




# **VDMTools<sup>®</sup>**

---

**Development Guidelines for  
Real-Time Systems Using  
VDMTools<sup>®</sup>**

### How to contact IFAD:

	+45 63 15 71 31	Phone
	+45 65 93 29 99	Fax
	IFAD Forskerparken 10A DK - 5230 Odense M	Mail
	<a href="http://www.ifad.dk">http://www.ifad.dk</a> <a href="ftp.ifad.dk">ftp.ifad.dk</a>	Web Anonymous FTP server
	<a href="mailto:toolbox@ifad.dk">toolbox@ifad.dk</a> <a href="mailto:info@ifad.dk">info@ifad.dk</a> <a href="mailto:sales@ifad.dk">sales@ifad.dk</a>	Technical support General information Sales and pricing

*Development Guidelines for Real-Time Systems Using VDMTools<sup>®</sup>* — Revised for 6.6

© COPYRIGHT 2000 by IFAD

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement.

This document is subject to change without notice

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Characteristics of Reactive Real-Time Systems	1
1.1.1	Challenges of Reactive Systems	1
1.1.2	Challenges of Concurrency	2
1.1.3	Challenges of Real-Time	2
1.2	Overview of VDM++ and <b>VDMTools®</b>	2
1.2.1	Threads	3
1.2.2	Duration Statements	4
1.3	Timing Analysis	4
1.4	Structure of document	7
<b>2</b>	<b>Development Process For Real-Time Systems</b>	<b>9</b>
2.1	Requirements Capture	10
2.1.1	Capturing Requirements with UML Use Cases	10
2.1.2	Capturing Requirements Using VDM-SL	14
2.1.3	Validating Requirements Capturing	15
2.1.4	Criteria for Completion	15
2.2	Sequential Design Model	16
2.2.1	If UML Was Used For Requirements Capture	16
2.2.2	When VDM-SL was used for Requirements Capturing	17
2.2.3	Class Descriptions in VDM++	18
2.2.4	Validation of Model	18
2.2.5	Criteria for Completion	19
2.3	Concurrent VDM++ Design Model	19
2.3.1	Identification of Threads	19
2.3.2	Communication	20
2.3.3	Synchronization Points	20
2.3.4	Modelling of Environment	20
2.3.5	Validation of Model	20
2.3.6	Criteria for Completion	20
2.4	Concurrent Real-Time VDM++ Design Model	21
2.4.1	Default Duration Information	21
2.4.2	Duration Statements	21

2.4.3	Task Switching Overhead	21
2.4.4	Validation of Model	22
2.4.5	Timing Analysis	22
2.4.6	Criteria for Completion	22
2.5	Implementation	23
2.6	The Different Test Phases	23
2.7	Discussion	24
<b>3</b>	<b>Example Development</b>	<b>25</b>
3.1	The Counter Measures System	25
3.2	UML Use Cases for Requirements Capture	26
3.2.1	Deploy Counter-measures	28
3.2.2	Detect Missile and Select Sequence	29
3.2.3	Fire Sequence	29
3.2.4	Summary	30
3.3	VDM-SL for Requirements Capture	30
3.3.1	Definition of Types	31
3.3.2	Value Definitions	32
3.3.3	The Counter Measures Functionality	32
3.3.4	Summary	35
3.4	VDM++ Class Skeletons	36
3.5	Sequential VDM++ Design Model	36
3.5.1	The Sensor Class	36
3.5.2	The Missile Detector Class	39
3.5.3	The Flare Controller Class	40
3.5.4	The Timer Class	44
3.5.5	The World Class	46
3.5.6	The IO Class	47
3.5.7	The SensorIO Class	48
3.5.8	Summary	49
3.6	Concurrent VDM++ Design Model	51
3.6.1	Summary	60
3.7	Real-Time Concurrent VDM++ Design Model	62
3.7.1	Summary	71
<b>4</b>	<b>Synchronization</b>	<b>73</b>
4.1	Synchronization Primitives	73
4.1.1	History Counters	74
4.1.2	Mutex	75
4.2	Wait-Notify	76
4.3	Thread Completion	77
4.4	Summary	79

