `self`. An example of this can be seen in the `Environment` class, which also shows how to change the default values of the instance variables `p` specifying the length of a period and `isP` that specifies if the thread is periodic or not.

```
class Environment is subclass of BaseThread

operations

public Environment: seq of char * [ThreadDef] ==> Environment
Environment (fname, tDef) ==
 (if tDef <> nil
  then (period := tDef.p;
        isPeriodic := tDef.isP;
       );
  BaseThread(self);
  );
```

All subclasses of the `BaseThread` class need to have a `Step` operation implementing the abstract operation defined in `BaseThread`.

```
Step : () ==> ()
Step() ==
  is subclass responsibility
```

The active thread depends upon whether we have a periodic thread or not:

```
thread
 (if isPeriodic
  then (while true
        do
         (Step();
          timeStamp.WaitRelative(period);
         )
       )
```

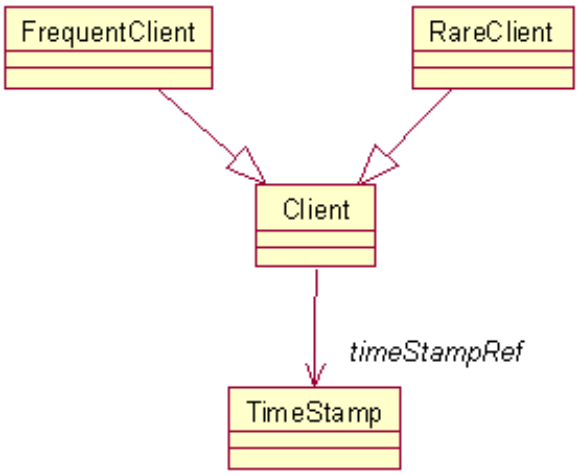

Figure 3.8: Class Diagram for Time Stamp Pattern

```
  else (Step();
        World`timerRef.WaitRelative(0);
        timeStamp.UnRegisterThread(registeredSelf);
       )
 );

end BaseThread
```

The `TimeStamp` singleton class maintains a map from thread ids to time (`wakeUpMap`), representing when a particular thread should be woken. The other instance variable in the class represents the current time. Note that this concept is orthogonal to the notion of simulated time described earlier in this document.

```
class TimeStamp

values

public stepLength : nat = 1;

instance variables

currentTime  : nat   := 0;
wakeUpMap    : map nat to [nat] := {|->};
barrierCount : nat := 0;
registeredThreads : set of BaseThread := {};
isInitialising : bool := true;
-- singleton instance of class
private static timeStamp : TimeStamp := new TimeStamp();
```

To get the singleton instance of the `TimeStamp` class the operation `GetInstance` is called.

```
operations

-- private constructor (singleton pattern)
private TimeStamp : () ==> TimeStamp
TimeStamp() ==
  skip;

-- public operation to get the singleton instance
public static GetInstance: () ==> TimeStamp
GetInstance() ==
  return timeStamp;
```

Whenever a thread is started it gets registered with a reference to it. This happens automatically in the constructor of the `BaseThread`.

```
public RegisterThread : BaseThread ==> ()
RegisterThread(t) ==
 (barrierCount := barrierCount + 1;
  registeredThreads := registeredThreads union {t};
 );

public UnRegisterThread : BaseThread ==> ()
UnRegisterThread(t) ==
 (barrierCount := barrierCount - 1;
  registeredThreads := registeredThreads \ {t};
 );
```

When all threads have registered themselves and initialisation is finished all these threads can be started by explicitly calling the operation `DoneInitialising`.

```
public IsInitialising: () ==> bool
IsInitialising() ==
  return isInitialising;

public DoneInitialising: () ==> ()
DoneInitialising() ==
 (if isInitialising
  then (isInitialising := false;
        for all t in set registeredThreads
        do
          start(t);
       );
 );
```

To model periodic behaviour, a client can be put to sleep using `WaitRelative`.

```
public WaitRelative : nat ==> ()
WaitRelative(val) ==
  WaitAbsolute(currentTime + val);
```

Absolute waits are performed using `WaitAbsolute`. Note that if time given is less than the current time, then the client will never be woken.

```
public WaitAbsolute : nat ==> ()
WaitAbsolute(val) ==
 (AddToWakeUpMap(threadid, val);
  -- Last to enter the barrier notifies the rest.
  BarrierReached();
  -- Wait till time is up
  Awake();
);
```

`AddToWakeUpMap` is used to add new waits to the `wakeUpMap`.

```
AddToWakeUpMap : nat * nat ==> ()
AddToWakeUpMap(tId, val) ==
  wakeUpMap := wakeUpMap ++ { tId |-> val };
```

The operation `BarrierReached` evaluates the `wakeUpMap` when all period threads have entered the mapping. Time is incremented and all threads that needs to be awoken is removed from the `wakeUpMap`.

```
BarrierReached : () ==> ()
BarrierReached() ==
  while  (card dom wakeUpMap = barrierCount) do
   (currentTime := currentTime + stepLength;
    let threadSet : set of nat = {th | th in set dom wakeUpMap
                                     & wakeUpMap(th) <> nil and
                                       wakeUpMap(th) <= currentTime }
    in
      for all t in set threadSet
      do
         wakeUpMap := {t} <-: wakeUpMap;
   )
post forall x in set rng wakeUpMap & x = nil or x >= currentTime;
```

All threads block on the operation `Awake` until they are removed from the `wakeUpMap` as described above.

```
operations

Awake: () ==> ()
Awake() == skip;

sync
  per Awake => threadid not in set dom wakeUpMap;
```

The current time of the class may be obtained via the `GetTime` operation.

```
public GetTime : () ==> nat
GetTime() ==
  return currentTime;
```

Since `barrierCount`, `registeredThreads` and `wakeUpMap` is manipulated by a number of different operations, we need to set access to them to be mutually exclusive.

```
sync
  mutex(IsInitialising);
  mutex(DoneInitialising);
  mutex(AddToWakeUpMap);
  mutex(NotifyThread);
  mutex(BarrierReached);
  mutex(AddToWakeUpMap, NotifyThread);
  mutex(AddToWakeUpMap, BarrierReached);
  mutex(NotifyThread, BarrierReached);
  mutex(AddToWakeUpMap, NotifyThread, BarrierReached);

end TimeStamp
```

3.6.2 Updating the CM Class

All constructors from classes that need to be active must be updated with extra parameters indicating if they are periodic and if so what the required period is. This update of the constructors needs to be used inside the `CM` class. In this case we wish them all to be periodic with a period of one time unit. Thus CM becomes:

```
class CM

instance variables

-- maintain a link to the detector
public static detector : MissileDetector := new MissileDetector(1, true);

-- sensors are unchanged
public static controller0 : FlareController := new FlareController(0, 1, true);
public static controller1 : FlareController :=
                             new FlareController(120, 1, true);
public static controller2 : FlareController :=
                             new FlareController(240, 1, true);

public static dispenser0 : FlareDispenser := new FlareDispenser(0, 1, true);
public static dispenser1 : FlareDispenser := new FlareDispenser(30, 1, true);
public static dispenser2 : FlareDispenser := new FlareDispenser(60, 1, true);
public static dispenser3 : FlareDispenser := new FlareDispenser(90, 1, true);

public static dispenser4 : FlareDispenser := new FlareDispenser(0, 1, true);
public static dispenser5 : FlareDispenser := new FlareDispenser(30, 1, true);
public static dispenser6 : FlareDispenser := new FlareDispenser(60, 1, true);
public static dispenser7 : FlareDispenser := new FlareDispenser(90, 1, true);

public static dispenser8 : FlareDispenser := new FlareDispenser(0, 1, true);
public static dispenser9 : FlareDispenser := new FlareDispenser(30, 1, true);
```

```
public static dispenser10 : FlareDispenser := new FlareDispenser(60, 1, true);
public static dispenser11 : FlareDispenser := new FlareDispenser(90, 1, true);

end CM
```

3.6.3 Updating the World Class

The `timerRef` instance variable is changed from a reference to the `Timer` class to a reference to the `TimeStamp` singleton class. This is done in order to synchronize properly between the different independent threads.

However the `Run` operation is drastically changed to indicate that now initialisation is complete. Thus, it now becomes:

```
-- the run function blocks the user-interface thread
-- until all missiles in the file have been processed
public Run: () ==> ()
Run () ==
  (timerRef.DoneInitialising();
   -- wait for the environment to handle all input
   env.isFinished();
   -- wait for the missile detector to finish
   CM`detector.isFinished();
   -- print the result
   env.showResult())
```

Note how this new definition relies upon the different operations having permission predicates defined such that they are blocked until they are indeed ready to perform the desired action.

3.6.4 Updating the Environment Class

The `Environment` class needs to be declared active so it must now inherit from the `BaseThread` class:

```
class Environment is subclass of GLOBAL, BaseThread
```

In addition the `Run` operation is replaced with a `Step` operation. This looks like:

```
public Step : () ==> ()
Step() ==
 (if World`timerRef.GetTime() < simtime
  then createSignal()
  else busy := false;
 );
```

In essence `Step` here is similar but a bit simpler than the `Run` operation. Note that a new instance variable called `simtime` is introduced. It is defined as:

```
simtime : Time;
```

and it is initialised in the Environment constructor. This is used to simulate how long time we at most wish to simulate the entire system. Actually in a case like the CM example where the system only needs to respond to input events present in the Environment it is not strictly needed, but since we would also like to be able to describe reactive systems where the system to be designed is able to react to the absence of events from the Environment this kind of principle needs to be introduced because in that case it would be needed. The constructor is also extended with information about its period so it now looks like:

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

  if tDef <> nil
  then (period := tDef.p;
        isPeriodic := tDef.isP;
       );
  BaseThread(self);
  );
```

where period and isPeriodic are new instance variables declared in the BaseThread superclass.

The isFinished operation is also simplified by moving its predicate into a permission predicate. Finally a synchronisation constraint is made for the operations called handleEvent and createSignal ensuring that they cannot be invoked if they are already running (the **mutex** predicates in the **sync** section below).

```
public isFinished : () ==> ()
isFinished () == skip;

sync

mutex (handleEvent);
mutex (createSignal);
per isFinished => not busy;
```

3.6.5 Updating the Missile Detector Class

Just like for the Environment class the MissileDetector class needs to inherit from the BaseThread class:

```
class MissileDetector is subclass of GLOBAL, BaseThread
```

In the same way the constructor is extended with the periodic information.

The Step operation is split into a test and an auxiliary operation called processSensor:

```
public Step: () ==> ()
```

```
Step() ==
  if len threats > 0
  then processSensor();

processSensor: () ==> ()
processSensor() ==
( def mk_ (evid,pmt, pa, pt) = getThreat() in
     for all id in set dom ranges do
       def mk_(papplhs, pappsize) = ranges(id) in
         if canObserve(pa, papplhs, pappsize)
         then controllers(id).addThreat(evid,pmt,pa,pt);
   busy := len threats > 0);
```

The operation `isFinished` is different from the sequential model, since it uses a permission predicate to block until the thread has finished. This means that the `forall` quantification is changed to a loop over all the `controllers`.

```
public isFinished: () ==> ()
isFinished () ==
  for all id in set dom controllers do
    controllers(id).isFinished()
```

From a synchronisation point of view the `Step` and `processSensor` operations need to be protected. In the same way the `addThreat` and `getThreat` operations modify the same instance variables and therefore they need to be declared mutual exclusive using a **`mutex`** predicate. In addition a permission predicate is introduced for the `getThreat` and the `isFinished` operations. `getThreat` is used as a "blocking read" from the main thread of control of the `MissileDetector`.

```
sync

mutex (Step);
mutex (processSensor);
mutex (addThreat,getThreat);

per getThreat => len threats > 0;
per isFinished => not busy
```

3.6.6 Updating the Flare Controller Class

Just like for the `MissileDetector` class also needs to inherit from the `BaseThread` class. In addition the `Step` operation is adjusted in a fashion similar to what we saw for the `Missile-Detector` class:

```
operations

public Step: () ==> ()
Step() ==
```

```
  if len threats > 0
  then processThreat();

processThreat: () ==> ()
processThreat() ==
  (def mk_ (evid,pmt, pa, pt) = getThreat() in
      for all id in set dom ranges do
        def mk_(papplhs, pappsize) = ranges(id) in
          if canObserve(pa, papplhs, pappsize)
          then dispensers(id).addThreat(evid,pmt,pt);
    busy := len threats > 0 );
```

The operation `isFinished` is different from the sequential model, since it uses a permission predicate to block until the thread has finished. This means that the **`forall`** quantification is changed to a loop over all the `dispensers`.

```
public isFinished: () ==> ()
isFinished () ==
  for all id in set dom dispensers do
    dispensers(id).isFinished();
```

From a synchronisation point of view `addThreat` and `getThreat` modify the same instance variables and therefore they need to be declared mutual exclusive using a **`mutex`** predicate just like for the `MissileDetector` class above. In addition a permission predicate is introduced for the `getThreat` and the `isFinished` operations. `getThreat` is used as a "blocking read" from the main thread of control of the `FlareController`.

```
sync

mutex (addThreat,getThreat);
mutex (Step);
mutex (processThreat);

per getThreat => len threats > 0;
per isFinished => not busy
```

3.6.7 Updating the Flare Dispenser Class

Just like for the `FlareController` class also needs to inherit from the `BaseThread` class. In addition the `Step` operation is adjusted in a fashion similar to what we saw for the `Flare-Controller` class: The majority of the functionality is placed inside a new operation called `evalQueue`.

```
public Step: () ==> ()
Step() ==
  evalQueue();

private evalQueue: () ==> ()
```

```
evalQueue () ==
 (if len curplan > 0
  then (dcl curtime : Time := World`timerRef.GetTime(),
            done : bool := false;
       while not done do
         (dcl first : PlanStep := hd curplan,
              next : Plan := tl curplan;
          let mk_(fltp, fltime) = first in
            (if fltime <= curtime
             then (releaseFlare(eventid,fltp,fltime,curtime);
                   curplan := next;
                   if len next = 0
                   then (curprio := 0;
                         done := true;
                         busy := false ) )
             else done := true ) ) ) );
```

As previously the `isFinished` operation is changed by turning the predicate into a permission predicate. From a synchronisation point of view a **`mutex`** predicate is used for the `addThreat` and the `evalQueue` operations.

```
public isFinished: () ==> ()
isFinished () ==
  skip

sync

mutex (addThreat,evalQueue);
per isFinished => not busy
```

3.6.8 Validation of the Model

As for the sequential VDM++ model presented earlier there is a need to validate the correct behaviour of the concurrent VDM++ model presented above. Both of these models use the standard `IO` class for input and output. As previously one would start this by providing simple test cases and then gradually make them more and more complex. Since the input format used for the concurrent VDM++ model is identical to the one used for the sequential VDM++ model all the test cases with different `scenario.txt` files can be reused without any changes. The issue with the relative angles for the `FlareDispenser`'s are still present so the interested reader can do the same kind of update to this model.

As for the sequential VDM++ model the electronic version of this model available on the web there is a rather complex `scenario.txt` that is sufficient to ensure that all parts of the model have been exercised at least once just by interpreting `new World().Run()`. Again is this documented using the test coverage feature from Overture.

3.6.9 Summary

In Section 3.6 a concurrent design-oriented model of the counter measures system has been presented. This model reuses the breakdown into its static components (classes) from Section 3.5. In addition it adds the dynamic architecture (threads). The main virtue of this model is the introduction of the dynamic architecture while still ensuring the functionally correct behaviour. The precision of the model here again enables validation using traditional testing techniques.

This model was tested with the test cases used for the VDM-SL model and the sequential VDM++ model in Sections 3.3 and 3.5. In this way old test cases were reused during system acceptance testing, and the VDM-SL model was used as an oracle to compare the functional behaviour of this concurrent VDM++ model in the same way as for the sequential VDM++ model.

What can be learnt from this concurrent design model is that we now have a common understanding of the dynamic architecture of the system. In addition we have validated that the new model functionally corresponds to the behaviour described at the more abstract levels in Sections 3.3 and 3.5. This validation was again conducted using traditional testing techniques and a large amount of reuse of the test cases from the sequential VDM++ model was possible.

3.7 Real-Time Concurrent and Distributed VDM-RT Design Model

The real-time model differs from the concurrent model in a number of ways:

- The `CM` class is changed into a system (by changing the **`class`** keyword to a **`system`** keyword). In addition static instance variables are introduced for the `CPU`'s and the `BUS`'es that are desired for deployment of the system functionality. Finally a constructor is introduced for the `CM` system which deploys the different statically declared instance variables to the different `CPU`'s.

- A number of the operations from the concurrent model that simply needs to signal another thread are turned asynchronous using the **`async`** keyword.

- Duration statements can be used to indicate portions of the model whose execution time is known from previous experience.

- Cycle statements can be used to indicate portions where the number of clock cycles is known in advance from earlier experience.

Otherwise the model is largely unchanged from the concurrent one. An overview of the different classes can be seen from Figure 3.9.

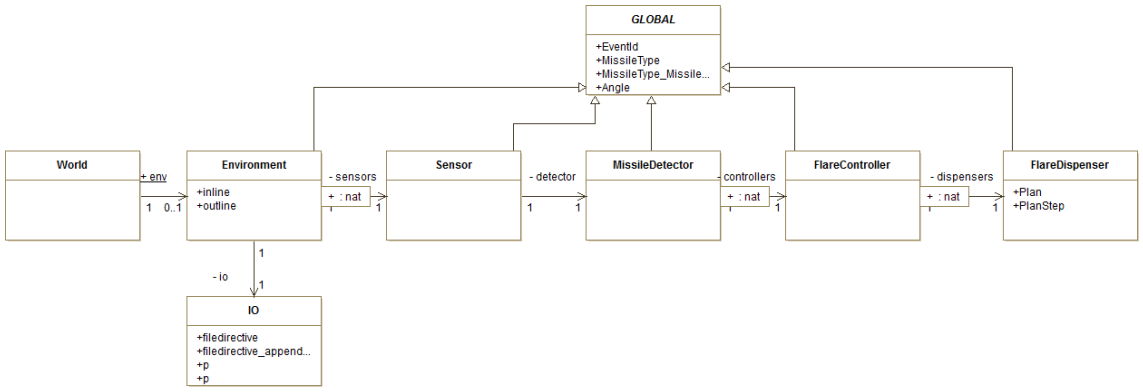

Figure 3.9: Class Diagram for the Real-Time Distributed Counter Measures Model

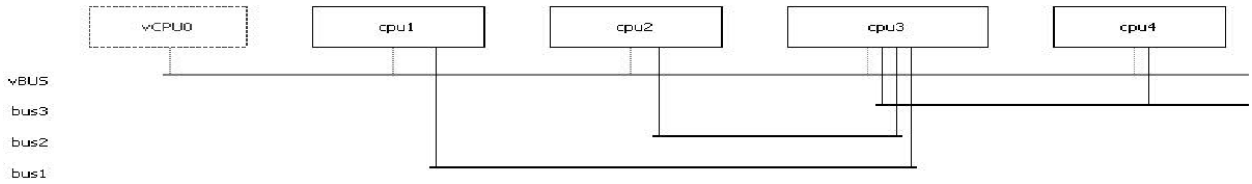

Figure 3.10: CPU Architecture for the Distributed Counter Measures Model

3.7.1 Updating the Counter Measures Class

As explained above the `CM` class have a number of new instance variables. The experimental hardware architecture considered here consists of six `CPU`'s and three `BUS`'es. The declaration of these looks like:

```
instance variables
cpu1 : CPU := new CPU (<FCFS>,1E6);
cpu2 : CPU := new CPU (<FCFS>,1E6);
cpu3 : CPU := new CPU (<FP>,1E9);
cpu4 : CPU := new CPU (<FCFS>,1E6);
cpu5 : CPU := new CPU (<FCFS>,1E6);
cpu6 : CPU := new CPU (<FCFS>,1E6);
bus1 : BUS := new BUS (<FCFS>,1E6,{cpu1,cpu3});
bus2 : BUS := new BUS (<FCFS>,1E6,{cpu2,cpu3});
bus3 : BUS := new BUS (<FCFS>,1E6,{cpu3,cpu4,cpu5,cpu6});
```

A graphical overview of this hardware architecture can be automatically produced using the RealTime Log Viewer feature from Overture. For this rather complex architecture it looks as shown in Figure 3.10.

The `CM` constructor then deploys all static instances to these different `CPU`'s and set priorities for a number of the operations used at the different `CPU`'s.