<b>5</b>	<b>Periodicity</b>	<b>81</b>
5.1	Periodic Threads . . . . .	81
5.1.1	Periodic Threads and Scheduling . . . . .	82
5.2	Modelling Periodic Events . . . . .	82
5.3	Statically Schedulable Systems . . . . .	83
5.4	Summary . . . . .	83
<b>6</b>	<b>Interrupts</b>	<b>85</b>
6.1	Introduction . . . . .	85
6.2	Modelling Interrupts in VDM++ . . . . .	85
6.3	Example . . . . .	86
6.3.1	The Interrupt Simulator . . . . .	86
6.3.2	The Interrupt Handler . . . . .	88
6.3.3	The Interrupt Buffer . . . . .	89
6.3.4	Other Classes . . . . .	90
6.4	Summary . . . . .	91
<b>7</b>	<b>Scheduling Policies</b>	<b>93</b>
7.1	Primary Scheduling Algorithm . . . . .	93
7.2	Secondary Scheduling Algorithm . . . . .	94
7.2.1	Round-Robin Scheduling . . . . .	94
7.2.2	Priority-based Scheduling . . . . .	94
7.2.3	Priority of Default Thread . . . . .	95
<b>8</b>	<b>Time Trace Analysis</b>	<b>97</b>
8.1	Timed Trace Files . . . . .	97
8.1.1	Example . . . . .	97
8.2	Analysis Tools . . . . .	98
8.2.1	Generic Analysis Tools . . . . .	98
8.2.2	Bespoke Analysis Tools . . . . .	98
8.3	Calibration . . . . .	102
<b>9</b>	<b>Postscript</b>	<b>105</b>
<b>10</b>	<b>References</b>	<b>107</b>
<b>A</b>	<b>Glossary</b>	<b>111</b>
<b>B</b>	<b>Design Patterns</b>	<b>113</b>
B.1	The Fresh Data Pattern . . . . .	113
B.1.1	The Inhabitor . . . . .	114
B.1.2	The Proxy . . . . .	116
B.1.3	The Consumer . . . . .	118
B.2	The Time Stamp Pattern . . . . .	119
B.2.1	The TimeStamp Class . . . . .	119

<b>C Examples In Full</b>	<b>125</b>
C.1 VDM-SL Model for Counter Measures System . . . . .	125
C.2 Sequential VDM++ Model for Counter Measures System . . . . .	128
C.3 Concurrent VDM++ Model for Counter Measures System . . . . .	136
C.4 Real-Time Concurrent VDM++ Model for Counter Measures System .	144
C.5 The Wait-Notify Mechanism . . . . .	153

# Chapter 1

## Introduction

This document describes the envisaged process by which reactive real-time systems should be developed using **VDMTools®**. In particular, the document describes how key features of reactive real-time systems can be modelled and analyzed using **VDMTools®**. 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.

### 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 **VDMTools®**.

#### 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 total correctness approaches are not appropriate.

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.

### 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.

### 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.

## 1.2 Overview of VDM++ and VDMTools®

In this section we give a brief description of those features of VDM++ specifically suited to modelling real-time systems. A more substantial overview of VDM++ can be found in [[LangManPP](#), [UserManPP](#)].



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.

**VDMTools®** is a suite of tools, one of which is the VDM++ Toolbox. The VDM++ Toolbox supports 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 the VDM++ Toolbox has a feature known as the Rose-VDM++ link, which supports round-trip engineering with the tool Rational Rose [Rose&00], enabling users to move back and forth between a graphical UML view and a textual detailed VDM++ view. As described in [Guidelines], this allows the complementary benefits of UML and **VDMTools®** to be exploited.

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

We now describe two key features of VDM++ used when modelling real-time systems: threads and duration statements.

### 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:

<pre> class A  instance variables   i : nat := 0  thread   while i &lt; 10 do     i := i + 1   end end A </pre>	<pre> class B  operations  op : () ==&gt; () op() == ( dcl a : A := new A();   start(a) ) end B </pre>
-----------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------

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 in the above example), which is executed to completion subject to scheduling and descheduling. A periodic thread has the form

```
periodic (period) (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**. Examples of this will follow.

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 Statements

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

```
duration (time) statement
```

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

## 1.3 Timing Analysis

The objective when performing timing analysis using **VDMTools®** 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.

Note that checking timing assertions 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).

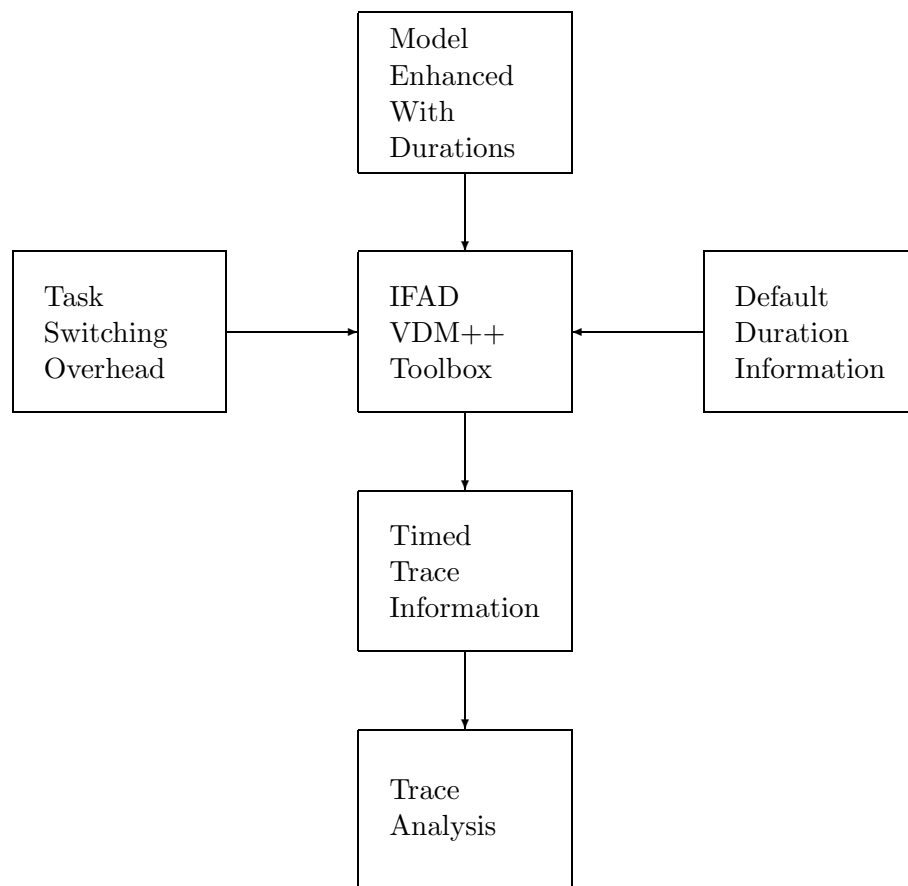


Figure 1.1: Approach to Timing Analysis

To support the above objective the VDM++ Toolbox aims to simulate the target execution environment. That is, it approximates the timing properties of the target processor, and emulates the scheduling policy to be used by the real-time kernel.

The process described and defined in this document assumes that the part of the system being designed will be implemented on a single processor. This does not mean that **VDMTools®** cannot model multi-processor systems; rather that the approach to scheduling and timing analysis within this process is not accurate for multi-processor systems. Note however, that it may be possible to model parts of the environment to the system in question and these parts may be implemented on different processors.

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

- Swapping in and out of threads
- Operation requests, activations and completions

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

Three sources of timing information are used during execution:

**Model enhanced with durations:** When executing a portion of the VDM++ model which falls under the scope of a duration statement, the internal clock is incremented by the given duration at the completion of the statement.

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

**Default duration information:** For those portions of the model not in the scope of a duration statement, conventional worst-case analysis is used. However it is parametrized in terms of the time taken to perform elementary assembly instructions. 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 duration information*.

An overview of the approach is shown in Figure 1.1. The IFAD VDM++ Toolbox 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 **VDMTools®**. 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 **VDMTools®** 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 modelling of systems using interrupts is presented. In Chapter 7 the different kinds of scheduling policy supported by the **VDMTools®** technology are presented. Finally in Chapter 8 it is illustrated what kind of post-execution trace analysis can be performed using **VDMTools®** 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 analysis and design phases.

In Figure 2.1 an overview of the different phases we touch upon in this chapter is presented. This is based on the traditional V-life cycle used by most industrial organizations. 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 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 real-time VDM++ 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 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.

Both approaches are described in the sequel. In Section 2.1.1 a conventional UML [UML&97] 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 [UML&97]. The presentation assumes that the tool Rational Rose [Rose&00] is used. An example of this analysis is given in Section 3.2.

Section 3.2,  
Page 26

#### 2.1.1.1 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.



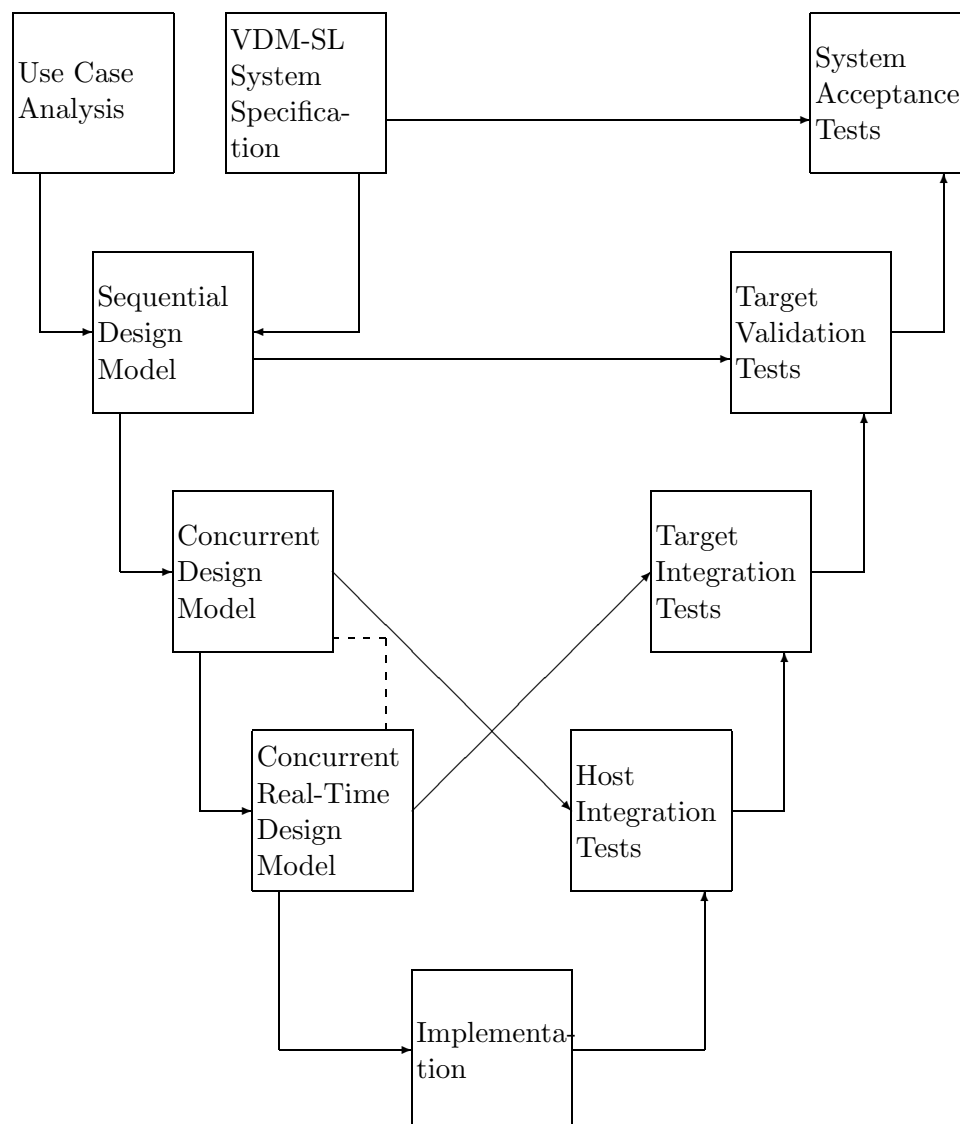


Figure 2.1: Overview of Development Process

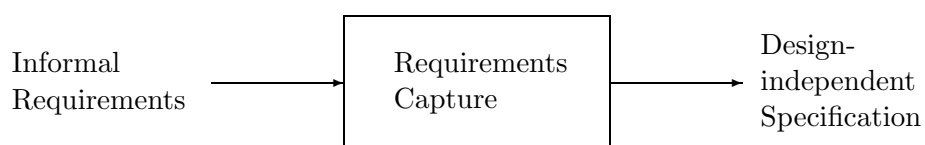


Figure 2.2: Overview of the input/output relationship for Requirements Capture