```
operations

public CM: () ==> CM
CM () ==
  (cpu3.deploy(detector);
   cpu3.setPriority(MissileDetector`addThreat,100);
   cpu1.deploy(sensor0);
   cpu1.setPriority(Sensor`trip,100);
   cpu1.deploy(sensor1);
   cpu2.deploy(sensor2);
   cpu2.setPriority(Sensor`trip,100);
   cpu2.deploy(sensor3);
   cpu3.deploy(controller0);
   cpu3.setPriority(FlareController`addThreat,80);
   cpu4.deploy(dispenser0);
   cpu4.setPriority(FlareDispenser`addThreat,100);
   cpu4.setPriority(FlareDispenser`evalQueue,80);
   cpu4.deploy(dispenser1);
   cpu4.deploy(dispenser2);
   cpu4.deploy(dispenser3);
   cpu3.deploy(controller1);
   cpu5.deploy(dispenser4);
   cpu5.setPriority(FlareDispenser`addThreat,100);
   cpu5.setPriority(FlareDispenser`evalQueue,80);
   cpu5.deploy(dispenser5);
   cpu5.deploy(dispenser6);
   cpu5.deploy(dispenser7);
   cpu3.deploy(controller2);
   cpu6.deploy(dispenser8);
   cpu6.setPriority(FlareDispenser`addThreat,100);
   cpu6.setPriority(FlareDispenser`evalQueue,80);
   cpu6.deploy(dispenser9);
   cpu6.deploy(dispenser10);
   cpu6.deploy(dispenser11);
   )
```

3.7.2 Updating the BaseRTThread Class

The `BaseThread` class used in the concurrent VDM++ model is updated to encompass real-time information. This updated class is called `BaseRTThread`.

```
class BaseRTThread

types

public static ThreadDef ::
  p : nat1
  isP : bool
  j : nat
```

```
  d : nat
  o : nat;

instance variables

protected period : nat1 := 1000E6;
protected isPeriodic : bool := true;
protected jitter : nat := 0;
protected delay : nat := 0;
protected offset : nat := 0;
```

The `ThreadDef` has added parameters for *jitter*, *delay* and *offset*. Additional instance variables for holding values for these real-time parameters have also been added.

The `thread` section of the class has also been updated using the `periodic` definition of threads using the real-time parameters defined above.

```
thread
  periodic(period, jitter, delay, offset)(Step);
```

This has the unfortunate side-effect that the real-time framework described here does not support non-periodic threads. If non-periodic threads are needed in a real-time model, this can be done by specifying the non-periodic behaviour in the `thread` section, and explicitly starting the non-periodic thread class using the `start` keyword.

An example of the use of these added real-time parameters can be seen in the `CM` system class.

```
system CM

instance variables

public static dispenser0 : FlareDispenser := new FlareDispenser(0,
                           mk_BaseRTThread`ThreadDef(1000E6,true,0,0,0));
```

3.7.3 Updating the RTTimeStamp Class

The `TimeStamp` has been simplified since periodic threads are supported using the `periodic` keyword, and hence threads does not need to be explicitly put to sleep and awoken. This means that the operations `WaitRelative`, `WaitAbsolute`, `BarrierReached`, `AddToWakeUpMap`, `NotifyThread`, `GetTime`and `ThreadDone` all have been removed from the class. The main difference is in the `sync` section, where less operations have to be synchronised. This simplified class is called `RTTimeStamp`.

```
sync
  mutex (RegisterThread);
  mutex (UnRegisterThread);
  mutex (RegisterThread, UnRegisterThread);
  mutex (IsInitialising);
  mutex (DoneInitialising);
```

3.7.4 Updating the World Class

The only change in the `World` class is the addition of additional parameters in instantiation of the `Environment` class.

```
public World: () ==> World
World () ==
   env := new Environment("scenario.txt",
              mk_BaseRTThread`ThreadDef(1000E6,true,10,900,0));
```

3.7.5 Updating the Environment Class

The constructor of the `Environment` has been updated. All active classes now inherit from `BaseRTThread`, and parameters for *jitter*, *delay* and *offset* can also be passed in the constructor. Just like in the concurrent model, all active threads must pass a reference to the superclass `BaseRTThread` to ensure that all threads are registered correctly.

```
class Environment is subclass of GLOBAL, BaseRTThread

operations

public Environment: seq of char * [ThreadDef] ==> Environment
Environment (fname, tDef) ==
 (def mk_ (-,input) = io.freadval[seq of inline](fname) in
    inlines := input;

  if tDef <> nil
  then (period := tDef.p;
        jitter := tDef.j;
        delay := tDef.d;
        offset := tDef.o;
       );
   BaseRTThread(self);
 );
```

Instead of using the `timerRef` reference the `Environment` class now makes use of the special **`time`** keyword. This can for example be seen in the operation `createSignal` where a **`duration`** statement also has been introduced.

```
private createSignal: () ==> ()
createSignal () ==
  duration (10)
  (if len inlines > 0
   then (dcl curtime : Time := time, done : bool := false;
         while not done do
           def mk_ (eventid, pmt, pa, pt) = hd inlines in
             if pt <= curtime
             then (for all id in set dom ranges do
                     def mk_(papplhs,pappsize) = ranges(id) in
```

```
                    if canObserve(pa,papplhs,pappsize)
                    then sensors(id).trip(eventid,pmt,pa);
                inlines := tl inlines;
                done := len inlines = 0)
            else done := true)
   else busy := false);
```

3.7.6 Updating the Sensor Class

The only change necessary for the `Sensor` class is that the `trip` operation is made asynchronous and the use of the **`time`** keyword instead of the `World`timerRef.GetTime()` expression. Thus, now the `trip` operation looks like:

```
async public trip: MissileType * Angle ==> ()
trip (pmt, pa) ==
  -- log and time stamp the observed threat
  detector.addThreat(pmt,pa,time)
pre canObserve(pa, aperture, SENSOR_APERTURE)
```

3.7.7 Updating the Missile Detector Class

The only update necessary for the `MissileDetector` class is that the `addThreat` operation is made asynchronous.

3.7.8 Updating the Flare Controller Class

Just like for the `MissileDetector` class the only update necessary for the `FlareController` class is that the `addThreat` operation is made asynchronous.

3.7.9 Updating the Flare Dispenser Class

In the `FlareDispenser` class the `addThreat` and the `evalQueue` operations are made asynchronous. In addition, as for a few other classes above there is now a reference to the **`time`** keyword directly. Thus, these two operations now look like:

```
async public addThreat: EventId * MissileType * Time ==> ()
addThreat (evid, pmt, ptime) ==
  if missilePriority(pmt) > curprio
  then (dcl newplan : Plan :=  [],
            newtime : Time := ptime;
        -- construct an absolute time plan
        for mk_(fltp, fltime) in responseDB(pmt) do
          (newplan := newplan ^ [mk_ (fltp, newtime)];
           newtime := newtime + fltime );
        -- immediately release the first action
```

```
        def mk_(fltp, fltime) = hd newplan in
          releaseFlare(evid,fltp,fltime,time);
        -- store the rest of the plan
        curplan := tl newplan;
        eventid := evid;
        curprio := missilePriority(pmt);
        busy := true )
pre pmt in set dom missilePriority and
    pmt in set dom responseDB;

async evalQueue: () ==> ()
evalQueue () ==
  duration (10)
  (if len curplan > 0
   then (dcl curtime : Time := time, done : bool := false;
         while not done do
           (dcl first : PlanStep := hd curplan,
                next : Plan := tl curplan;
            let mk_(fltp, fltime) = first in
              if fltime <= curtime
              then (releaseFlare(eventid,fltp,fltime,curtime);
                    curplan := next;
                    if len next = 0
                    then (curprio := 0;
                          done := true;
                          busy := false ) )
              else done := true ) ) );
```

3.7.10 Validation of the Model

As for the VDM models presented earlier there is a need to validate the correct behaviour of the real-time and distributed VDM-RT model presented above. Both of these models use the standard `IO` class for input and output. As previously one would start this by providing simple test cases and then gradually make them more and more complex. Since the input format used for the real-time and distributed VDM-RT model is identical to the one used for the concurrent VDM++ model all the test cases with different `scenario.txt` files can be reused again without any changes. However, due to the timing now being closer to the "real-world" implementation of this system one needs to expect that the timing of the output is not identical to what was in the previous model. It is then up to the specifier to judge whether the timings give raise to any kinds of bottlenecks. Once again the issue with the relative angles for the `FlareDispenser`'s are still present so the interested reader can do the same kind of update to this model.

As for the previous VDM++ models the electronic version of this model available on the web there is a rather complex `scenario.txt` that is sufficient to ensure that all parts of the model have been exercised at least once just by interpreting `new World().Run()`. Again is this documented using the test coverage feature from Overture.

3.7.11 Summary

In this section we have presented a model of the counter measures system that incorporates timing information and provides a distributed architecture. It should be pointed out that to some extent this model is still an idealization of the real world. Our model is deterministic and assumes hardware performs perfectly and responds within specified intervals, The model also takes quite a restricted view of what external events may occur. These are not serious flaws in the proposed approach for two reasons:

- Many of the assumptions made e.g. concerning determinism and hardware behaviour are also made during host integration testing; the behaviour of the system when these assumptions are broken could only be tested during target integration testing.
- It is in principal possible to model all manner of events. In particular an "unknown" event could be modelled, and the system behaviour described in the presence of such events.

However, it is worthwhile noting that work is undergoing to incorporate the appropriate handling of faulty behaviour in the environment is carried out in the DESTECS project [Broenink&10, Fitzgerald&10b].

Chapter 4

Synchronization

In this chapter issues relating to synchronization of concurrent threads are addressed. In particular the primitives for specifying synchronous access to shared objects are described, and further synchronization mechanisms that build on these primitives are also given.

4.1 Synchronization Primitives

Synchronization in VDM++ is performed using *permission predicates*. A permission predicate is an expression specifying the circumstances in which an operation may be executed.

```
per operation name => guard condition
```

The semantics of a permission predicate is that when a client requests an operation call, that operation's permission predicate is evaluated. If it is true, execution may proceed and the operation may be activated; if it is false, the client is blocked and the scheduler is invoked.

Permission predicates are listed in the sync section of a class. As a convenient abbreviation, the keyword `mutex` may be used to represent mutual exclusion between operations. This is described in more detail below, together with the different kinds of guard conditions.

A guard condition has scope over the instance variables of the class. For example, consider the following specification of a one-place buffer:

```
class Buffer

instance variables
  data : [nat] := nil

operations
  public Put : nat ==> ()
  Put(v) == data := v;

sync
  per Put => data = nil;
  public Get : () ==> nat
  Get() ==
   let n = data in
     (data := nil;
      return n)
  pre data <> nil

sync
  per Get => data <> nil

end Buffer
```

4.1.1 History Counters

A guard condition is also allowed to refer to *history counters*. These describe the history of a particular operation. There are three basic history counters:

#req(op) The number of times op has been requested in a particular object;

#act(op) The number of times op has been activated in a particular object;

#fin(op) The number of times op has completed execution in a particular object.

As a simple example, consider an N-place buffer:

```
class BufferN

values
  N : nat = 5

instance variables

  data : seq of nat := [];
  inv len data <= N

operations
  public Put : nat ==> ()
  Put(v) ==
    data := data ^ [v];
  public Get : () ==> nat
  Get() ==
    let n = hd data in
      (data := tl data;
       return n)
  pre data <> []

sync
  per Put =>
    #fin(Put) - #fin(Get) < N;
  per Get =>
    #fin(Get) < #fin(Put);

end BufferN
```

Note that in this example the permission predicates could have been equally well expressed in terms of the class's instance variables. However, in general, use of such guards allows more sophisticated synchronization predicates. For example we can express the requirement that only N invocations of `Put` may be active at any time:

```
per Put => #fin(Put) - #fin(Get) < N and #fin(Put) = #act(Put);
```

In addition to the basic history counters, two derived history counters exist:

#active(op) The number of currently active instances of op. So
#active(op) = **#act**(op) - **#fin**(op).

 # waiting(op) The number of non-activated requests for op. So
#waiting(op) = **#req**(op) - **#act**(op).

4.1.2 Mutex

It is possible to specify mutual exclusion (*mutex*) between operation invocations using the basic history counters. However it is such a common requirement that the keyword **mutex** may be used as shorthand for a **mutex** predicate.

A **mutex** predicate allows the user to specify either that all operations of the class are to be executed mutually exclusive, or that a list of operations are to be executed mutually exclusive to each other. Operations that appear in one mutex predicate are allowed to appear in other **mutex** predicates as well, and may also be used in the usual permission predicates. Each **mutex** predicate will implicitly be translated to permission predicates using history guards for each operation mentioned in the name list. For instance,

```
sync
  mutex(opA, opB);
  mutex(opB, opC, opD);
  per opD => someVariable > 42;
```

would be translated to the following permission predicates (which are semantically equivalent to the above expression):

```
sync
  per opA => #active(opB) = 0;
  per opB => #active(opA) = 0 and
             #active(opC) + #active(opD) = 0;
  per opC => #active(opB) + #active(opD) = 0;
  per opD => #active(opB) + #active(opC) = 0 and
             someVariable > 42;
```

Note that it is only permitted to have one explicit permission predicate for each operation in each class.

A **mutex(all)** constraint specifies that all of the operations specified in that class *and any superclasses* are to be executed mutually exclusively.

It is clear that using **mutex**, concurrent execution of critical regions can be prevented.

4.2 Wait-Notify

A popular synchronization mechanism is the wait-notify used in the Java programming language [Gosling&00]. To understand this concept, consider two threads `Producer` and `Consumer` which communicate by an unsynchronized shared buffer, `b` as defined below:

```
class UnsyncBuffer

instance variables
  data : [nat] := nil

operations
  public Put : nat ==> ()
  Put(v) ==
    data := v;

  public Get : () ==> nat
  Get() ==
    let n = data in
      (data := nil;
       return n)
  pre data <> nil

end UnsyncBuffer
```

The unsynchronized buffer is used to communicate data between the producer and consumer

threads.

```
-- Producer thread
while true do
  (let v = Produce() in
     b.Put(v))
```

```
-- Consumer thread
while true do
  Consume(b.Get())
```

Of course the problem that arises is that when the consumer thread consumes data, there need not necessarily be any valid data in the buffer. Moreover, if the producer thread generates data "too fast" for the consumer, data could be lost.

A wait-notify object allows access to the buffer to be synchronized across the two threads. Suppose `o` is a wait-notify object.

```
-- Producer thread
while true do
  (let v = Produce() in
     b.Put(v)
   o.Notify();
   o.Wait())
```

```
-- Consumer thread
while true do
  (o.Wait();
   Consume(b.Get());
   o.Notify())
```

Following a call to `o.Wait`, a thread blocks until another thread calls `o.Notify`. Thus in this example we can see that following production of a value, the producer thread places this in the buffer, notifies the consumer thread and then blocks. The consumer thread is blocked until it is notified by the producer thread; it then consumes the data in the buffer, and afterwards notifies the producer thread.

The advantage of using Wait-Notify is that it can be used to synchronize access to an arbitrary shared object; that is, this shared object need not have been designed with sharing in mind. This greatly simplifies specification and design, as consideration of synchronization can thus be safely postponed until design of the dynamic architecture begins.

Care must be taken with the initialization of a wait-notify mechanism, otherwise deadlocks can occur. For instance in the above example, if the producer thread executes first, following its `notify` both producer and consumer are waiting for a `notify`, so this must be supplied externally. Alternatively the threads can be organized in such a way as to force the consumer to be executed first.

4.3 Thread Completion

A common use of permission predicates is to block the default thread until all other threads have completed. For example, consider the model in Figure 4.1.

When `Main` executes, it creates and starts an instance of `SubThread`, but it delivers the contents of the shared object before `SubThread` has necessarily had a chance to place any data in it.

```
class Shared

instance variables
  data : seq of nat := []

operations
  public Put : nat ==> ()
  Put(n) ==
    data := data ^ [n];

  public Get : () ==>
               seq of nat
  Get() == return data

sync

mutex(Put);
mutex(Put,Get)

end Shared

class SubThread

instance variables
  s : Shared

operations

  public Init : Shared ==> ()
  Init(ns) == s := ns

thread
  for i = 1 to 100 do
    s.Put(i * i)

end SubThread

class MainThread

operations

  public Main :
            () ==> seq of nat
  Main() ==
    (dcl s : Shared :=
                new Shared(),
         t : SubThread :=
                new SubThread();
     t.Init(s);
     start(t);
     return s.Get()
    )

end MainThread
```

Figure 4.1: Not Waiting For Thread Completion

In order to overcome this problem, the shared object should be used to determine completion. An operation `IsFinished` should be added to the specification of `Shared`. Calls to this operation should block until it is determined that `SubThread` has completed.

The specification for operation `Main` would then be:

```
public Main : () ==> seq of nat
Main() ==
  ( dcl s : Shared := new Shared(),
        t : SubThread := new SubThread();
    t.Init(s);
    start(t);
    s.IsFinished();
    return s.Get()
  )
```

The determination of thread completion can be achieved implicitly or explicitly. The implicit approach is based on the amount of data generated by the thread. For instance if it is known that the thread will generate 100 values, this can be used to determine completion:

```
class Shared

instance variables
  data : seq of nat := []

operations
  public Put : nat ==> ()
  Put(n) ==
    data := data ^ [n];

  public Get :
           () ==> seq of nat
  Get() == return data;

  public IsFinished :
             () ==> ()
  IsFinished() == skip;

sync

mutex(Put);
mutex(Put,Get);
per IsFinished =>
    #fin(Put) = 100

end Shared
```

In the explicit approach, the thread which is being waited on informs the shared object of its completion:

```
class Shared

instance variables

  data : seq of nat := [];
  finished : bool := false;

operations

  public Put : nat ==> ()
  Put(n) == data := data ^ [n];

  public Get :
           () ==> seq of nat
  Get() == return data;

  public IsFinished :
             () ==> ()
  IsFinished() == skip;

  public Finished : () ==> ()
  Finished() ==
    finished := true

sync
  mutex(Put);
  mutex(Put,Get);
  mutex(Finished);
  per IsFinished => finished

end Shared
```

4.4 Summary

In this chapter difference mechanisms for synchronization have been presented: synchronization primitives using permission predicates, and the Wait-Notify mechanism. In general synchronization based on permission predicates can be extremely powerful, as such predicates are highly expressive. The price of this power is that it can be very difficult to debug models suffering from deadlocks if such permission predicates are used. Therefore it is recommended that wherever possible the wait-notify mechanism is used, as it is easy to debug[1] breakpoints can be placed within the operations of the wait-notify class to help identify any problems.

[1] Alternatively the trace file produced by Overture or VDMTools can be used to get insight into problems such as deadlocks.

Chapter 5

Periodicity

Many real-time systems have an element of periodicity – something which repeats with fixed frequency. In this chapter we describe how periodicity can be modelled.

5.1 Periodic Threads

Threads may be periodic: that is, an operation can be called with fixed frequency. A typical use of this is to poll an external sensor.

```
class Sensor

operations
  public GetData : () ==> nat
  GetData() ==
    is not yet specified

end Sensor

class SensorPoll

instance variables
  lastReading : [nat] := nil;
  sensor : Sensor

operations
  public Init : Sensor ==> ()
  Init(ns) ==
    sensor := ns;

  PollSensor : () ==> ()
  PollSensor() ==
    lastReading :=
       sensor.GetData()

thread
```

```
  periodic (100,10,90,0) (PollSensor)

end SensorPoll

class Main

operations
  public Run : () ==> ()
  Run() ==
  ( dcl sp : SensorPoll :=
              new SensorPoll(),
        s : Sensor := new Sensor();
    sp.Init(s);
    start(sp)
  )

end Main
```

Here the sensor thread `sp` will execute approximately every 100 time units (1 time unit = 1 nanosecond) following execution of the statement **`start`**`(sp)`.

Note that in a simple system such as the one above, `PollSensor` will be invoked approximately every 100 time units. The second parameter to the periodic statement indicates that 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 90 time units between two instances of this periodic thread. Finally, the last parameter indicates that no offset is required for the invocation of this periodic thread. 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.

In systems with more threads, the periodicity merely informs the scheduler when a particular periodic thread is schedulable. Thus in practice it is possible for periodic threads to be delayed. In fact, it is often an interesting property of a model that periodic threads are being delayed, as it could indicate that portions of the model need to be redesigned. Therefore in the time trace file a special event is output if a periodic thread is delayed.

5.1.1 Periodic Threads and Scheduling

Periodic threads are subject to the same scheduling policy as other threads. This has a number of implications for their use. Specifically:

- Even if a periodic thread which has not completed by the end of its period another periodic thread will be made possible at the next period. Thus it is possible to multiple invocations of a periodic thread at the same time. However, as a user one should be careful about such usage, since it means that an increasing number of threads may be produced.

- Under priority-based scheduling, a periodic thread could miss its deadline because a higher priority thread has been scheduled.

5.2 Modelling Periodic Events

Typically, an external event is modelled as an operation invocation. Thus to model a periodic event, it should be modelled as a periodic operation invocation. For example, suppose we wish to model a clock:

```
class Clock

instance variables
  curtime : nat := 0

operations
  public GetTime : () ==> nat
  GetTime() == return curtime;

  Tick : () ==> ()
  Tick() == curtime := curtime + 1

thread
  periodic (100,0,0,0)(Tick)

end Clock
```

An instance of `Clock` has a thread which will be executed every 100 time units. Such a clock could be shared amongst several threads, and used to synchronize behaviour across the sharing threads. Other examples of timers can be found in Section 3.5.9 and in Appendix B.2.

5.3 Statically Schedulable Systems

Systems which are known to be statically schedulable with specific periods can be simply modelled using periodic threads. For example suppose that we have a system with three threads `A`, `B` and `C`, which have frequencies 30, 70 and 110 time units respectively.