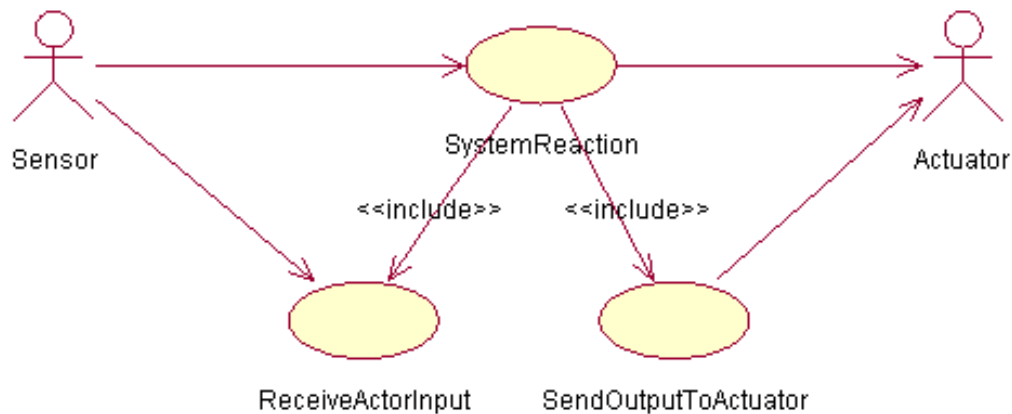


Figure 2.3: General Use Case for Embedded System

- Arrange the actors and use cases into related packages.
- 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.

#### 2.1.1.2 Structure the use case model

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

These stereotypes are defined in the UML 1.3 standard [UML&97].

#### 2.1.1.3 Specify the use cases in detail

For each use case

- Detail the interaction between the actors and the system.

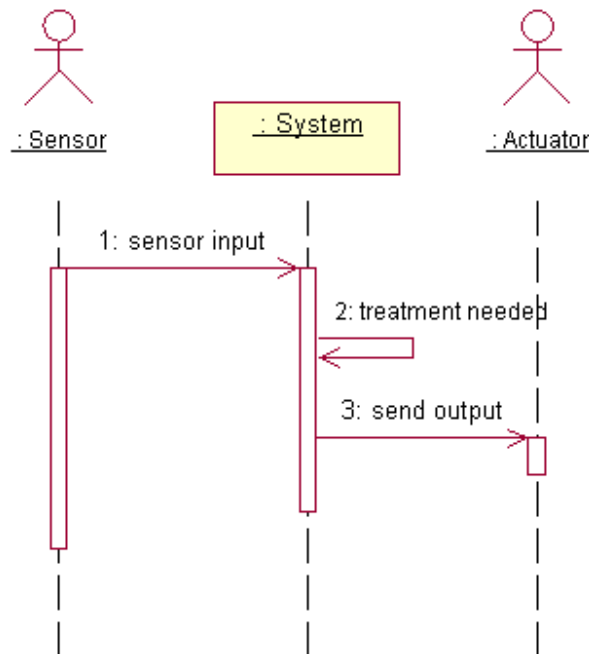


Figure 2.4: General Sequence Diagram for Embedded System

- 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. One can use pseudo-code, e.g. IF/THEN/ELSE/ENDIF and LOOP/REPEAT in these diagrams if one wishes to show several cases. Typically a diagram should fit on one sheet of A4/letter paper. An example of this is shown in Figure 2.4.

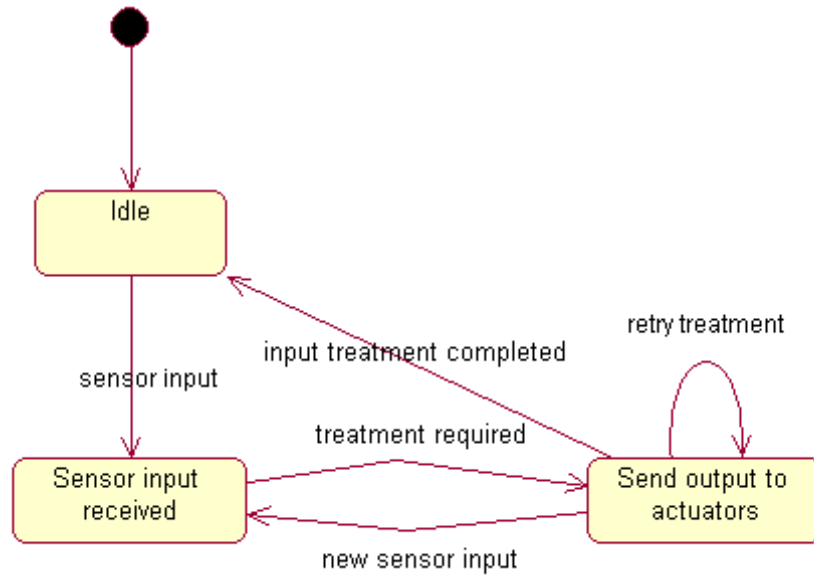


Figure 2.5: General Activity Diagram for Embedded System

**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&98]. 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.

Section 3.3,  
Page 30

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.

### 2.1.4 Criteria for Completion

This stage is complete when:

1. The architecturally significant use cases have been completed **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, 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.

### 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.

#### 2.2.1.1 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. 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.

	Actor	«Entity Class»	«Boundary Class»	«Control Class»
Actor	N/A	✗	✓	✗
«Entity Class»	✗	✗ <sup>1</sup>	✗	✓
«Boundary Class»	✓	✗	✗	✓
«Control Class»	✗	✓	✓	✓

Table 2.1: Rules for Associations Between Classes

A number of recommendations apply to the use of associations between classes. These can be found in Table 2.1 where ✗ indicates that they should not be combined and ✓ indicates that they can be combined.