```
class A
...
thread
  periodic (30,0,0,0) (OpA)
end A

class B
...
thread
  periodic (70,0,0,0) (OpB)
end B

class C
...
thread
```

```
  periodic (110,0,0,0) (OpC)
end C

class Main

operations
  public Run : () ==> ()
  Run() ==
  ( dcl a : A := new A(),
        b : B := new B(),
        c : C := new C();
    ...
    startlist({a,b,c});
    ...
  )
end Main
```

In case it was desirable to let one of them start later than the others it would be possible to set the offset to a positive value for that one.

5.4 Summary

Systems exhibiting periodicity can be described using VDM in a straightforward manner, using language primitives. Based on these primitives, features such as timers and clocks can be built into models.

Note that in general, if a system is statically schedulable and does not have high CPU utilization requirements, then rate monotonic analysis [Audsley&93, Burns95] may be used to schedule the system. In this case the problems of meeting deadlines etc are automatically resolved, so for such systems rate monotonic analysis is preferable.

Chapter 6

Scheduling Policies

Overture and VDMTools supports a number of different scheduling policies. In this chapter we describe these various policies, and summarize the implications for the model of each particular policy.

At any point during execution Overture and VDMTools employs two complementary scheduling policies: the primary scheduling policy and secondary scheduling policy.

The primary scheduling policy determines when a thread should be descheduled. The secondary scheduling policy determines the order in which the scheduler tries to find the next thread to schedule. We consider each category separately.

6.1 Primary Scheduling Algorithm

Overture and VDMTools offers a choice of primary scheduling algorithm between time limited and cooperative. In general, a thread executes until an operation call is made, for which the permission predicate is false, at which point the thread blocks.

In a cooperative scheduling algorithm, there is no other way in which a thread may be descheduled: each thread is allowed to run to completion.

In a time limited scheduling algorithm no thread is allowed to execute continuously for more than some defined period; when this period is complete the scheduler will deschedule the thread.

Two time limited scheduling algorithms are available: *instruction number scheduling* and *time slice scheduling*. Under instruction number scheduling the amount of time each thread executes for is limited by the number of instructions executed during execution of the thread. Thus under instruction number scheduling, the decisions made by the scheduler are independent of duration statements or the default cycles information. Under time slice scheduling, the amount of time each thread executes for is limited by a fixed amount of simulated time. Thus under this scheduling algorithm the amount of time each thread executes for is affected by duration statements, and also by the default duration information.

6.2 Secondary Scheduling Algorithm

Two secondary scheduling algorithms are supported: round-robin or priority-based.

6.2.1 Round-Robin Scheduling

Under round-robin, the scheduler uses an arbitrary, fixed order in which to find the next thread to execute. Consider the following example. Suppose we have threads `t1`, `t2` and `t3`, and that the order that the scheduler uses is `[t1,t2,t3]`. Suppose further that `t1` is descheduled and at this point `t2` is blocked, and `t3` is schedulable.

The scheduler would first test whether `t2` is schedulable; since it is blocked, it would then check whether `t3` is schedulable. Thus `t3` would be scheduled.

Suppose now that when `t3` is descheduled, `t1` and `t2` are both schedulable. The scheduler would first check `t1`, and since this is schedulable, it would be selected and executed.

Thus under round-robin scheduling a weak fairness property exists [Lamport91]: a thread that is never blocked will eventually be scheduled.

6.2.2 Priority-based Scheduling

Priority-based scheduling is a variant on round-robin scheduling. A numeric priority is assigned to each thread. When the scheduler needs to select the next thread to be scheduled, all the highest priority threads are checked using a standard round-robin; if no schedulable thread is found, then all threads of the second-highest priority are checked using a standard round-robin, and so on.

To illustrate this, consider the following example. Suppose we have threads `a1`, `a2`, `a3`, `b1`, `c1` and `c2`. Threads `a1`, `a2` and `a3` have priority 3; `b1` has priority 2 and `c1` and `c2` have priority 1. Suppose that the round-robin orders used are:

Priority	Order
3	`[a1,a2,a3]`
2	`[b1]`
1	`[c1, c2]`

Consider scheduling in the following situation:

Thread	Status
a1	Blocked
a2	Schedulable
a3	Schedulable
b1	Schedulable
c1	Schedulable
c2	Schedulable

The scheduler examines the highest priority threads first. Thus first thread `a1` would be checked; it is blocked so `a2` would be checked. Since `a2` is schedulable it would be selected.

Now consider another situation:

Thread	Status
a1	Blocked
a2	Blocked
a3	Blocked
b1	Blocked
c1	Blocked
c2	Schedulable

The scheduler examines threads of priority 3 first; they are all blocked. It then examines threads of priority 2; they are also all blocked. Finally it examines threads of priority 1: first `c1` is checked, but it is blocked, then `c2` is checked. Since `c2` is schedulable it is selected.

Note that for threads which are not of the highest priority, there are no fairness properties: it is possible for them to starve.

6.2.3 Priority of Default Thread

Consider the following model shown in Figure 6.1

Suppose that in the Overture or VDMTools interpreter the expression `new B().Main()` is executed using round-robin scheduling. The model will then execute, and a result will be delivered consisting of a sequence with at least 11 elements. The thread initiated by issuing a command in the interpreter is called the default thread.

Consider now the situation in which the model is executed using priority-based scheduling, where `A` has priority 2, and `B` has priority 1. In this case the computation will never terminate, since `B` will never be scheduled due to having a lower priority than `A`, which is always schedulable.

To avoid this problem, Overture and VDMTools implements the following policy: the default thread is always given a strictly higher priority than any other thread in the system, regardless of what priority may actually have been specified for the class from which the default thread is derived. In this way the default thread is always checked first by the scheduler.

```
class A

instance variables
  data : seq of nat := []

operations
  public IsFinished:() ==> ()
  IsFinished() == skip;

  public Get: () ==>
              seq of nat
  Get() == return data

sync
  per IsFinished =>
      len data > 10

thread
  (dcl i : nat := 0;
   while true do
   ( data := data ^ [i];
     i := i + 1 ))

end A

class B

operations
  public Main:() ==> seq of nat
  Main() ==
  ( dcl a : A := new A();
    start(a);
    a.IsFinished();
    a.Get())

end B
```

Figure 6.1: Example illustrating the Priority of the Default Thread

Chapter 7

Time Trace Analysis

In this chapter we describe the format of the time trace files, and the kinds of analysis that may be performed on them.

7.1 Timed Trace Files

The time trace files generated during execution contain information about thread swapping, messages passed between CPUs and operation requests, activations and completions.

7.1.1 Example

An extract from a trace file is shown below as an example.

Each entry in the time trace file consists of an event name, and event information, separated by `->`. Events fall into different categories, object history events, message events, declaration events, deployment events and thread events. The category dictates the event information provided.

Object history events consist of operation requests, activations and completions. The information provided for such events are: the operation called; information whether it is an asynchronous operation; the object reference id and the `CPU` id on which the operation has been called; the class which the object is an instance of; and the time at which the event occurred.

Message events are similar to operation requests except that they identify requests, activations and completions of messages communicated over a `BUS`. These messages are automatically derived by Overture when an operation is to be invoked on an instance that has been deployed to a different `CPU`. Each message is given a unique identification. Thus, it is only the message request events that contain all information about the `BUS` used, the `CPU` it is coming from and to, etc. Importantly it also contains a size attribute that is derived from the values that are passed over as parameters for the operation to be called at the other `CPU`.

```
CPUdecl ->  id: 13 expl: false sys: "none" name: "vCPU 7"
DeployObj ->  objref: 160 clnm: "FlareDispenser" cpunm: 0 time: 728
OpRequest -> id: 1 opname: "FlareController`addDispenser" objref: 138
             clnm: "FlareController" cpunm: 0 async: false time: 736
MessageRequest -> busid: 0 fromcpu: 0 tocpu: 9 msgid: 23
                  callthr: 1 opname: "FlareController`addDispenser"
                  objref: 138 size: 80 time: 736
MessageActivate -> msgid: 23 time: 736
MessageCompleted -> msgid: 23 time: 736
ThreadSwapOut -> id: 15 objref: 152 clnm: "FlareDispenser"
                 cpunm: 9 overhead: 2 time: 736
ThreadCreate -> id: 16 period: false objref: 138
                clnm: "FlareController" cpunm: 9 time: 738
ThreadSwapIn -> id: 16 objref: 138 clnm: "FlareController"
                cpunm: 9 overhead: 2 time: 738
```

Figure 7.1: Example extract from a `logfile`

Declaration events are included in system "classes" where it is possible to declare `CPU`'s and `BUS`'es. In addition `CPU`'s may be implicitly declared when new objects are created inside the `World` class and this is also logged.

Deployment events are logged whenever objects are created. An object may initially be created and deployed at the virtual `CPU` (always with cpuid 0) and then subsequently deployed at a different `CPU`.

Thread events correspond to a thread being created, killed, swapped in or out, or (for periodic threads) a thread which has missed its deadline being swapped in. In this case the thread id is provided, together with the object reference id for the object owning the thread, the class which the object is an instance of, the time of the event, and (for delayed periodic threads) the delay.

7.2 Analysis Tools

Since the time trace file quickly becomes large, it is essential to use tools to analyze them. Here we give some examples of such tools. Tools may be generic or bespoke, that is they may produce general statistic concerning traces, or they may produce application-relevant information. However, most users will initially use a generic tool and only dedicate extra effort to making a bespoke solution when the generic tool is not capable of performing an especially desired analysis.

7.2.1 Generic Analysis Tools

At the moment one external generic analysis tool for trace files produced by Overture exists. This is the RealTime Log Viewer feature from Overture.

The RealTime Log Viewer feature is able to type check a trace file and if it is parsed and checked to be correct it is able to show:

- A system architecture with the `CPU`'s and the connecting `BUS`'es as shown in Figure 3.10.
- An execution overview the illustrates when the different `CPU`'s and `BUS`'es are active (see Figure 7.2 for an example).
- A detailed graphical overview for each `CPU` where the different objects that have been deployed to that `CPU` and the different threads executing are shown (see Figure 7.3 for an example).

7.2.2 Bespoke Analysis Tools

There are of course many different bespoke analysis tools which could be developed for trace files, according to the application domain and the real-time properties of interest. Here we focus on a couple of general approaches.

Use of Standard Text-Processing Tools

Since the time trace file is simply an ASCII file, manipulation of such files is straight-forward (e.g. in Perl [Wall&92]). An example of this is a Perl script which is used to generate an Excel spreadsheet from a trace file from an earlier version of the missile counter measures model, as shown in Figure 7.4. In this figure a graphical depiction of the times at which the various threads in a model are active is shown. Since standard office tools are able to accept comma separated files, writing such a script is simply a matter of processing a stream of data.

Visualization

A common bespoke analysis tool is a visualization of a trace, superimposing the time at which particular events occur on the activity lines of threads. An example of this is shown in Figure 7.5.

Given the preponderance of GUI toolkits such as Java Swing [JavaSwing], Qt [Qt] and Visual Basic [VisualBasic], such visualizations are relatively easy to develop, but certainly more time consuming than using the RealTime Log Viewer feature. Moreover they can be extremely powerful in communicating the behaviour of a model to a domain expert.

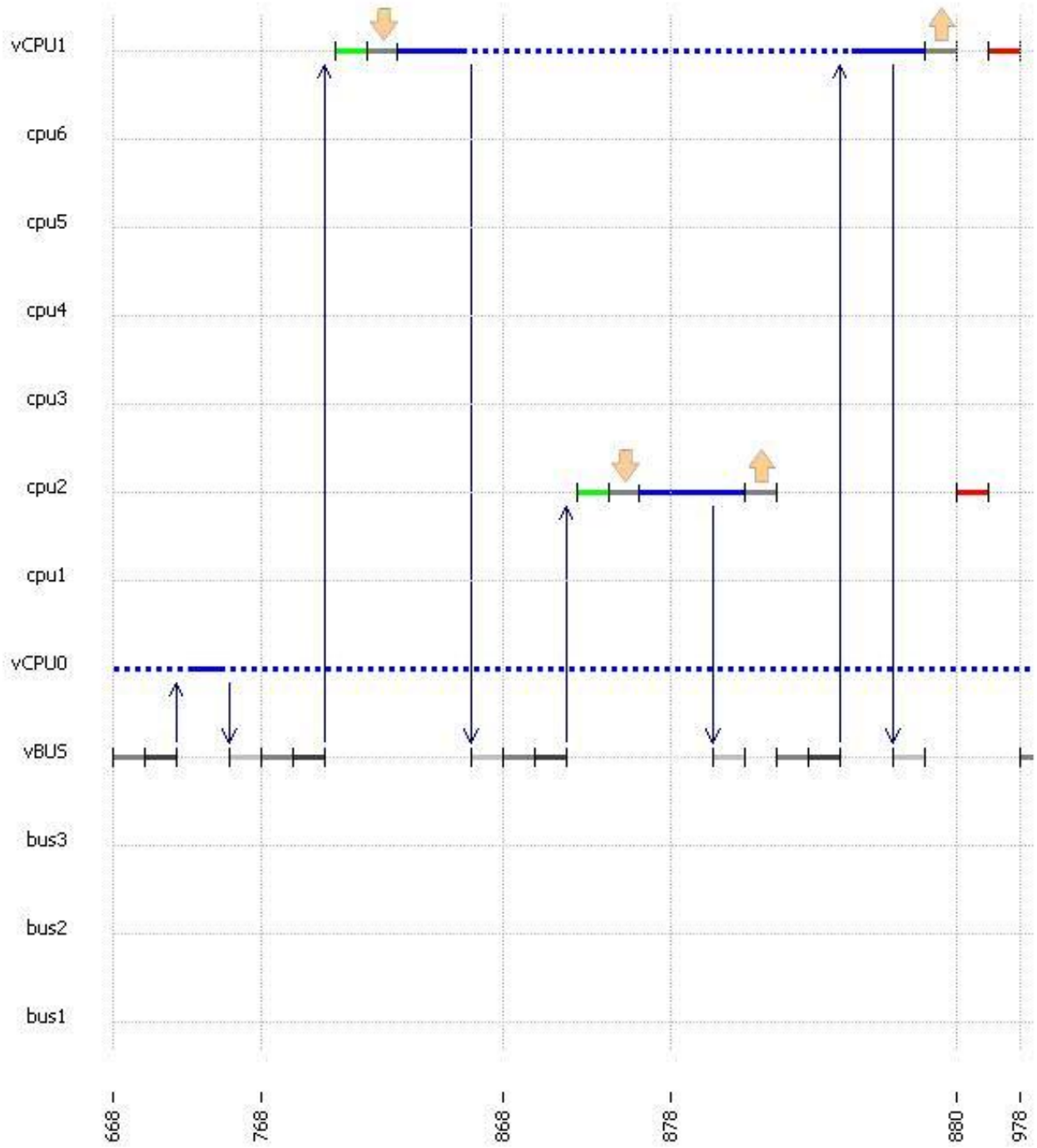

Figure 7.2: An extract from an execution overview for the counter measures example

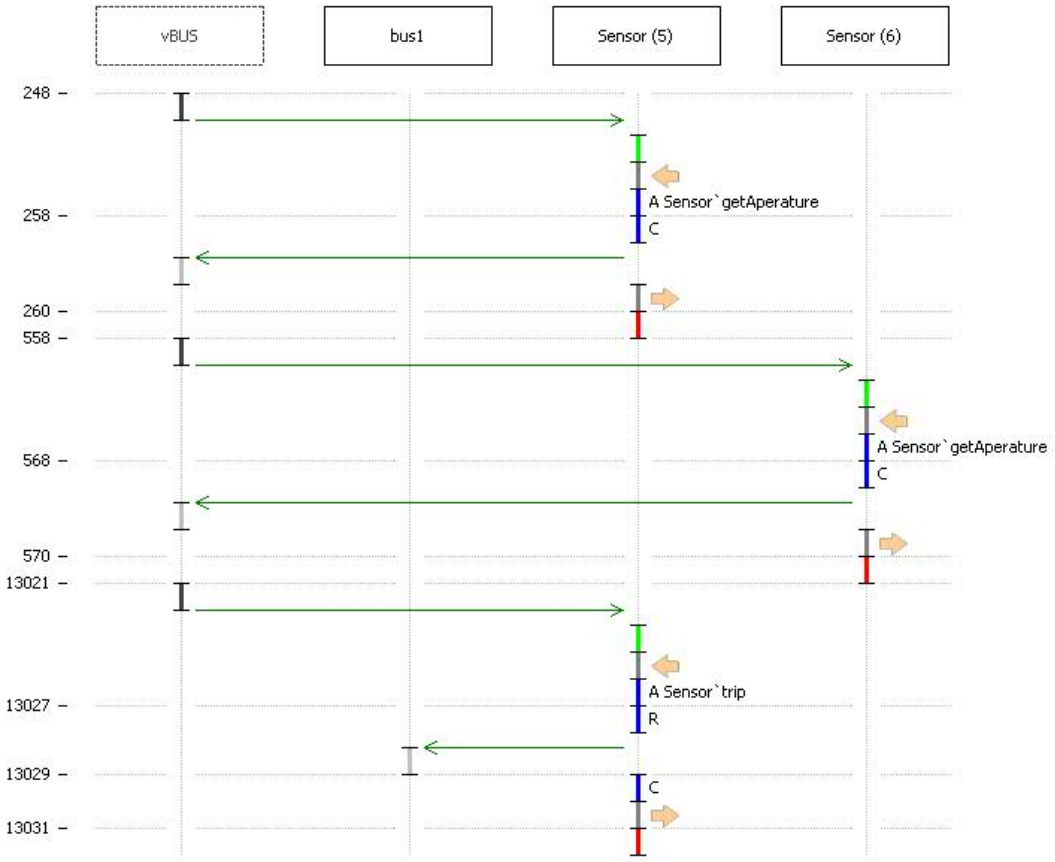

Figure 7.3: An extract from one `CPU` execution for the counter measures example

Timed Assertions

The time trace file can be thought of as simply being a sequence. It is therefore possible to specify VDM-SL predicates on such a sequence.

For this to be possible, an abstract VDM-SL representation of a trace file is needed. This is now described.

A number of types are defined to represent the different fields in a time trace file. We represent a string as a sequence of characters, and object references and thread ids as natural numbers.

```
types

  String = seq of char;
  OBJ_Ref = nat;
  ThreadId = nat;
```

A trace is an ordered sequence of trace events.

```
Trace = seq of TraceEvent
```

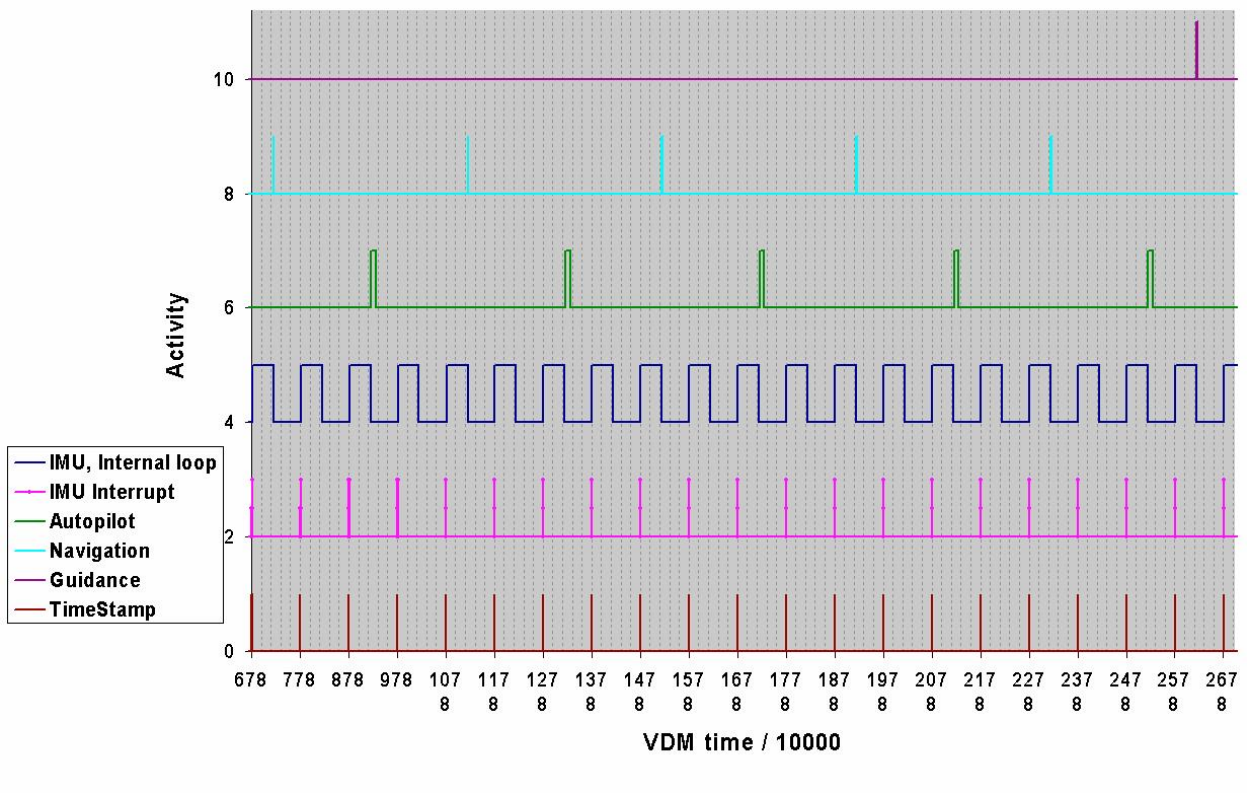

Figure 7.4: A chart in Microsoft Excel generated from a time trace file

There are many different kinds of trace events:

```
TraceEvent =
     ThreadSwapIn | ThreadSwapOut | DelayedThreadSwapIn |
     OpRequest | OpActivate | OpCompleted | ThreadCreate |
     ThreadKill |  MessageRequest | MessageActivate |
     MessageCompleted | ReplyRequest | CPUdecl | BUSdecl |
     DeployObj;
```

A thread swap in event consists of the id of the thread being swapped in, the object reference owning the thread, the class name in which the thread is defined, the cpu where it is running, the timing overhead of swapping in the thread and the time on that `CPU`.

```
  ThreadSwapIn :: id       : ThreadId
                  objref   : [OBJ_Ref]
                  clnm     : String
                  cpunm    : nat
                  overhead : nat
                  attime   : nat;
```

A delayed thread swap in has an extra field representing the delay.

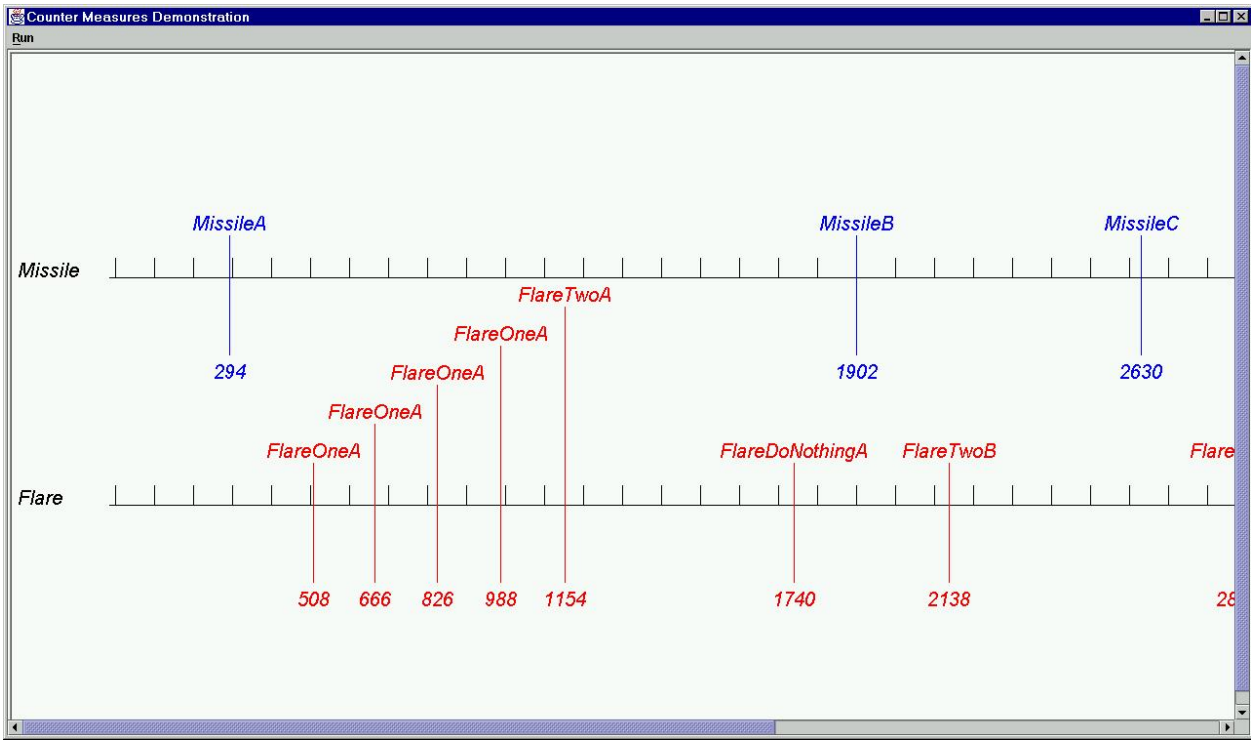

Figure 7.5: Visualization of a time trace file

```
DelayedThreadSwapIn :: id       : ThreadId
                       objref   : [OBJ_Ref]
                       clnm     : String
                       delay    : real
                       cpunm    : nat
                       overhead : nat
                       attime   : nat;
```

A thread swap out contains the same information as a thread swap in.

```
ThreadSwapOut :: id       : ThreadId
                 objref   : [OBJ_Ref]
                 clnm     : String
                 cpunm    : nat
                 overhead : nat
                 attime   : nat;
```

An operation request contains the thread in which the request was made, the name of the operation, a reference to the object on which the request occurred, the name of the class which this object is an instance of, the cpu where it is running, the arguments to the operation (if a special user option is set for storing this detailed level of information), whether it is an asynchronous operation and finally the time on that CPU.

```
OpRequest :: id     : ThreadId
             opname : String
             objref : OBJ_Ref
             clnm   : String
             cpunm  : nat
             args   : [seq of VAL]
             isasync: bool
             attime : nat;
```

Operation activations and completions contain part of the information as operation requests (the result of the operation is only present if the user have explicitly asked to get this information logged).

```
OpActivate :: id     : ThreadId
              opname : String
              objref : OBJ_Ref
              clnm   : String
              cpunm  : nat
              isasync: bool
              attime : nat;

OpCompleted :: id     : ThreadId
               opname : String
               objref : OBJ_Ref
               clnm   : String
               cpunm  : nat
               res    : [VAL]
               isasync: bool
               attime : nat;
```

Trace events are also present whenever threads are being created and killed after completion. Note that the creating of a thread also contains information whether it is a periodic thread.

```
ThreadCreate :: id     : ThreadId
                period : bool
                objref : [OBJ_Ref]
                clnm   : [String]
                cpunm  : nat
                attime : nat;

ThreadKill :: id    : ThreadId
              cpunm : nat
              attime: nat;
```

A number of events appear whenever messages are sent between different CPU's. It follow the same request, activate and complete scheme as for the operations. However, for synchronous operations there is a special reply request message event that is used to direct the result of executing an operation on a different CPU to the right CPU and the right thread there as well. Otherwise the fields used below should be rather self-explanatory.

```
MessageRequest ::
  busid   : nat
  fromcpu : nat
  tocpu   : nat
  msgid   : nat
  callthr : ThreadId
  opname  : String
  objref  : [OBJ_Ref]
  size    : nat
  attime  : nat;

ReplyRequest ::
  busid     : nat
  fromcpu   : nat
  tocpu     : nat
  msgid     : nat
  origmsgid : nat
  callthr   : ThreadId
  calleethr : ThreadId
  size      : nat
  attime    : nat;

MessageActivate ::
  msgid : nat
  attime: nat;

MessageCompleted ::
  msgid : nat
  attime: nat;
```

Declaration of CPU's and BUS'es are also logged in the trace file in order to be able to draw the overall system architecture.

```
CPUdecl ::
  id   : nat
  name : String
  expl : bool;

BUSdecl ::
  id   : nat
  topo : set of nat
  name : String;

DeployObj ::
  objref : OBJ_Ref
  cpunm  : nat
  attime : nat
```

This concludes the definition of types needed for representation of trace files.

To illustrate timed assertions, we show a couple of examples. A simple assertion could be that

any thread which is delayed, has delay within some desired maximum. This is expressed by the function `MaximumDelay`.

```
functions

MaximumDelay: real * Trace -> bool
MaximumDelay(maxDelay, trace) ==
  forall ti in set elems trace &
     is_DelayedThreadSwapIn(ti) => ti.delay <= maxDelay;
```

A slightly more complicated assertion relates to the time taken for an operation to be execute. We can specify that whenever a particular operation is activated, it completes execution within some specified period using the function `MaximumOpExecutionTime`.

```
MaximumOpExecutionTime: String * real * Trace -> bool
MaximumOpExecutionTime(opname, maxExecTime, trace) ==
  forall i in set inds trace &
     is_OpActivate(trace(i)) =>
        trace(i).opname = opname =>
           let opcompleteIndex = NextOpComplete(opname, i,
                                                trace) in
             trace(opcompleteIndex).attime - trace(i).attime <=
             maxExecTime;
```

`MaximumOpExecutionTime` uses the auxiliary function `NextOpComplete`. This finds the index of the operation completion corresponding to the operation activation that occurred at index `i`.

```
NextOpComplete: String * nat * Trace -> nat
NextOpComplete(opname, i, trace) ==
  hd [ j | j in set inds trace
         & j > i and
           is_OpCompleted(trace(j)) and
           trace(j).opname = opname]
pre exists j in set inds trace &
       is_OpCompleted(trace(j)) and
       j > i and trace(j).opname = opname
```

It is also possible to imagine that this kind of timed assertion analysis at some stage can be incorporated in generic tools such as RealTime Log Viewer or possibly directly inside Overture or VDMTools in the future.

7.3 Calibration

Analysis of the time trace files involves interpretation of the time at which events occur. According to the approach described in this document, these times are a simulation of the way in which time would progress on the target processors using the target real-time kernel. The way in which the target machine influences simulated time is by the use of the default times, which correspond to

the time taken to execute assembly instructions on the target processors.

However, this is an approximation, since the VDM interpreter has its own instruction set, which will not coincide with that of any processor. Therefore there will inevitably be an element of adjustment of the default times, to improve the precision of the simulation. This adjustment is referred to as *calibration*.

Normally calibration occurs by comparing time traces obtained by execution of the actual application on the target, with those obtained by executing the VDM model. Since the VDM model is deterministic, it can be rerun with one scenario but different sets of default times, to allow convergence to the actual application. Note that by the very nature of the approach, the simulation will never precisely match the timing behaviour of the actual application; the property desired is that the application exhibits no timing bottlenecks that were not identified by the VDM model.

Chapter 8

Postscript

In this document we have described how reactive real-time and distributed systems can be developed using Overture. We have focused on how key features of reactive real-time systems can be modelled and analyzed using Overture. The main message is that this is a viable alternative to the conventional development approaches. More time is spent in the early phases, but more confidence is gained in the different designs' ability to meet the necessary timing requirements earlier than conventionally. The main question is whether the investment in the early phases is worthwhile. We feel that it is justified, in particular in situations where access to the final hardware platform is limited or not yet determined at all. However, we do not wish to claim that the approach we have presented here is a magic recipe always ensuring correct systems. We simply wish to state that the approach which has been described is more rigorous than the conventional development approach and we feel that it is a pragmatic step in the right direction.

References

[Audsley&93] Audsley, N. and Burns, A. and Richardson, M. and Tindall, K. and Wellings, A.J. Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[Back&98] Ralph-Johan Back and Joakim Wright. *Refinement Calculus: A Systematic Introduction*. Springer, 1998.

[Ben-Ari82] M. Ben-Ari. *Priciples of Concurrent Programming*. Prentice-Hall International, 1982. 172 pages.

[Boehm88] B. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21(5):61–72, 1988.

[Booch&97] Grady Booch and Ivar Jacobson and Jim Rumbaugh. *The Unified Modelling Language, version 1.1*. Technical Report, Rational Software Corporation, September 1997. Available at: http://www.rational.com/.

[Broenink&10] Broenink, J. F. and Larsen, P. G. and Verhoef, M. and Kleijn, C. and Jovanovic, D. and Pierce, K. and Wouters F. Design Support and Tooling for Dependable Embedded Control Software. In *Proceedings of Serene 2010 International Workshop on Software Engineering for Resilient Systems*, pages 77–82, ACM, April 2010.

[Burns95] Burns, Alan. Preemptive Priority-Based Scheduling: An Appropriate Engineering Approach. In S.H. Son, editor, *Advances in Real-Time Systems*, pages 225–248, Prentice-Hall, 1995.

[CashPoint] The VDM Tool Group. *A "Cash-point" Service Example*. Technical Report, CSK Systems, June 2000.

[Chandy&88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design*. Addison-Wesley, 1988. 516 pages.

[Craigen&93] Dan Craigen and Susan Gerhart and Ted Ralston. *An International Survey of Industrial Applications of Formal Methods*. Volume Volume 2 Case Studies, U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD 20899, USA, March 1993. 188 pages.

[Dawes91] John Dawes. *The VDM-SL Reference Guide*. Pitman, 1991. 217 pages. ISBN 0-273-03151-1.

This is the best reference manual for the complete VDM-SL which is being standardised. It refers to a draft version of the standard which is quite close to the current version of the standard.

[Douglass99] Bruce Powel Douglass. *Doing Hard Time – Developing Real-Time Systems with UML Objects, Frameworks, and Patterns*. Addison-Wesley, 1999. 766 pages. ISBN 0-201-49837-5.

[Fitzgerald&05] John Fitzgerald and Peter Gorm Larsen and Paul Mukherjee and Nico Plat and Marcel Verhoef. *Validated Designs for Object–oriented Systems*. Springer, New York, 2005.

[Fitzgerald&08] J. S. Fitzgerald and P. G. Larsen and M. Verhoef. Vienna Development Method. *Wiley Encyclopedia of Computer Science and Engineering*, 2008. 11 pages. edited by Benjamin Wah, John Wiley & Sons, Inc..

[Fitzgerald&09] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, Second edition, 2009. ISBN 0-521-62348-0.

[Fitzgerald&10b] John Fitzgerald and Peter Gorm Larsen and Ken Pierce and Marcel Verhoef and Sune Wolff. Collaborative Modelling and Co-simulation in the Development of Dependable Embedded Systems. In D. Méry and S. Merz, editors, *IFM 2010, Integrated Formal Methods*, pages 12–26, Springer-Verlag, October 2010.

[Fitzgerald&13b] John Fitzgerald and Peter Gorm Larsen and Ken Pierce and Marcel Verhoef. A Formal Approach to Collaborative Modelling and Co-simulation for Embedded Systems. *Mathematical Structures in Computer Science*, August 2013.

[Fitzgerald&98a] J.S. Fitzgerald and C.B. Jones. Proof in the Validation of a Formal Model of a Tracking System for a Nuclear Plant. In J.C. Bicarregui, editor, *Proof in VDM: Case Studies*, Springer-Verlag, 1998.

[Fitzgerald&98b] John Fitzgerald and Peter Gorm Larsen. *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.

[Gosling&00] James Gosling, Bill Joy, Guy Steele and Gilad Bracha. *The Java Language Specification, Second Edition*. *The Java Series*, Addison Wesley, 2000.

[Hinchey&95] Michael G. Hinchey and Jonathan P. Bowen, editors. *Applications of Formal Methods*. Prentice Hall, 1995. ISBN 0-13-366949-1.

[Hoare85] Tony Hoare. *Communication Sequential Processes*. Prentice-Hall International, Englewood Cliffs, New Jersey 07632, 1985. 256 pages.

This is a masterpiece which can be used as reference manual on parallel programming.

[JavaSwing] Java Swing. December 2000. http://java.sun.com/products/jfc/ tsc/index.html.

[Jones90] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, Englewood Cliffs, New Jersey, second edition, 1990. 333 pages. ISBN 0-13-880733-7.

This book deals with the Vienna Development Method. The approach explains formal (functional) specifications and verified design with an emphasis on the study of proofs in the development process.

[Kruchten00] Philippe Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley, 2000. 320 pages.

[Lamport91] Leslie Lamport. *The Temporal Logic of Actions*. Technical Report 79, DEC. System Research Center, December 1991. 73 pages.

[LangManPP] The VDM Tool Group. *The IFAD VDM++ Language*. Technical Report, CSK Systems, January 2008.
ftp://ftp.ifad.dk /pub/vdmtools/doc/langmanpp_letter.pdf.

[Larsen&10] Peter Gorm Larsen and Kenneth Lausdahl and Nick Battle. *The VDM-10 Language Manual*. Technical Report TR-2010-06, The Overture Open Source Initiative, April 2010.

[Lea99] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, second edition, 1999. ISBN 0-201-31009-0.

[MDA] OMG. Model Driven Architecture. 2005.

[Meyer88] Bertrand Meyer. *Object-oriented Software Construction*. Prentice-Hall International, 1988. 534 pages.

This book is based on the object-oriented programming language Eiffel which includes interesting features. It is possible to write pre- and post-conditions and invariants in a way similar to in Meta-IV.

[Milner89] Robin Milner. *Communication and Concurrency*. Prentice-Hall International, 1989.

[Morgan90] Carroll Morgan. *Programming from Specifications*. Prentice-Hall, London, UK, 1990. 255 pages.

[Qt] Qt. December 2000. http://www.trolltech.com/.

[Rose&00] *Rational Rose 2000 Using Rose*. Rational Software Corporation, `http://www.rational.com/rose`.

[Royce70] W. Royce. Managing the development of large software systems. In *WESCON, August 1970. Reprinted in the Proceedings of the 9th International Conference on Software Engineering (ICSE), Washington D.C., IEEE Computer Society Press*, 1987.

[Rumbaugh&91] James Rumbaugh and Michael Blaha and William Premerlani and Frederick Eddy and William Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall International, 1991. ISBN 0-13-630054-5.

[Sommerville82] I. Sommerville. *Software Engineering*. Addison-Wesley, London, 1982.

[UML20] OMG. Unified Modeling Language: Superstructure. *http://www.uml.org*, August 2005. pages. .

[VDM++MethodGuide] The VDM Tool Group. *VDM++ Method Guidelines*. Technical Report, CSK Systems, January 2008.

[VisualBasic] Microsoft Visual Basic. December 2000. http://msdn.microsoft. com/vbasic/.

[Wall&92] Larry Wall and Randal L. Schwartz. *Programming Perl*. O'Reilly and Associates, Inc, 1992. 465 pages.

[Woodcock&09] Woodcock, Jim and Larsen, Peter Gorm and Bicarregui, Juan and Fitzgerald, John. Formal Methods: Practice and Experience. *ACM Computing Surveys*, 41(4):1–36, October 2009.

[Woodcock&96] Jim Woodcock and Jim Davies. *Using Z – Specification, Refinement, and Proof*. Prentice Hall International Series in Computer Science, 1996. 385 pages.

Appendix A

Glossary

Blocked The state of a thread which is unable to proceed because it is waiting for a permission predicate to become true.

Bottleneck Part of the system whose timing behaviour critically affects the overall performance of the system.

Concurrent Real-Time Distributed VDM-RT Design Model A model which defines a particular dynamic and physical architecture including its real-time behaviour and deployment to `CPU`'s.

Concurrent VDM++ Design Model A model which defines a particular dynamic architecture, without worrying in the first instance about real-time behaviour.

Default Cycles Information A mapping recording the execution times of assembly instructions on the target processor.

Default Thread The thread which initiated execution of the model. Either started by the user in the Overture or VDMTools interpreter, or started by a command in a script.

Dynamic Architecture Mapping of computations to processes (threads).

Hard Deadline A point in time by which the system must have performed some action; failure to meet such a deadline is unacceptable.

Jitter The property that a periodic event is not perfectly periodic, but occurs within some interval of its expected occurrence.

Schedulable For a thread, this means that the thread has been started, is not currently being executed but is not blocked. For a system, this means that it is possible for the system to be executed without any thread missing its deadline.

Sequential VDM++ Design Model This must describe both the data that is to be computed, and how it is to be structured into static classes, without making any commitment to a specific dynamic architecture.

Soft Deadline A point in time by which the system must have performed some action; occasional failure to meet such a deadline is acceptable, but persistent failure could lead to degraded system performance.

Static Architecture Arrangement of system behaviour into objects.

Time Trace File File generated during execution of a real-time model by Overture or VDMTools, containing information about the models run-time behaviour.

Trace Events The occurrence of a thread being swapped in or out, or an operation request, activation or completion.

Use Case This is a possible use of a system.

VDM-SL system specification This is a precise abstract design independent description of a system.

Appendix B

Design Patterns

In this chapter a few design patterns are presented. These are abstract techniques which have been found to be useful during development of a number of real-time applications.

B.1 The Fresh Data Pattern

The Fresh Data Pattern is a pattern for synchronizing data access. It is used in models where a hardware device is being modelled but the data values generated by the device are specified by another thread. The fresh data pattern then mediates communication between three threads:

The Inhabitor – a thread which inhabits the model with test data corresponding to the data which would normally be generated by the hardware device.

The Proxy – a thread which is a model of the hardware device. It provides interfaces and actions corresponding to those provided by the actual hardware device.

The Consumer – a thread which consumes data generated by the hardware device. The relationship between the inhabitor and proxy is invisible to the consumer.

A diagram showing the relationship between the main classes in the pattern is shown in Figure B.1.

The sequence diagram shown in Figure B.2 illustrates an excerpt from the pattern. The pattern is based on periodic execution of `Inhabitor`Inhabit`. In the sequence diagram it is shown that in the previous period there is a call to `Proxy`WaitFreshData`. This call is blocked. The next period begins with a call to `Inhabitor`Inhabit`. This in turn calls its own operation `GetData` to acquire some data, and then calls `PutData` to send this data to the `Proxy`. The call to `PutData` releases the blocked call to `WaitFreshData`.

The `Proxy` can then inform the `Consumer` that data is available, which the `Consumer` can access at its own leisure.

The `Data` class is used to represent the data values used in the pattern. Thus it is just a place holder and therefore has no contents in this case.

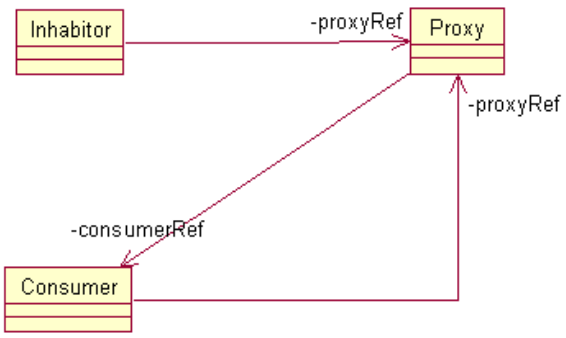

Figure B.1: Class Diagram for Fresh Data Pattern

```
class Data
end Data
```

B.1.1 The Inhabitor

The `Inhabitor` class is used to inhabit the model with data for testing purposes. It places this data in a `Proxy`, which it has a reference to. This reference is stored as an instance variable.

```
class Inhabitor

instance variables
  proxyRef : Proxy
```

The constructor is used to initialize the reference to the `Proxy`.

```
operations
  public Inhabitor : Proxy ==> Inhabitor
  Inhabitor (proxy) ==
    proxyRef := proxy;
```

Data is sent to the proxy using the `Inhabit` operation.

```
Inhabit : () ==> ()
Inhabit() ==
  proxyRef.PutData(GetData());
```

The operation `GetData` is used to actually acquire data values. Here it is left unspecified; in a model it might read data from a file.

```
GetData : () ==> Data
GetData() ==
  is not yet specified
```

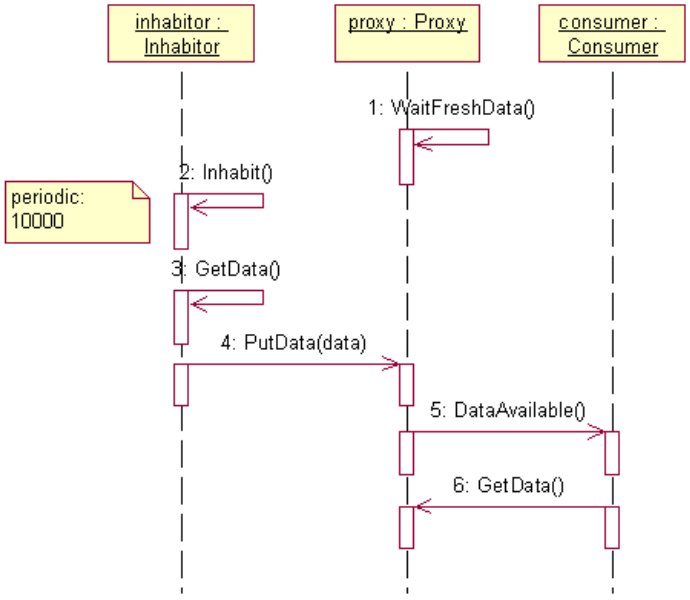

Figure B.2: Sequence Diagram for Fresh Data Pattern

The `Inhabitor` thread calls the `Inhabit` operation with some fixed period, here arbitrarily chosen to be 10000 with a jitter allowed to 100 time units and the minimum delay being 9900 time units.

```
thread
  periodic (10000,100,9900,0)(Inhabit)

end Inhabitor
```

B.1.2 The Proxy

The `Proxy` class provides the same interface as that intended for the actual device it is modelling, but it takes the data that the device would generate from the `Inhabitor`.

It has three instance variables:

`d` the data value most recently generated, or `nil` if no fresh data exists.

`freshData` a Boolean value indicating whether fresh data exists.

`consumerRef` a reference to the `Consumer`.

```
class Proxy

instance variables
  d : [Data] := nil;
  freshData : bool := false;
  consumerRef : Consumer
```

The constructor is used to initialize the `Consumer` reference.

```
operations

public Proxy : Consumer ==> Proxy
Proxy(consumer) ==
  consumerRef := consumer;
```

`PutData` is used by the `Inhabitor` to send data to the `Proxy`.

```
public PutData : Data ==> ()
PutData(newData) ==
  ( d := newData;
    freshData := true
  );
```

`GetData` is used by the `Consumer` to retrieve fresh data from the `Proxy`.

```
public GetData : () ==> Data
GetData() ==
  let od = d in
  ( d := nil;
    return od
  );
```

`WaitFreshData` is used by this class's thread to wait until fresh data is available.

```
WaitFreshData : () ==> ()
WaitFreshData() ==
  freshData := false;

sync
  per WaitFreshData => freshData
```

The thread for this class repeatedly waits for fresh data and then informs the consumer of its arrival.

```
thread

while true do
( WaitFreshData();
  consumerRef.DataAvailable()
)

end Proxy
```

B.1.3 The Consumer

The consumer represents the interface to the remainder of the system. It takes data values from the `Proxy` and uses them as it sees fit.

Three instance variables are defined:

`dataAvailable` a Boolean value indicating whether fresh data is available or not.

`proxyRef` a reference to the `Proxy`, used to retrieve data.

`d` the data value retrieved from the `Proxy`.

```
class Consumer

instance variables
  dataAvailable : bool := false;
  proxyRef : Proxy;
  d : Data
```

The constructor is used to initialize the reference to the `Proxy`.

```
operations
  public Consumer : Proxy ==> Consumer
  Consumer(proxy) ==
    proxyRef := proxy;
```

`DataAvailable` is used by the `Proxy` to indicate arrival of fresh data.

```
  public DataAvailable : () ==> ()
  DataAvailable() ==
    dataAvailable := true;
```

`GetData` is used to acquire fresh data from the proxy. It blocks whenever fresh data is not available.

```
  GetData : () ==> ()
  GetData() ==
    (d := proxyRef.GetData();
     dataAvailable := false
    );

sync
  per GetData => dataAvailable
```

The thread for this class repeatedly takes data when it is available.

```
thread

  while true do
    GetData()

end Consumer
```

B.2 The Time Stamp Pattern

The Time Stamp pattern is for use in synchronous systems, where each different thread has its own execution period. Such threads we refer to as clients. A single instance of a `TimeStamp` class will be shared amongst all of the different clients, and is used to ensure each client is awoken for its execution period. An example arrangement is shown in Figure 3.8 with two client classes included for illustrative purposes.

Each client executes its `ComputationPhase` – that computation it is intended to perform periodically. When it completes its `ComputationPhase` it calls `TimeStamp`'`WaitRelative` with its execution period, and is then awoken at that time, or as soon as possible thereafter. Note that since the clients are incidental to the pattern, they are not given in the following specification.

B.2.1 The TimeStamp Class

The `TimeStamp` class maintains a map from thread ids to time (`wakeUpMap`), representing when a particular thread should be woken. The other instance variable in the class represents the current time. Note that this concept is orthogonal to the notion of simulated time described earlier in this document.

```
class TimeStamp

values

public stepLength : nat = 1;

instance variables

currentTime  : nat   := 0;
wakeUpMap    : map nat to [nat] := {|->};
barrierCount : nat := 0;
registeredThreads : set of BaseThread := {};
isInitialising : bool := true;
-- singleton instance of class
private static timeStamp : TimeStamp := new TimeStamp();
```

Other classes can access the `TimeStamp` singleton class using the `GetInstance` operation.

```
operations

-- private constructor (singleton pattern)
private TimeStamp : () ==> TimeStamp
TimeStamp() ==
        skip;

-- public operation to get the singleton instance
public static GetInstance: () ==> TimeStamp
GetInstance() ==
  return timeStamp;
```

Whenever a thread is started it gets registered with a reference to it:

```
public RegisterThread : BaseThread ==> ()
RegisterThread(t) ==
 (barrierCount := barrierCount + 1;
  registeredThreads := registeredThreads union {t};
 );

public UnRegisterThread : BaseThread ==> ()
UnRegisterThread(t) ==
 (barrierCount := barrierCount - 1;
  registeredThreads := registeredThreads \ {t};
 );
```

When all threads have registered themselves and initialisation is finished all these threads can be started.

```
public IsInitialising: () ==> bool
IsInitialising() ==
  return isInitialising;

public DoneInitialising: () ==> ()
DoneInitialising() ==
 (if isInitialising
  then (isInitialising := false;
        for all t in set registeredThreads
        do
          start(t);
       );
 );
```

A client may request an absolute wait. using `WaitRelative`.

```
public WaitRelative : nat ==> ()
WaitRelative(val) ==
  WaitAbsolute(currentTime + val);
```

Absolute waits are performed using `WaitAbsolute`. Note that if time given is less than the current time, then the client will never be woken.

```
public WaitAbsolute : nat ==> ()
WaitAbsolute(val) ==
 (AddToWakeUpMap(threadid, val);
  -- Last to enter the barrier notifies the rest.
  BarrierReached();
  -- Wait till time is up
  Awake();
);
```

`AddToWakeUpMap` is used to add new waits to the `wakeUpMap`.

```
AddToWakeUpMap : nat * nat ==> ()
```

```
AddToWakeUpMap(tId, val) ==
  wakeUpMap := wakeUpMap ++ { tId |-> val };
```

The operation `BarrierReached` evaluates the `wakeUpMap` when all period threads have entered the mapping. Time is incremented and all threads that needs to be awoken is removed from the `wakeUpMap`.

```
BarrierReached : () ==> ()
BarrierReached() ==
  while  (card dom wakeUpMap = barrierCount) do
   (currentTime := currentTime + stepLength;
    let threadSet : set of nat = {th | th in set dom wakeUpMap
                                     & wakeUpMap(th) <> nil and
                                       wakeUpMap(th) <= currentTime }
    in
      for all t in set threadSet
      do
         wakeUpMap := {t} <-: wakeUpMap;
   )
post forall x in set rng wakeUpMap & x = nil or x >= currentTime;
```

All threads block on the operation `Awake` until they are removed from the `wakeUpMap` as described above.

```
operations

Awake: () ==> ()
Awake() == skip;

sync
  per Awake => threadid not in set dom wakeUpMap;
```

The current time of the class may be obtained via the `GetTime` operation.

```
public GetTime : () ==> nat
GetTime() ==
  return currentTime;
```

Since `barrierCount`, `registeredThreads` and `wakeUpMap` is manipulated by a number of different operations, we need to set access to them to be mutually exclusive.

```
sync
  per Awake => threadid not in set dom wakeUpMap;
  mutex (IsInitialising);
  mutex (DoneInitialising);
  mutex (AddToWakeUpMap);
  mutex (NotifyThread);
  mutex (BarrierReached);
  mutex (AddToWakeUpMap, NotifyThread);
  mutex (AddToWakeUpMap, BarrierReached);
  mutex (NotifyThread, BarrierReached);
```

```
  mutex (AddToWakeUpMap, NotifyThread, BarrierReached);

end TimeStamp
```

Appendix C

Examples In Full

C.1 VDM-SL Model for Counter Measures System

```
types

MissileInputs = seq of MissileInput;

MissileInput = MissileType * Angle;

MissileType = <MissileA> | <MissileB> | <MissileC> | <None>;

Angle = nat
inv num == num <= 360;

Output = map MagId to seq of OutputStep;

MagId = token;

OutputStep = FlareType * AbsTime;

Response = FlareType * nat;

AbsTime = nat;

FlareType = <FlareOneA>  | <FlareTwoA>  | <FlareOneB> |
            <FlareTwoB>  | <FlareOneC>  | <FlareTwoC> |
            <DoNothingA> | <DoNothingB> | <DoNothingC>;

Plan = seq of (FlareType * Delay);

Delay = nat;

values

responseDB : map MissileType to Plan =
```

```
{<MissileA> |-> [mk_(<FlareOneA>,900), mk_(<FlareTwoA>,500),
                 mk_(<DoNothingA>,100), mk_(<FlareOneA>,500)],
 <MissileB> |-> [mk_(<FlareTwoB>,500), mk_(<FlareTwoB>,700)],
 <MissileC> |-> [mk_(<FlareOneC>,400), mk_(<DoNothingC>,100),
                 mk_(<FlareTwoC>,400), mk_(<FlareOneC>,500)]
};

missilePriority : map MissileType to nat
                = {<MissileA> |-> 1,
                   <MissileB> |-> 2,
                   <MissileC> |-> 3,
                   <None>     |-> 0};

stepLength : nat = 100;

testval1 : MissileInputs = [mk_(<MissileA>,88),
                            mk_(<MissileB>,70),
                            mk_(<MissileA>,222),
                            mk_(<MissileC>,44)];

testval2 : MissileInputs = [mk_(<MissileC>,188),
                            mk_(<MissileB>,70),
                            mk_(<MissileA>,2),
                            mk_(<MissileC>,44)];

testval3 : MissileInputs = [mk_(<MissileA>,288),
                            mk_(<MissileB>,170),
                            mk_(<MissileA>,222),
                            mk_(<MissileC>,44)];

functions

CounterMeasures: MissileInputs -> Output
CounterMeasures(missileInputs) ==
  CM(missileInputs,{|->},{|->},0);

CM: MissileInputs * Output * map MagId to [MissileType] *
    nat -> Output
CM( missileInputs, outputSoFar, lastMissile, curTime) ==
  if missileInputs = []
  then outputSoFar
  else let mk_(curMis,angle) = hd missileInputs,
           magid = Angle2MagId(angle)
       in
         if magid not in set dom lastMissile or
            (magid in set dom lastMissile and
             missilePriority(curMis) >
             missilePriority(lastMissile(magid)))
         then let newOutput =
                     InterruptPlan(curTime,outputSoFar,
                                   responseDB(curMis),
```

```
                                  magid)
              in CM(tl missileInputs, newOutput,
                    lastMissile ++ {magid |-> curMis},
                    curTime + stepLength)
         else CM(tl missileInputs, outputSoFar,
                 lastMissile,curTime + stepLength)
measure CMLen;

CMLen: MissileInputs * Output * map MagId to [MissileType] * nat -> nat
CMLen(list,-,-,-) == len list;

InterruptPlan: nat * Output * Plan * MagId -> Output
InterruptPlan(curTime,expOutput,plan,magid) ==
  {magid |-> (if magid in set dom expOutput
              then LeavePrefixUnchanged(expOutput(magid),
                                        curTime)
              else []) ^
              MakeOutputFromPlan(curTime, plan)}
  munion
  ({magid} <-: expOutput);

LeavePrefixUnchanged: seq of OutputStep * nat ->
                      seq of OutputStep
LeavePrefixUnchanged(output_l, curTime) ==
  [output_l(i) | i in set inds output_l
               & let mk_(-,t) = output_l(i) in t <= curTime];

MakeOutputFromPlan : nat * seq of Response -> seq of OutputStep
MakeOutputFromPlan(curTime, response) ==
  let output = OutputAtTimeZero(response) in
    [let mk_(flare,t) = output(i)
     in
       mk_(flare,t+curTime)
    | i in set inds output];

OutputAtTimeZero : seq of Response -> seq of OutputStep
OutputAtTimeZero(response) ==
  let absTimes = RelativeToAbsoluteTimes(response) in
    let mk_(firstFlare,-) = hd absTimes in
      [mk_(firstFlare,0)] ^
      [ let mk_(-,t) = absTimes(i-1),
            mk_(f,-) = absTimes(i) in
          mk_(f,t) | i in set {2,...,len absTimes}];

RelativeToAbsoluteTimes : seq of Response ->
                          seq of (FlareType * nat)
RelativeToAbsoluteTimes(ts) ==
  if ts = []
  then []
  else let mk_(f,t) = hd ts,
           ns = RelativeToAbsoluteTimes(tl ts) in
```

```
        [mk_(f,t)] ^ [ let mk_(nf, nt) = ns(i)
                       in mk_(nf, nt + t)
                     | i in set inds ns]
measure RespLen;

RespLen: seq of Response -> nat
RespLen(l) ==
  len l;

Angle2MagId: Angle -> MagId
Angle2MagId(angle) ==
  if angle < 90
  then mk_token("Magazine 1")
  elseif angle < 180
  then mk_token("Magazine 2")
  elseif angle < 270
  then mk_token("Magazine 3")
  else mk_token("Magazine 4");
```

C.2 Sequential VDM++ Model for Counter Measures System

C.2.1 The CM Class

```
class CM

instance variables

-- maintain a link to the detector
public static detector : MissileDetector := new MissileDetector();

public static sensor0 : Sensor := new Sensor(detector,0);
public static sensor1 : Sensor := new Sensor(detector,90);
public static sensor2 : Sensor := new Sensor(detector,180);
public static sensor3 : Sensor := new Sensor(detector,270);

public static controller0 : FlareController := new FlareController(0);
public static controller1 : FlareController := new FlareController(120);
public static controller2 : FlareController := new FlareController(240);

public static dispenser0 : FlareDispenser := new FlareDispenser(0);
public static dispenser1 : FlareDispenser := new FlareDispenser(30);
public static dispenser2 : FlareDispenser := new FlareDispenser(60);
public static dispenser3 : FlareDispenser := new FlareDispenser(90);

public static dispenser4 : FlareDispenser := new FlareDispenser(0);
public static dispenser5 : FlareDispenser := new FlareDispenser(30);
public static dispenser6 : FlareDispenser := new FlareDispenser(60);
public static dispenser7 : FlareDispenser := new FlareDispenser(90);
```

```
public static dispenser8 : FlareDispenser := new FlareDispenser(0);
public static dispenser9 : FlareDispenser := new FlareDispenser(30);
public static dispenser10 : FlareDispenser := new FlareDispenser(60);
public static dispenser11 : FlareDispenser := new FlareDispenser(90);

end CM
```

C.2.2 The World Class

```
class World

instance variables

-- maintain a link to the environment
public static env : [Environment] := nil;
public static timerRef : Timer := Timer`GetInstance();

operations

public World: () ==> World
World () ==
  (-- set-up the sensors
   env := new Environment("scenario.txt");
   env.addSensor(CM`sensor0);
   env.addSensor(CM`sensor1);
   env.addSensor(CM`sensor2);
   env.addSensor(CM`sensor3);

   -- add the first controller with four dispensers
   CM`controller0.addDispenser(CM`dispenser0);
   CM`controller0.addDispenser(CM`dispenser1);
   CM`controller0.addDispenser(CM`dispenser2);
   CM`controller0.addDispenser(CM`dispenser3);
   CM`detector.addController(CM`controller0);

   -- add the second controller with four dispensers
   CM`controller1.addDispenser(CM`dispenser4);
   CM`controller1.addDispenser(CM`dispenser5);
   CM`controller1.addDispenser(CM`dispenser6);
   CM`controller1.addDispenser(CM`dispenser7);
   CM`detector.addController(CM`controller1);

   -- add the third controller with four dispensers
   CM`controller2.addDispenser(CM`dispenser8);
   CM`controller2.addDispenser(CM`dispenser9);
   CM`controller2.addDispenser(CM`dispenser10);
   CM`controller2.addDispenser(CM`dispenser11);
   CM`detector.addController(CM`controller2);
```

```
  );

-- the run function blocks the user-interface thread
-- until all missiles in the file have been processed
public Run: () ==> ()
Run () ==
  env.Run()

end World
```

C.2.3 The Global Class

```
class GLOBAL

values

public SENSOR_APERTURE = 90;
public FLARE_APERTURE = 120;
public DISPENSER_APERTURE = 30

types

-- there are three different types of missiles
public
MissileType = <MissileA> | <MissileB> | <MissileC> | <None>;

-- there are nine different flare types, three per missile
public FlareType =
    <FlareOneA> | <FlareTwoA> | <DoNothingA> |
    <FlareOneB> | <FlareTwoB> | <DoNothingB> |
    <FlareOneC> | <FlareTwoC> | <DoNothingC>;

-- the angle at which the missile is incoming
public Angle = nat
inv num == num <= 360;

public EventId = nat;

public Time = nat

operations

public canObserve: Angle * Angle * Angle ==> bool
canObserve (pangle, pleft, psize) ==
  def pright = (pleft + psize) mod 360 in
    if pright < pleft
    -- check between [0,pright> and [pleft,360>
    then return (pangle < pright or pangle >= pleft)
    -- check between [pleft, pright>
```

```
    else return (pangle >= pleft and pangle < pright);

public getAperture: () ==> Angle * Angle
getAperture () == is subclass responsibility;

end GLOBAL
```

C.2.4 The Environment Class

```
class Environment is subclass of GLOBAL

types

public inline  = EventId * MissileType * Angle * Time;
public outline = EventId * FlareType * Angle * Time * Time;

instance variables

-- access to the VDMTools stdio
io : IO := new IO();

-- the input file to process
inlines : seq of inline := [];

-- the output file to print
outlines : seq of outline := [];

-- maintain a link to all sensors
ranges : map nat to (Angle * Angle) := {|->};
sensors : map nat to Sensor := {|->};
inv dom ranges = dom sensors;

-- information about the latest event that has arrived
evid : [EventId] := nil;

busy : bool := true;

operations

public Environment: seq of char ==> Environment
Environment (fname) ==
  def mk_ (-,input) = io.freadval[seq of inline](fname) in
    inlines := input;

public addSensor: Sensor ==> ()
addSensor (psens) ==
  (dcl id : nat := card dom ranges + 1;
   atomic (
    ranges := ranges munion {id |-> psens.getAperture()};
```

```
    sensors := sensors munion {id |-> psens}
   )
  );

public Run: () ==> ()
Run () ==
 (while not (isFinished() and CM`detector.isFinished()) do
    (evid := createSignal();
     CM`detector.Step();
     World`timerRef.StepTime();
    );
 showResult()
 );

private createSignal: () ==> [EventId]
createSignal () ==
  (if len inlines > 0
   then (dcl curtime : Time := World`timerRef.GetTime(),
             done : bool := false;
         while not done do
           def mk_ (eventid, pmt, pa, pt) = hd inlines in
             if pt <= curtime
             then (for all id in set dom ranges do
                     def mk_(papplhs,pappsize) = ranges(id) in
                       if canObserve(pa,papplhs,pappsize)
                       then sensors(id).trip(eventid,pmt,pa);
                   inlines := tl inlines;
                   done := len inlines = 0;
                   return eventid )
             else (done := true;
                   return nil ))
   else (busy := false;
         return nil));

public handleEvent: EventId * FlareType * Angle * Time * Time ==> ()
handleEvent (newevid,pfltp,angle,pt1,pt2) ==
  (outlines := outlines ^ [mk_ (newevid,pfltp, angle,pt1, pt2)] );

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

public isFinished : () ==> bool
isFinished () ==
  return inlines = [] and not busy;

end Environment
```

C.2.5 The Sensor Class

```
class Sensor is subclass of GLOBAL

instance variables

-- the missile detector this sensor is connected to
private detector : MissileDetector;

-- the left hand-side of the viewing angle of the sensor
private aperture : Angle;

operations

public Sensor: MissileDetector * Angle ==> Sensor
Sensor (pmd, psa) == ( detector := pmd; aperture := psa);

-- get the left hand-side start point and opening angle
public getAperture: () ==> GLOBAL`Angle * GLOBAL`Angle
getAperture () == return mk_ (aperture, SENSOR_APERTURE);

-- trip is called asynchronously from the environment to
-- signal an event. the sensor triggers if the event is
-- in the field of view. the event is stored in the
-- missile detector for further processing
public trip: EventId * MissileType * Angle ==> ()
trip (evid, pmt, pa) ==
  -- log and time stamp the observed threat
  detector.addThreat(evid, pmt,pa,World`timerRef.GetTime())
pre canObserve(pa, aperture, SENSOR_APERTURE)

end Sensor
```

C.2.6 The Missile Detector Class

```
class MissileDetector is subclass of GLOBAL

-- the primary task of the MissileDetector is to
-- collect all sensor data and dispatch each event
-- to the appropriate FlareController

instance variables

-- maintain a link to each controller
ranges : map nat to (Angle * Angle) := {|->};
controllers : map nat to FlareController := {|->};
inv dom ranges = dom controllers;
```

```
-- collects the observations from all attached sensors
threats : seq of (EventId * MissileType * Angle * Time) := [];

-- status of the missile detector
busy : bool := false

operations

-- addController is only used to instantiate the model
public addController: FlareController ==> ()
addController (pctrl) ==
  (dcl nid : nat := card dom ranges + 1;
   atomic
    (ranges := ranges munion {nid |-> pctrl.getAperture()};
     controllers := controllers munion {nid |-> pctrl}
    );
  );

public Step: () ==> ()
Step() ==
  (if threats <> []
   then def mk_ (evid,pmt, pa, pt) = getThreat() in
          for all id in set dom ranges do
            def mk_(papplhs, pappsize) = ranges(id) in
              if canObserve(pa, papplhs, pappsize)
              then controllers(id).addThreat(evid,pmt,pa,pt);
    busy := len threats > 0;
    for all id in set dom controllers do
      controllers(id).Step()
  );

-- addThreat is a helper operation to modify the event
-- list. currently events are stored first come first served.
-- one could imagine using a different ordering instead.
public addThreat: EventId * MissileType * Angle * Time ==> ()
addThreat (evid,pmt,pa,pt) ==
  (threats := threats ^ [mk_ (evid,pmt,pa,pt)];
   busy := true );

-- getThreat is a local helper operation to modify the event list
private getThreat: () ==> EventId * MissileType * Angle * Time
getThreat () ==
  (dcl res : EventId * MissileType * Angle * Time := hd threats;
   threats := tl threats;
   return res );

public isFinished: () ==> bool
isFinished () ==
  return forall id in set dom controllers &
            controllers(id).isFinished()
```

```
end MissileDetector
```

C.2.7 The Flare Controller Class

```
class FlareController is subclass of GLOBAL

instance variables

-- the left hand-side of the working angle
private aperture : Angle;

-- maintain a link to each dispenser
ranges : map nat to (Angle * Angle) := {|->};
dispensers : map nat to FlareDispenser := {|->};
inv dom ranges = dom dispensers;

-- the relevant events to be treated by this controller
threats : seq of (EventId * MissileType * Angle * Time) := [];

-- the status of the controller
busy : bool := false

operations

public FlareController: Angle ==> FlareController
FlareController (papp) == aperture := papp;

public addDispenser: FlareDispenser ==> ()
addDispenser (pfldisp) ==
  let angle = aperture + pfldisp.GetAngle() in
    (dcl id : nat := card dom ranges + 1;
     atomic
     (ranges := ranges munion
                {id |-> mk_(angle, DISPENSER_APERTURE)};
      dispensers := dispensers munion {id |-> pfldisp});
     );

public Step: () ==> ()
Step() ==
  (if threats <> []
   then def mk_ (evid,pmt, pa, pt) = getThreat() in
          for all id in set dom ranges do
            def mk_(papplhs, pappsize) = ranges(id) in
              if canObserve(pa, papplhs, pappsize)
              then dispensers(id).addThreat(evid,pmt,pt);
   busy := len threats > 0;
   for all id in set dom dispensers do
     dispensers(id).Step());
```

```
-- get the left hand-side start point and opening angle
public getAperture: () ==> GLOBAL`Angle * GLOBAL`Angle
getAperture () == return mk_(aperture, FLARE_APERTURE);

-- addThreat is a helper operation to modify the event
-- list. currently events are stored first come first served.
-- one could imagine using a different ordering instead
public addThreat: EventId * MissileType * Angle * Time ==> ()
addThreat (evid,pmt,pa,pt) ==
  (threats := threats ^ [mk_ (evid,pmt,pa,pt)];
   busy := true );

-- getThreat is a local helper operation to modify the event list
private getThreat: () ==> EventId * MissileType * Angle * Time
getThreat () ==
  (dcl res : EventId * MissileType * Angle * Time := hd threats;
   threats := tl threats;
   return res );

public isFinished: () ==> bool
isFinished () ==
  return forall id in set dom dispensers &
            dispensers(id).isFinished();

end FlareController
```

C.2.8 The Flare Dispenser Class

```
class FlareDispenser is subclass of GLOBAL

values

responseDB : map MissileType to Plan =
  {<MissileA> |-> [mk_(<FlareOneA>,900),
                   mk_(<FlareTwoA>,500),
                   mk_(<DoNothingA>,100),
                   mk_(<FlareOneA>,500)],
   <MissileB> |-> [mk_(<FlareTwoB>,500),
                   mk_(<FlareTwoB>,700)],
   <MissileC> |-> [mk_(<FlareOneC>,400),
                   mk_(<DoNothingC>,100),
                   mk_(<FlareTwoC>,400),
                   mk_(<FlareOneC>,500)] };

missilePriority : map MissileType to nat =
  {<None>     |-> 0,
   <MissileA> |-> 1,
   <MissileB> |-> 2,
   <MissileC> |-> 3 }
```

```
types

public Plan = seq of PlanStep;

public PlanStep = FlareType * Time;

instance variables

public curplan : Plan := [];
curprio        : nat := 0;
busy           : bool := false;
aperture       : Angle;
eventid        : [EventId];

operations

public FlareDispenser: nat ==> FlareDispenser
FlareDispenser(ang) ==
  aperture := ang;

public Step: () ==> ()
Step() ==
  if len curplan > 0
  then (dcl curtime : Time := World`timerRef.GetTime(),
            first : PlanStep := hd curplan,
            next : Plan := tl curplan;
        let mk_(fltp, fltime) = first in
          (if fltime <= curtime
           then (releaseFlare(eventid,fltp,fltime,curtime);
                 curplan := next;
                 if len next = 0
                 then (curprio := 0;
                       busy := false ) )
           )
    );

public GetAngle: () ==> nat
GetAngle() ==
  return aperture;

public addThreat: EventId * MissileType * Time ==> ()
addThreat (evid, pmt, ptime) ==
  if missilePriority(pmt) > curprio
  then (dcl newplan : Plan :=  [],
            newtime : Time := ptime;
        -- construct an absolute time plan
        for mk_(fltp, fltime) in responseDB(pmt) do
          (newplan := newplan ^ [mk_ (fltp, newtime)];
           newtime := newtime + fltime );
        -- immediately release the first action
```

```
        def mk_(fltp, fltime) = hd newplan;
            t = World`timerRef.GetTime() in
          releaseFlare(evid,fltp,fltime,t);
        -- store the rest of the plan
        curplan := tl newplan;
        eventid := evid;
        curprio := missilePriority(pmt);
        busy := true )
pre pmt in set dom missilePriority and
    pmt in set dom responseDB;

private releaseFlare: EventId * FlareType * Time * Time ==> ()
releaseFlare (evid,pfltp, pt1, pt2) ==
  World`env.handleEvent(evid,pfltp,aperture,pt1,pt2);

public isFinished: () ==> bool
isFinished () ==
  return not busy

end FlareDispenser
```

C.2.9 The Timer Class

```
class Timer

instance variables

currentTime : nat := 0;
private static timerInstance : Timer := new Timer();

values

stepLength : nat = 10;

operations

private Timer: () ==> Timer
Timer() ==
  skip;

public static GetInstance: () ==> Timer
GetInstance() ==
  return timerInstance;

public StepTime : () ==> ()
StepTime() ==
  currentTime := currentTime + stepLength;

public GetTime : () ==> nat
```

```
GetTime() ==
  return currentTime;

end Timer
```

C.2.10 The IO Class

```
class IO

--      Overture STANDARD LIBRARY: INPUT/OUTPUT
--      ---------------------------------------
--
-- Standard library for the Overture Interpreter. When the interpreter
-- evaluates the preliminary functions/operations in this file,
-- corresponding internal functions is called instead of issuing a run
-- time error. Signatures should not be changed, as well as name of
-- module (VDM-SL) or class (VDM++). Pre/post conditions is
-- fully user customisable.
-- Dont care's may NOT be used in the parameter lists.
--
-- The in/out functions  will return false if an error occurs. In this
-- case an internal error string will be set (see 'ferror').

types

public
filedirective = <start>|<append>

functions

-- Write VDM value in ASCII format to std out:
public
writeval[@p]: @p -> bool
writeval(val)==
  is not yet specified;

-- Write VDM value in ASCII format to file.
-- fdir = <start> will overwrite existing file,
-- fdir = <append> will append output to the file (created if
-- not existing).
public
fwriteval[@p]:seq1 of char * @p * filedirective -> bool
fwriteval(filename,val,fdir) ==
  is not yet specified;

-- Read VDM value in ASCII format from file
public
freadval[@p]:seq1 of char -> bool * [@p]
freadval(f) ==
```

```
  is not yet specified
  post let mk_(b,t) = RESULT in not b => t = nil;

operations

-- Write text to std out. Surrounding double quotes will be stripped,
-- backslashed characters should be interpreted.
public
echo: seq of char ==> bool
echo(text) ==
  fecho ("",text,nil);

-- Write text to file like 'echo'
public
fecho: seq of char * seq of char * [filedirective] ==> bool
fecho (filename,text,fdir) ==
  is not yet specified
  pre filename = "" <=> fdir = nil;

-- The in/out functions  will return false if an error occur. In this
-- case an internal error string will be set. 'ferror' returns this
-- string and set it to "".
public
ferror:()  ==> seq of char
ferror () ==
  is not yet specified;

-- New simplified format printing operations
-- The questionmark in the signature simply means any type
public static print: ? ==> ()
print(arg) ==
  is not yet specified;

-- New simplified format printing operations
-- The questionmark in the signature simply means any type
public static printf: seq of char * seq of ? ==> ()
printf(format, args) ==
  is not yet specified;

end IO
```

C.3 Concurrent VDM++ Model for Counter Measures System

C.3.1 The CM Class

```
class CM

instance variables
```

```
  -- maintain a link to the detector
  public static detector : MissileDetector := new MissileDetector(nil);

  public static sensor0 : Sensor := new Sensor(detector,0);
  public static sensor1 : Sensor := new Sensor(detector,90);
  public static sensor2 : Sensor := new Sensor(detector,180);
  public static sensor3 : Sensor := new Sensor(detector,270);

  public static controller0 : FlareController := new FlareController(0, nil);
  public static controller1 : FlareController := new FlareController(120, nil);
  public static controller2 : FlareController := new FlareController(240, nil);

  public static dispenser0 : FlareDispenser := new FlareDispenser(0, nil);
  public static dispenser1 : FlareDispenser := new FlareDispenser(30, nil);
  public static dispenser2 : FlareDispenser := new FlareDispenser(60, nil);
  public static dispenser3 : FlareDispenser := new FlareDispenser(90, nil);

  public static dispenser4 : FlareDispenser := new FlareDispenser(0, nil);
  public static dispenser5 : FlareDispenser := new FlareDispenser(30, nil);
  public static dispenser6 : FlareDispenser := new FlareDispenser(60, nil);
  public static dispenser7 : FlareDispenser := new FlareDispenser(90, nil);

  public static dispenser8 : FlareDispenser := new FlareDispenser(0, nil);
  public static dispenser9 : FlareDispenser := new FlareDispenser(30, nil);
  public static dispenser10 : FlareDispenser := new FlareDispenser(60, nil);
  public static dispenser11 : FlareDispenser := new FlareDispenser(90, nil);

end CM
```

C.3.2 The World Class

```
class World

instance variables

public static timerRef : TimeStamp := TimeStamp`GetInstance();
public static env : [Environment] := nil;

operations

public World: () ==> World
World () ==
  (-- set-up the sensors
   env := new Environment("scenario.txt", nil);

   env.addSensor(CM`sensor0);
   env.addSensor(CM`sensor1);
   env.addSensor(CM`sensor2);
   env.addSensor(CM`sensor3);
```

```
   -- add the first controller with four dispensers
   CM`controller0.addDispenser(CM`dispenser0);
   CM`controller0.addDispenser(CM`dispenser1);
   CM`controller0.addDispenser(CM`dispenser2);
   CM`controller0.addDispenser(CM`dispenser3);
   CM`detector.addController(CM`controller0);

   -- add the second controller with four dispensers
   CM`controller1.addDispenser(CM`dispenser4);
   CM`controller1.addDispenser(CM`dispenser5);
   CM`controller1.addDispenser(CM`dispenser6);
   CM`controller1.addDispenser(CM`dispenser7);
   CM`detector.addController(CM`controller1);

   -- add the third controller with four dispensers
   CM`controller2.addDispenser(CM`dispenser8);
   CM`controller2.addDispenser(CM`dispenser9);
   CM`controller2.addDispenser(CM`dispenser10);
   CM`controller2.addDispenser(CM`dispenser11);
   CM`detector.addController(CM`controller2);
   );

-- the run function blocks the user-interface thread
-- until all missiles in the file have been processed
public Run: () ==> ()
Run () ==
  (-- start the environment
   timerRef.DoneInitialising();
   -- wait for the environment to handle all input
   env.isFinished();
   -- wait for the missile detector to finish
   CM`detector.isFinished();
   -- print the result
   env.showResult())

end World
```

C.3.3 The Global Class

```
class GLOBAL

values

public SENSOR_APERTURE = 90;
public FLARE_APERTURE = 120;
public DISPENSER_APERTURE = 30

types
```

```
-- there are three different types of missiles
public MissileType = <MissileA> | <MissileB> | <MissileC> | <None>;

-- there are nine different flare types, three per missile
public FlareType =
    <FlareOneA> | <FlareTwoA> | <DoNothingA> |
    <FlareOneB> | <FlareTwoB> | <DoNothingB> |
    <FlareOneC> | <FlareTwoC> | <DoNothingC>;

-- the angle at which the missile is incoming
public Angle = nat
inv num == num < 360;

public EventId = nat;

public Time = nat;

operations

public canObserve: Angle * Angle * Angle ==> bool
canObserve (pangle, pleft, psize) ==
  def pright = (pleft + psize) mod 360 in
    if pright < pleft
    -- check between [0,pright> and [pleft,360>
    then return (pangle < pright or pangle >= pleft)
    -- check between [pleft, pright>
    else return (pangle >= pleft and pangle < pright);

end GLOBAL
```

C.3.4 The Environment Class

```
class Environment is subclass of GLOBAL, BaseThread

types

public InputTP   = (Time * seq of inline);

public inline  = EventId * MissileType * Angle * Time;
public outline = EventId * FlareType * Angle * Time * Time

instance variables

-- access to the VDMTools stdio
io : IO := new IO();

-- the input file to process
inlines : seq of inline := [];
```

```
-- the output file to print
outlines : seq of outline := [];

-- maintain a link to all sensors
ranges : map nat to (Angle * Angle) := {|->};
sensors : map nat to Sensor := {|->};
inv dom ranges = dom sensors;

busy : bool := true;

-- Amount of time we want to simulate
simtime : Time;

operations

public Environment: seq of char * [ThreadDef] ==> Environment
Environment (fname, tDef) ==
 (def mk_ (-,mk_(timeval,input)) = io.freadval[InputTP](fname) in
    (inlines := input;
     simtime := timeval);

  if tDef <> nil
  then (period := tDef.p;
        isPeriodic := tDef.isP;
       );
  BaseThread(self);
  );

public addSensor: Sensor ==> ()
addSensor (psens) ==
  (dcl id : nat := card dom ranges + 1;
   atomic (
    ranges := ranges munion {id |-> psens.getAperture()};
    sensors := sensors munion {id |-> psens}
   )
  );

private createSignal: () ==> ()
createSignal () ==
  (if len inlines > 0
   then (dcl curtime : Time := World`timerRef.GetTime(),
             done : bool := false;
         while not done do
           def mk_ (eventid, pmt, pa, pt) = hd inlines in
             if pt <= curtime
             then (for all id in set dom ranges do
                     def mk_(papplhs,pappsize) = ranges(id) in
                       if canObserve(pa,papplhs,pappsize)
                       then sensors(id).trip(eventid,pmt,pa);
                   inlines := tl inlines;
```

```
                    done := len inlines = 0;
                    return)
              else (done := true;
                    return))
   else (busy := false;
         return));

public handleEvent: EventId * FlareType * Angle * Time * Time ==> ()
handleEvent (evid,pfltp,angle,pt1,pt2) ==
  (outlines := outlines ^ [mk_ (evid,pfltp,angle,pt1,pt2)] );

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

public isFinished : () ==> ()
isFinished () == skip;

public Step : () ==> ()
Step() ==
 (if World`timerRef.GetTime() < simtime
  then createSignal()
  else busy := false;
 );

sync

mutex (handleEvent);
mutex (createSignal);
per isFinished => not busy;

end Environment
```

C.3.5 The Sensor Class

```
class Sensor is subclass of GLOBAL

instance variables

-- the missile detector this sensor is connected to
private detector : MissileDetector;

-- the left hand-side of the viewing angle of the sensor
private aperture : Angle;

operations

public Sensor: MissileDetector * Angle ==> Sensor
Sensor (pmd, psa) == ( detector := pmd; aperture := psa);
```

```
-- get the left hand-side start point and opening angle
public getAperture: () ==> GLOBAL`Angle * GLOBAL`Angle
getAperture () == return mk_ (aperture, SENSOR_APERTURE);

-- trip is called asynchronously from the environment to
-- signal an event. the sensor triggers if the event is
-- in the field of view. the event is stored in the
-- missile detector for further processing
public trip: EventId * MissileType * Angle ==> ()
trip (evid, pmt, pa) ==
  -- log and time stamp the observed threat
  detector.addThreat(evid, pmt,pa,World`timerRef.GetTime())
pre canObserve(pa, aperture, SENSOR_APERTURE)

end Sensor
```

C.3.6 The Missile Detector Class

```
class MissileDetector is subclass of GLOBAL, BaseThread

-- the primary task of the MissileDetector is to
-- collect all sensor data and dispatch each event
-- to the appropriate FlareController

instance variables

-- maintain a link to each controller
ranges : map nat to (Angle * Angle) := {|->};
controllers : map nat to FlareController := {|->};
inv dom ranges = dom controllers;

-- collects the observations from all attached sensors
threats : seq of (EventId * MissileType * Angle * Time) := [];

-- status of the missile detector
busy : bool := false

operations

public MissileDetector: [ThreadDef] ==> MissileDetector
MissileDetector(tDef)==
 (if tDef <> nil
  then (period := tDef.p;
        isPeriodic := tDef.isP;
       );
  BaseThread(self);
 );
```

```
-- addController is only used to instantiate the model
public addController: FlareController ==> ()
addController (pctrl) ==
  (dcl nid : nat := card dom ranges + 1;
   atomic
    (ranges := ranges munion {nid |-> pctrl.getAperture()};
     controllers := controllers munion {nid |-> pctrl}
    );
   );

-- addThreat is a helper operation to modify the event
-- list. currently events are stored first come first served.
-- one could imagine using a different ordering instead.
public addThreat: EventId * MissileType * Angle * Time ==> ()
addThreat (evid,pmt,pa,pt) ==
  (threats := threats ^ [mk_ (evid,pmt,pa,pt)];
   busy := true );

-- getThreat is a local helper operation to modify the event list
private getThreat: () ==> EventId * MissileType * Angle * Time
getThreat () ==
  (dcl res : EventId * MissileType * Angle * Time := hd threats;
   threats := tl threats;
   return res );

public isFinished: () ==> ()
isFinished () ==
  for all id in set dom controllers do
    controllers(id).isFinished();

Step: () ==> ()
Step() ==
( if threats <> []
  then (def mk_ (evid,pmt, pa, pt) = getThreat() in
          for all id in set dom ranges do
            def mk_(papplhs, pappsize) = ranges(id) in
              if canObserve(pa, papplhs, pappsize)
              then controllers(id).addThreat(evid,pmt,pa,pt);
        busy := len threats > 0);
);

sync
  mutex (Step);

  -- addThreat and getThreat modify the same instance variables
  -- therefore they need to be declared mutual exclusive
  mutex (addThreat,getThreat);

  -- getThreat is used as a 'blocking read' from the main
  -- thread of control of the missile detector
  per getThreat => len threats > 0;
```

```
  per isFinished => not busy

end MissileDetector
```

C.3.7 The Flare Controller Class

```
class FlareController is subclass of GLOBAL, BaseThread

instance variables

-- the left hand-side of the working angle
private aperture : Angle;

-- maintain a link to each dispenser
ranges : map nat to (Angle * Angle) := {|->};
dispensers : map nat to FlareDispenser := {|->};
inv dom ranges = dom dispensers;

-- the relevant events to be treated by this controller
threats : seq of (EventId * MissileType * Angle * Time) := [];

-- the status of the controller
busy : bool := false

operations

public FlareController: Angle * [ThreadDef] ==> FlareController
FlareController (papp, tDef) ==
 (aperture := papp;

  if tDef <> nil
  then (period := tDef.p;
        isPeriodic := tDef.isP;
       );
  BaseThread(self);
 );

public addDispenser: FlareDispenser ==> ()
addDispenser (pfldisp) ==
  let angle = aperture + pfldisp.GetAngle() in
    (dcl id : nat := card dom ranges + 1;
     atomic
      (ranges := ranges munion
                 {id |-> mk_(angle, DISPENSER_APERTURE)};
       dispensers := dispensers munion {id |-> pfldisp}
      );
     );

-- get the left hand-side start point and opening angle
```

```
public getAperture: () ==> GLOBAL`Angle * GLOBAL`Angle
getAperture () == return mk_(aperture, FLARE_APERTURE);

-- addThreat is a helper operation to modify the event
-- list. currently events are stored first come first served.
-- one could imagine using a different ordering instead
public addThreat: EventId * MissileType * Angle * Time ==> ()
addThreat (evid,pmt,pa,pt) ==
  (threats := threats ^ [mk_ (evid,pmt,pa,pt)];
   busy := true );

-- getThreat is a local helper operation to modify the event list
private getThreat: () ==> EventId * MissileType * Angle * Time
getThreat () ==
  (dcl res : EventId * MissileType * Angle * Time := hd threats;
   threats := tl threats;
   return res );

public isFinished: () ==> ()
isFinished () ==
  for all id in set dom dispensers do
    dispensers(id).isFinished();

Step: () ==> ()
Step() ==
  (if threats <> []
  then (def mk_ (evid,pmt, pa, pt) = getThreat() in
         for all id in set dom ranges do
           def mk_(papplhs, pappsize) = ranges(id) in
             if canObserve(pa, papplhs, pappsize)
             then dispensers(id).addThreat(evid,pmt,pt);
        busy := len threats > 0 );
       );

sync

-- addThreat and getThreat modify the same instance variables
-- therefore they need to be declared mutual exclusive
mutex (addThreat,getThreat);
mutex (Step);

-- getThreat is used as a 'blocking read' from the main
-- thread of control of the missile detector
per getThreat => len threats > 0;
per isFinished => len threats = 0 --not busy

end FlareController
```

C.3.8 The Flare Dispenser Class

```
class FlareDispenser is subclass of GLOBAL, BaseThread

values

responseDB : map MissileType to Plan =
  {<MissileA> |-> [mk_(<FlareOneA>,900),
                   mk_(<FlareTwoA>,500),
                   mk_(<DoNothingA>,100),
                   mk_(<FlareOneA>,500)],
   <MissileB> |-> [mk_(<FlareTwoB>,500),
                   mk_(<FlareTwoB>,700)],
   <MissileC> |-> [mk_(<FlareOneC>,400),
                   mk_(<DoNothingC>,100),
                   mk_(<FlareTwoC>,400),
                   mk_(<FlareOneC>,500)] };

missilePriority : map MissileType to nat =
  {<None>     |-> 0,
   <MissileA> |-> 1,
   <MissileB> |-> 2,
   <MissileC> |-> 3 }

types

public Plan = seq of PlanStep;

public PlanStep = FlareType * Time;

instance variables

public curplan : Plan := [];
curprio        : nat := 0;
busy           : bool := false;
aperture       : Angle;
eventid        : [EventId];

operations

public FlareDispenser: Angle * [ThreadDef] ==> FlareDispenser
FlareDispenser(ang, tDef) ==
 (aperture := ang;

  if tDef <> nil
  then (period := tDef.p;
        isPeriodic := tDef.isP;
       );
  BaseThread(self);
 );
```

```
public GetAngle: () ==> nat
GetAngle() ==
  return aperture;

public addThreat: EventId * MissileType * Time ==> ()
addThreat (evid, pmt, ptime) ==
  if missilePriority(pmt) > curprio
  then (dcl newplan : Plan :=  [],
            newtime : Time := ptime;
        -- construct an absolute time plan
        for mk_(fltp, fltime) in responseDB(pmt) do
          (newplan := newplan ^ [mk_ (fltp, newtime)];
           newtime := newtime + fltime );
        -- immediately release the first action
        def mk_(fltp, fltime) = hd newplan;
            t = World`timerRef.GetTime() in
          releaseFlare(evid,fltp,fltime,t);
         -- store the rest of the plan
         curplan := tl newplan;
         eventid := evid;
         curprio := missilePriority(pmt);
         busy := true )
pre pmt in set dom missilePriority and
    pmt in set dom responseDB;

private Step: () ==> ()
Step () ==
  (if len curplan > 0
   then (dcl curtime : Time := World`timerRef.GetTime(),
             done : bool := false;
         while not done do
           (dcl first : PlanStep := hd curplan,
                next : Plan := tl curplan;
            let mk_(fltp, fltime) = first in
              if fltime <= curtime
              then (releaseFlare(eventid,fltp,fltime,curtime);
                    curplan := next;
                    if len next = 0
                    then (curprio := 0;
                          done := true;
                          busy := false ) )
              else done := true ) ) );

private releaseFlare: EventId * FlareType * Time * Time ==> ()
releaseFlare (evid, pfltp, pt1, pt2) ==
  World`env.handleEvent(evid,pfltp,aperture,pt1,pt2);

public isFinished: () ==> ()
isFinished () == skip;
```

```
sync

mutex (Step);
mutex (addThreat);
per isFinished => not busy

end FlareDispenser
```

C.3.9 The TimeStamp Class

```
\begin{vdm_al}
class TimeStamp

values

public stepLength : nat = 1;

instance variables

currentTime  : nat   := 0;
wakeUpMap    : map nat to [nat] := {|->};
barrierCount : nat := 0;
registeredThreads : set of BaseThread := {};
isInitialising : bool := true;
-- singleton instance of class
private static timeStamp : TimeStamp := new TimeStamp();

operations

-- private constructor (singleton pattern)
private TimeStamp : () ==> TimeStamp
TimeStamp() ==
  skip;

-- public operation to get the singleton instance
public static GetInstance: () ==> TimeStamp
GetInstance() ==
  return timeStamp;

public RegisterThread : BaseThread ==> ()
RegisterThread(t) ==
 (barrierCount := barrierCount + 1;
  registeredThreads := registeredThreads union {t};
 );

public UnRegisterThread : BaseThread ==> ()
UnRegisterThread(t) ==
 (barrierCount := barrierCount - 1;
  registeredThreads := registeredThreads \ {t};
```

```
 );

public IsInitialising: () ==> bool
IsInitialising() ==
  return isInitialising;

public DoneInitialising: () ==> ()
DoneInitialising() ==
 (if isInitialising
  then (isInitialising := false;
        for all t in set registeredThreads
        do
          start(t);
       );
 );

public WaitRelative : nat ==> ()
WaitRelative(val) ==
 (WaitAbsolute(currentTime + val);
 );

public WaitAbsolute : nat ==> ()
WaitAbsolute(val) == (
  AddToWakeUpMap(threadid, val);
  -- Last to enter the barrier notifies the rest.
  BarrierReached();
  -- Wait till time is up
  Awake();
);

BarrierReached : () ==> ()
BarrierReached() ==
 (while (card dom wakeUpMap = barrierCount)
  do
   (currentTime := currentTime + stepLength;
    let threadSet : set of nat = {th | th in set dom wakeUpMap
                & wakeUpMap(th) <> nil and wakeUpMap(th) <= currentTime }
    in
      for all t in set threadSet
      do
        wakeUpMap := {t} <-: wakeUpMap;
   );
)
post forall x in set rng wakeUpMap & x = nil or x >= currentTime;

AddToWakeUpMap : nat * [nat] ==> ()
AddToWakeUpMap(tId, val) ==
   wakeUpMap := wakeUpMap ++ { tId |-> val };

public NotifyThread : nat ==> ()
NotifyThread(tId) ==
```

```
 wakeUpMap := {tId} <-: wakeUpMap;

public GetTime : () ==> nat
GetTime() ==
  return currentTime;

Awake: () ==> ()
Awake() == skip;

public ThreadDone : () ==> ()
ThreadDone() ==
  AddToWakeUpMap(threadid, nil);

sync
  per Awake => threadid not in set dom wakeUpMap;
  mutex (IsInitialising);
  mutex (DoneInitialising);
  mutex (AddToWakeUpMap);
  mutex (NotifyThread);
  mutex (BarrierReached);
  mutex (AddToWakeUpMap, NotifyThread);
  mutex (AddToWakeUpMap, BarrierReached);
  mutex (NotifyThread, BarrierReached);
  mutex (AddToWakeUpMap, NotifyThread, BarrierReached);

end TimeStamp
```

C.3.10 The BaseThread Class

```
class BaseThread

types

public static ThreadDef ::
  p : nat1
  isP : bool;

instance variables

protected period : nat1 := 1;
protected isPeriodic : bool := true;

protected registeredSelf : BaseThread;
protected timeStamp : TimeStamp := TimeStamp`GetInstance();

operations

protected BaseThread : BaseThread ==> BaseThread
BaseThread(t) ==
```

```
 (registeredSelf:= t;
  timeStamp.RegisterThread(registeredSelf);
  if(not timeStamp.IsInitialising())
  then start(registeredSelf);
 );

Step : () ==> ()
Step() ==
  is subclass responsibility;

thread

 (if isPeriodic
  then (while true
        do
         (Step();
          timeStamp.WaitRelative(period)
         )
       )
  else (Step();
        timeStamp.WaitRelative(0);
        timeStamp.UnRegisterThread(registeredSelf);
       );
 );

end BaseThread
```

C.3.11 The IO Class

```
class IO

--      Overture STANDARD LIBRARY: INPUT/OUTPUT
--      ---------------------------------------
--
-- Standard library for the Overture Interpreter. When the interpreter
-- evaluates the preliminary functions/operations in this file,
-- corresponding internal functions is called instead of issuing a run
-- time error. Signatures should not be changed, as well as name of
-- module (VDM-SL) or class (VDM++). Pre/post conditions is
-- fully user customisable.
-- Dont care's may NOT be used in the parameter lists.
--
-- The in/out functions  will return false if an error occurs. In this
-- case an internal error string will be set (see 'ferror').

types

public
filedirective = <start>|<append>
```

```
functions

-- Write VDM value in ASCII format to std out:
public
writeval[@p]: @p -> bool
writeval(val)==
  is not yet specified;

-- Write VDM value in ASCII format to file.
-- fdir = <start> will overwrite existing file,
-- fdir = <append> will append output to the file (created if
-- not existing).
public
fwriteval[@p]:seq1 of char * @p * filedirective -> bool
fwriteval(filename,val,fdir) ==
  is not yet specified;

-- Read VDM value in ASCII format from file
public
freadval[@p]:seq1 of char -> bool * [@p]
freadval(f) ==
  is not yet specified
  post let mk_(b,t) = RESULT in not b => t = nil;

operations

-- Write text to std out. Surrounding double quotes will be stripped,
-- backslashed characters should be interpreted.
public
echo: seq of char ==> bool
echo(text) ==
  fecho ("",text,nil);

-- Write text to file like 'echo'
public
fecho: seq of char * seq of char * [filedirective] ==> bool
fecho (filename,text,fdir) ==
  is not yet specified
  pre filename = "" <=> fdir = nil;

-- The in/out functions  will return false if an error occur. In this
-- case an internal error string will be set. 'ferror' returns this
-- string and set it to "".
public
ferror:()  ==> seq of char
ferror () ==
  is not yet specified;

-- New simplified format printing operations
-- The questionmark in the signature simply means any type
```

```
public static print: ? ==> ()
print(arg) ==
  is not yet specified;

-- New simplified format printing operations
-- The questionmark in the signature simply means any type
public static printf: seq of char * seq of ? ==> ()
printf(format, args) ==
  is not yet specified;

end IO
```

C.4 Real-Time Concurrent VDM-RT Model for Counter Measures System

C.4.1 The CM Class

```
system CM

instance variables

-- cpu to deploy sensor 1 and 2
cpu1 : CPU := new CPU (<FCFS>,1E6);

-- cpu to deploy sensor 3 and 4
cpu2 : CPU := new CPU (<FCFS>,1E6);

-- cpu to deploy the MissileDetector
-- and the FlareControllers
cpu3 : CPU := new CPU (<FP>,1E9);

-- cpus for the flare dispensers
cpu4 : CPU := new CPU (<FCFS>,1E8);
cpu5 : CPU := new CPU (<FCFS>,1E8);
cpu6 : CPU := new CPU (<FCFS>,1E8);

-- bus to connect sensors 1 and 2 to the missile detector
bus1 : BUS := new BUS (<FCFS>,1E3,{cpu1,cpu3});

-- bus to connect sensors 3 and 4 to the missile detector
bus2 : BUS := new BUS (<FCFS>,1E3,{cpu2,cpu3});

-- bus to connect flare controllers to the dispensers
bus3 : BUS := new BUS (<FCFS>,1E3,{cpu3,cpu4,cpu5,cpu6});

-- maintain a link to the detector
public static detector : MissileDetector := new MissileDetector(nil);
```

```
public static sensor0 : Sensor := new Sensor(detector,0);
public static sensor1 : Sensor := new Sensor(detector,90);
public static sensor2 : Sensor := new Sensor(detector,180);
public static sensor3 : Sensor := new Sensor(detector,270);

public static controller0 : FlareController := new FlareController(0, nil);
public static controller1 : FlareController := new FlareController(120, nil);
public static controller2 : FlareController := new FlareController(240, nil);

public static dispenser0 : FlareDispenser := new FlareDispenser(0,
                            mk_BaseRTThread`ThreadDef(1000E6,true,0,0,0));
public static dispenser1 : FlareDispenser := new FlareDispenser(30,
                            mk_BaseRTThread`ThreadDef(1000E6,true,0,0,0));
public static dispenser2 : FlareDispenser := new FlareDispenser(60,
                            mk_BaseRTThread`ThreadDef(1000E6,true,0,0,0));
public static dispenser3 : FlareDispenser := new FlareDispenser(90,
                            mk_BaseRTThread`ThreadDef(1000E6,true,0,0,0));

public static dispenser4 : FlareDispenser := new FlareDispenser(0,
                            mk_BaseRTThread`ThreadDef(1000E6,true,0,0,0));
public static dispenser5 : FlareDispenser := new FlareDispenser(30,
                            mk_BaseRTThread`ThreadDef(1000E6,true,0,0,0));
public static dispenser6 : FlareDispenser := new FlareDispenser(60,
                            mk_BaseRTThread`ThreadDef(1000E6,true,0,0,0));
public static dispenser7 : FlareDispenser := new FlareDispenser(90,
                            mk_BaseRTThread`ThreadDef(1000E6,true,0,0,0));

public static dispenser8 : FlareDispenser := new FlareDispenser(0,
                            mk_BaseRTThread`ThreadDef(1000E6,true,0,0,0));
public static dispenser9 : FlareDispenser := new FlareDispenser(30,
                            mk_BaseRTThread`ThreadDef(1000E6,true,0,0,0));
public static dispenser10 : FlareDispenser := new FlareDispenser(60,
                            mk_BaseRTThread`ThreadDef(1000E6,true,0,0,0));
public static dispenser11 : FlareDispenser := new FlareDispenser(90,
                            mk_BaseRTThread`ThreadDef(1000E6,true,0,0,0));

operations

public CM: () ==> CM
CM () ==
  (cpu3.deploy(detector);
--   cpu3.setPriority(MissileDetector`addThreat,100);

   -- set-up sensor 0 and 1
   cpu1.deploy(sensor0);
--   cpu1.setPriority(Sensor`trip,100);
   cpu1.deploy(sensor1);

   -- set-up sensor 2 and 3
   cpu2.deploy(sensor2);
```

```
--   cpu2.setPriority(Sensor`trip,100);
   cpu2.deploy(sensor3);

   -- add the first controller with four dispensers
   cpu3.deploy(controller0);
--   cpu3.setPriority(FlareController`addThreat,80);
   -- add the dispensers to the controller
   cpu4.deploy(dispenser0);
--   cpu4.setPriority(FlareDispenser`addThreat,100);
--   cpu4.setPriority(FlareDispenser`evalQueue,80);
   cpu4.deploy(dispenser1);
   cpu4.deploy(dispenser2);
   cpu4.deploy(dispenser3);

   -- add the second controller with four dispensers
   cpu3.deploy(controller1);
   -- add the dispensers to the controller
   cpu5.deploy(dispenser4);
--   cpu5.setPriority(FlareDispenser`addThreat,100);
--   cpu5.setPriority(FlareDispenser`evalQueue,80);
   cpu5.deploy(dispenser5);
   cpu5.deploy(dispenser6);
   cpu5.deploy(dispenser7);

   -- add the third controller with four dispensers
   cpu3.deploy(controller2);
   -- add the dispensers to the controller
   cpu6.deploy(dispenser8);
--   cpu6.setPriority(FlareDispenser`addThreat,100);
--   cpu6.setPriority(FlareDispenser`evalQueue,80);
   cpu6.deploy(dispenser9);
   cpu6.deploy(dispenser10);
   cpu6.deploy(dispenser11);
   )

end CM
```

C.4.2 The World Class

```
class World

instance variables

-- maintain a link to the environment
public static timerRef : RTTimeStamp := RTTimeStamp`GetInstance();
public static env : [Environment] := nil;

operations
```

```
public World: () ==> World
World () ==
  (-- set-up the sensors
   env := new Environment("scenario.txt",
                          mk_BaseRTThread`ThreadDef(1000E6,true,10,900,0));
   env.addSensor(CM`sensor0);
   env.addSensor(CM`sensor1);
   env.addSensor(CM`sensor2);
   env.addSensor(CM`sensor3);

   -- add the first controller with four dispensers
   CM`controller0.addDispenser(CM`dispenser0);
   CM`controller0.addDispenser(CM`dispenser1);
   CM`controller0.addDispenser(CM`dispenser2);
   CM`controller0.addDispenser(CM`dispenser3);
   CM`detector.addController(CM`controller0);

   -- add the second controller with four dispensers
   CM`controller1.addDispenser(CM`dispenser4);
   CM`controller1.addDispenser(CM`dispenser5);
   CM`controller1.addDispenser(CM`dispenser6);
   CM`controller1.addDispenser(CM`dispenser7);
   CM`detector.addController(CM`controller1);

   -- add the third controller with four dispensers
   CM`controller2.addDispenser(CM`dispenser8);
   CM`controller2.addDispenser(CM`dispenser9);
   CM`controller2.addDispenser(CM`dispenser10);
   CM`controller2.addDispenser(CM`dispenser11);
   CM`detector.addController(CM`controller2);
   );

-- the run function blocks the user-interface thread
-- until all missiles in the file have been processed
public Run: () ==> ()
Run () ==
  (-- start the environment
   timerRef.DoneInitialising();
   -- wait for the environment to handle all input
   env.isFinished();
   -- wait for the missile detector to finish
   CM`detector.isFinished();
   -- print the result
   env.showResult())

end World
```

C.4.3 The Global Class

```
class GLOBAL

values

public SENSOR_APERTURE = 90;
public FLARE_APERTURE = 120;
public DISPENSER_APERTURE = 30

types

-- there are three different types of missiles
public MissileType = <MissileA> | <MissileB> | <MissileC>;

-- there are nine different flare types, three per missile
public FlareType =
  <FlareOneA> | <FlareTwoA> | <DoNothingA> |
  <FlareOneB> | <FlareTwoB> | <DoNothingB> |
  <FlareOneC> | <FlareTwoC> | <DoNothingC>;

-- the angle at which the missile is incoming
public Angle = nat
inv num == num <= 360;

public Time = nat

operations

public canObserve: Angle * Angle * Angle ==> bool
canObserve (pangle, pleft, psize) ==
  def pright = (pleft + psize) mod 360 in
    if pright < pleft
    -- check between [0,pright> and [pleft,360>
    then return (pangle < pright or pangle >= pleft)
    -- check between [pleft, pright>
    else return (pangle >= pleft and pangle < pright);

public getAperture: () ==> Angle * Angle
getAperture () == is subclass responsibility;

end GLOBAL
```

C.4.4 The Environment Class

```
class Environment is subclass of GLOBAL, BaseRTThread

types

public inline  = EventId * MissileType * Angle * Time;
```

```
public outline = EventId * FlareType * Angle * nat * Time

instance variables

-- access to the VDMTools stdio
io : IO := new IO();

-- the input file to process
inlines : seq of inline := [];

-- the output file to print
outlines : seq of outline := [];

-- maintain a link to all sensors
ranges : map nat to (Angle * Angle) := {|->};
sensors : map nat to Sensor := {|->};
inv dom ranges = dom sensors;

busy : bool := true;

operations

public Environment: seq of char * [ThreadDef] ==> Environment
Environment (fname, tDef) ==
 (def mk_ (-,input) = io.freadval[seq of inline](fname) in
    inlines := input;

  if tDef <> nil
  then (period := tDef.p;
        jitter := tDef.j;
        delay := tDef.d;
        offset := tDef.o;
       );
   BaseRTThread(self);
 );

public addSensor: Sensor ==> ()
addSensor (psens) ==
  duration (0)
  (dcl id : nat := card dom ranges + 1;
   atomic (
    ranges := ranges munion {id |-> psens.getAperture()};
    sensors := sensors munion {id |-> psens}
   )
  );

private createSignal: () ==> ()
createSignal () ==
  duration (0)
  (if len inlines > 0
   then (dcl curtime : Time := time, done : bool := false;
```

```
        while not done do
          def mk_ (eventid, pmt, pa, pt) = hd inlines in
            if pt <= curtime
            then (for all id in set dom ranges do
                    def mk_(papplhs,pappsize) = ranges(id) in
                      if canObserve(pa,papplhs,pappsize)
                      then sensors(id).trip(eventid,pmt,pa);
                  inlines := tl inlines;
                  done := len inlines = 0)
            else done := true)
   else busy := false);

public handleEvent: EventId * FlareType * Angle * Time * Time ==> ()
handleEvent (evid,pfltp,angle,pt1,pt2) ==
  duration (0)
  (outlines := outlines ^ [mk_ (evid,pfltp,angle,pt1,pt2)] );

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

public isFinished : () ==> ()
isFinished () == skip;

public GetAndPurgeOutlines: () ==> seq of outline
GetAndPurgeOutlines() ==
  let res = outlines
  in
    (outlines := [];
     return res);

public Step : () ==> ()
Step() ==
 (createSignal();
 );

sync
  mutex (handleEvent);
  mutex (createSignal);
  per isFinished => not busy;

end Environment
```

C.4.5 The Sensor Class

```
class Sensor is subclass of GLOBAL

instance variables
```

```
-- the missile detector this sensor is connected to
private detector : MissileDetector;

-- the left hand-side of the viewing angle of the sensor
private aperture : Angle;

operations

public Sensor: MissileDetector * Angle ==> Sensor
Sensor (pmd, psa) == ( detector := pmd; aperture := psa);

-- get the left hand-side start point and opening angle
public getAperture: () ==> GLOBAL`Angle * GLOBAL`Angle
getAperture () == return mk_ (aperture, SENSOR_APERTURE);

-- trip is called asynchronously from the environment to
-- signal an event. the sensor triggers if the event is
-- in the field of view. the event is stored in the
-- missile detector for further processing
async public trip: MissileType * Angle ==> ()
trip (pmt, pa) ==
  -- log and time stamp the observed threat
  detector.addThreat(pmt,pa,time)
pre canObserve(pa, aperture, SENSOR_APERTURE)

end Sensor
```

C.4.6 The Missile Detector Class

```
class MissileDetector is subclass of GLOBAL, BaseRTThread

-- the primary task of the MissileDetector is to
-- collect all sensor data and dispatch each event
-- to the appropriate FlareController

instance variables

-- maintain a link to each controller
ranges : map nat to (Angle * Angle) := {|->};
controllers : map nat to FlareController := {|->};
inv dom ranges = dom controllers;

-- collects the observations from all attached sensors
threats : seq of (EventId * MissileType * Angle * Time) := [];

-- status of the missile detector
busy : bool := false

operations
```

```
public MissileDetector: [ThreadDef] ==> MissileDetector
MissileDetector(tDef)==
 (if tDef <> nil
  then (period := tDef.p;
        jitter := tDef.j;
        delay := tDef.d;
        offset := tDef.o;
       );
  BaseRTThread(self);
 );

-- addController is only used to instantiate the model
public addController: FlareController ==> ()
addController (pctrl) ==
  (dcl nid : nat := card dom ranges + 1;
   atomic
    (ranges := ranges munion {nid |-> pctrl.getAperture()};
     controllers := controllers munion {nid |-> pctrl}
    );
   );

-- addThreat is a helper operation to modify the event
-- list. currently events are stored first come first served.
-- one could imagine using a different ordering instead.
async public addThreat: EventId * MissileType * Angle * Time ==> ()
addThreat (evid,pmt,pa,pt) ==
  (threats := threats ^ [mk_ (evid,pmt,pa,pt)];
   busy := true );

-- getThreat is a local helper operation to modify the event list
private getThreat: () ==> EventId * MissileType * Angle * Time
getThreat () ==
  (dcl res : EventId * MissileType * Angle * Time := hd threats;
   threats := tl threats;
   return res );

public isFinished: () ==> ()
isFinished () ==
  for all id in set dom controllers do
    controllers(id).isFinished();

Step: () ==> ()
Step() ==
 (if threats <> []
  then (def mk_ (evid,pmt, pa, pt) = getThreat() in
          for all id in set dom ranges do
            def mk_(papplhs, pappsize) = ranges(id) in
              if canObserve(pa, papplhs, pappsize)
              then controllers(id).addThreat(evid,pmt,pa,pt);
        busy := len threats > 0);
```

```
 );

sync

-- addThreat and getThreat modify the same instance variables
-- therefore they need to be declared mutual exclusive
mutex (addThreat,getThreat);

-- getThreat is used as a 'blocking read' from the main
-- thread of control of the missile detector
per getThreat => len threats > 0;
per isFinished => not busy

end MissileDetector
```

C.4.7 The Flare Controller Class

```
class FlareController is subclass of GLOBAL, BaseRTThread

instance variables

-- the left hand-side of the working angle
private aperture : Angle;

-- maintain a link to each dispenser
ranges : map nat to (Angle * Angle) := {|->};
dispensers : map nat to FlareDispenser := {|->};
inv dom ranges = dom dispensers;

-- the relevant events to be treated by this controller
threats : seq of (EventId * MissileType * Angle * Time) := [];

-- the status of the controller
busy : bool := false

operations

public FlareController: Angle * [ThreadDef] ==> FlareController
FlareController (papp, tDef) ==
 (aperture := papp;

  if tDef <> nil
  then (period := tDef.p;
        jitter := tDef.j;
        delay := tDef.d;
        offset := tDef.o;
       );
  BaseRTThread(self);
 );
```

```
public addDispenser: FlareDispenser ==> ()
addDispenser (pfldisp) ==
  let angle = aperture + pfldisp.GetAngle() in
    (dcl id : nat := card dom ranges + 1;
     atomic
      (ranges := ranges munion
                 {id |-> mk_(angle, DISPENSER_APERTURE)};
       dispensers := dispensers munion {id |-> pfldisp}
      );
      start (pfldisp) );

-- get the left hand-side start point and opening angle
public getAperture: () ==> GLOBAL`Angle * GLOBAL`Angle
getAperture () == return mk_(aperture, FLARE_APERTURE);

-- addThreat is a helper operation to modify the event
-- list. currently events are stored first come first served.
-- one could imagine using a different ordering instead
async public addThreat: EventId * MissileType * Angle * Time ==> ()
addThreat (evid,pmt,pa,pt) ==
  (threats := threats ^ [mk_ (evid,pmt,pa,pt)];
   busy := true );

-- getThreat is a local helper operation to modify the event list
private getThreat: () ==> EventId * MissileType * Angle * Time
getThreat () ==
  (dcl res : EventId * MissileType * Angle * Time := hd threats;
   threats := tl threats;
   return res );

public isFinished: () ==> ()
isFinished () ==
  for all id in set dom dispensers do
    dispensers(id).isFinished();

Step: () ==> ()
Step() ==
  (if threats <> []
   then (def mk_ (evid,pmt, pa, pt) = getThreat() in
         for all id in set dom ranges do
           def mk_(papplhs, pappsize) = ranges(id) in
             if canObserve(pa, papplhs, pappsize)
             then dispensers(id).addThreat(evid,pmt,pt);
         busy := len threats > 0;
        );
   );

sync

-- addThreat and getThreat modify the same instance variables
```

```
-- therefore they need to be declared mutual exclusive
mutex (addThreat,getThreat);

-- getThreat is used as a 'blocking read' from the main
-- thread of control of the missile detector
per getThreat => len threats > 0;
per isFinished => not busy

end FlareController
```

C.4.8 The Flare Dispenser Class

```
class FlareDispenser is subclass of GLOBAL, BaseRTThread

values

responseDB : map MissileType to Plan =
  {<MissileA> |-> [mk_(<FlareOneA>,900),
                   mk_(<FlareTwoA>,500),
                   mk_(<DoNothingA>,100),
                   mk_(<FlareOneA>,500)],
   <MissileB> |-> [mk_(<FlareTwoB>,500),
                   mk_(<FlareTwoB>,700)],
   <MissileC> |-> [mk_(<FlareOneC>,400),
                   mk_(<DoNothingC>,100),
                   mk_(<FlareTwoC>,400),
                   mk_(<FlareOneC>,500)] };

missilePriority : map MissileType to nat =
  {<MissileA> |-> 1,
   <MissileB> |-> 2,
   <MissileC> |-> 3 }

types

public Plan = seq of PlanStep;

public PlanStep = FlareType * Time;

instance variables

public curplan : Plan := [];
curprio        : nat := 0;
busy           : bool := false;
aparature      : Angle;
eventid        : [EventId];

operations
```

```
public FlareDispenser: Angle * [ThreadDef] ==> FlareDispenser
FlareDispenser(ang, tDef) ==
 (aparature := ang;

  if tDef <> nil
  then (period := tDef.p;
        jitter := tDef.j;
        delay := tDef.d;
        offset := tDef.o;
       );
  BaseRTThread(self);
 );

public GetAngle: () ==> nat
GetAngle() ==
  return aparature;

async public addThreat: EventId * MissileType * Time ==> ()
addThreat (evid, pmt, ptime) ==
  if missilePriority(pmt) > curprio
  then (dcl newplan : Plan :=  [],
            newtime : Time := ptime;
        -- construct an absolute time plan
        for mk_(fltp, fltime) in responseDB(pmt) do
          (newplan := newplan ^ [mk_ (fltp, newtime)];
           newtime := newtime + fltime );
        -- immediately release the first action
        def mk_(fltp, fltime) = hd newplan in
          releaseFlare(evid,fltp,fltime,time);
        -- store the rest of the plan
        curplan := tl newplan;
        eventid := evid;
        curprio := missilePriority(pmt);
        busy := true )
pre pmt in set dom missilePriority and
    pmt in set dom responseDB;

async Step: () ==> ()
Step () ==
  cycles (1E5)
  (if len curplan > 0
   then (dcl curtime : Time := time, done : bool := false;
         while not done do
           (dcl first : PlanStep := hd curplan,
                next : Plan := tl curplan;
            let mk_(fltp, fltime) = first in
              if fltime <= curtime
              then (releaseFlare(eventid,fltp,fltime,curtime);
                    curplan := next;
                    if len next = 0
                    then (curprio := 0;
```

```
                          done := true;
                          busy := false ) )
              else done := true ) ) );

private releaseFlare: EventId * FlareType * Time * Time ==> ()
releaseFlare (evid, pfltp, pt1, pt2) ==
  World`env.handleEvent(evid,pfltp,aparature,pt1,pt2);

public isFinished: () ==> ()
isFinished () == skip

sync
  mutex (addThreat,Step);
  per isFinished => not busy;

end FlareDispenser
```

C.4.9 The RTTimeStamp Class

```
class RTTimeStamp

instance variables

registeredThreads : set of BaseRTThread := {};
isInitialising : bool := true;
-- singleton instance of class
private static rtTimeStamp : RTTimeStamp := new RTTimeStamp();

operations

-- private constructor (singleton pattern)
private RTTimeStamp : () ==> RTTimeStamp
RTTimeStamp() ==
  skip;

-- public operation to get the singleton instance
public static GetInstance: () ==> RTTimeStamp
GetInstance() ==
  return rtTimeStamp;

public RegisterThread : BaseRTThread ==> ()
RegisterThread(t) ==
 (registeredThreads := registeredThreads union {t};
 );

public UnRegisterThread : BaseRTThread ==> ()
UnRegisterThread(t) ==
 (registeredThreads := registeredThreads \ {t};
 );
```

```
public IsInitialising: () ==> bool
IsInitialising() ==
  return isInitialising;

public DoneInitialising: () ==> ()
DoneInitialising() ==
 (if isInitialising
  then (isInitialising := false;
        for all t in set registeredThreads
        do
          start(t);
       );
 );

sync
  mutex (RegisterThread);
  mutex (UnRegisterThread);
  mutex (RegisterThread, UnRegisterThread);
  mutex (IsInitialising);
  mutex (DoneInitialising);

end RTTimeStamp
```

C.4.10 The BaseRTThread Class

```
class BaseRTThread

types

public static ThreadDef ::
  p : nat1
  isP : bool
  j : nat
  d : nat
  o : nat;

instance variables

protected period : nat1 := 1000E6;
protected isPeriodic : bool := true;
protected jitter : nat := 0;
protected delay : nat := 0;
protected offset : nat := 0;

protected registeredSelf : BaseRTThread;
protected timeStamp : RTTimeStamp := RTTimeStamp`GetInstance();

operations
```

```
protected BaseRTThread : BaseRTThread ==> BaseRTThread
BaseRTThread(t) ==
 (registeredSelf := t;
  timeStamp.RegisterThread(registeredSelf);
  if(not timeStamp.IsInitialising())
  then start(registeredSelf);
 );

Step : () ==> ()
Step() ==
  is subclass responsibility;

thread

periodic(period, jitter, delay, offset)(Step);

end BaseRTThread
```

C.4.11 The IO Class

```
class IO

--      Overture STANDARD LIBRARY: INPUT/OUTPUT
--      ---------------------------------------
--
-- Standard library for the Overture Interpreter. When the interpreter
-- evaluates the preliminary functions/operations in this file,
-- corresponding internal functions is called instead of issuing a run
-- time error. Signatures should not be changed, as well as name of
-- module (VDM-SL) or class (VDM++). Pre/post conditions is
-- fully user customisable.
-- Dont care's may NOT be used in the parameter lists.
--
-- The in/out functions  will return false if an error occurs. In this
-- case an internal error string will be set (see 'ferror').

types

public
filedirective = <start>|<append>

functions

-- Write VDM value in ASCII format to std out:
public
writeval[@p]: @p -> bool
writeval(val)==
  is not yet specified;
```

```
-- Write VDM value in ASCII format to file.
-- fdir = <start> will overwrite existing file,
-- fdir = <append> will append output to the file (created if
-- not existing).
public
fwriteval[@p]:seq1 of char * @p * filedirective -> bool
fwriteval(filename,val,fdir) ==
  is not yet specified;

-- Read VDM value in ASCII format from file
public
freadval[@p]:seq1 of char -> bool * [@p]
freadval(f) ==
  is not yet specified
  post let mk_(b,t) = RESULT in not b => t = nil;

operations

-- Write text to std out. Surrounding double quotes will be stripped,
-- backslashed characters should be interpreted.
public
echo: seq of char ==> bool
echo(text) ==
  fecho ("",text,nil);

-- Write text to file like 'echo'
public
fecho: seq of char * seq of char * [filedirective] ==> bool
fecho (filename,text,fdir) ==
  is not yet specified
  pre filename = "" <=> fdir = nil;

-- The in/out functions  will return false if an error occur. In this
-- case an internal error string will be set. 'ferror' returns this
-- string and set it to "".
public
ferror:()  ==> seq of char
ferror () ==
  is not yet specified;

-- New simplified format printing operations
-- The questionmark in the signature simply means any type
public static print: ? ==> ()
print(arg) ==
  is not yet specified;

-- New simplified format printing operations
-- The questionmark in the signature simply means any type
public static printf: seq of char * seq of ? ==> ()
printf(format, args) ==
```

```
  is not yet specified;

end IO
```