### 2.2.1.2 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 Rose-VDM++ link feature from the VDM++ Toolbox. This automatically generates a file for each class in the Rose repository, in Microsoft Word RTF format. Each such file contains the VDM++ skeleton of the class. From now on it is possible to adjust both the VDM++ model in the Microsoft Word files and the Rational Rose model. The Rose-VDM++ link is able to merge such changes.

### 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 reuse most of the VDM-SL operations inside VDM++ classes where structuring and design decisions are taken into account (see for example [CashPoint]). 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. However, 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 be synthesized from the VDM-SL model. This is not an automatic process and

<sup>1</sup>Unless there exists an aggregation or composition between the two classes.

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 its own class. 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.

Section 3.4, An example of synthesizing such class skeletons is given in Section 3.4.  
Page 36

### 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).

Section 3.5, An example of completed class descriptions in VDM++ is given in Section 3.5.  
Page 36

### 2.2.4 Validation of 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.

### 2.2.5 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.

Section 3.6,  
Page 51

### 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.

Note that in general the number of threads in the system should be minimized. This is because (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.

### 2.3.4 Modelling of 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 threads which represent this environment. These classes may then be used to inhabit the model with test data.

### 2.3.5 Validation of Model

The model should be executed using the same scheduling policy as that used by the target real-time kernel. Upon execution the model should be deadlock-free. Moreover the model should functionally yield the same results as the sequential model, perhaps modulo some abstraction function.

### 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. 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 VDM++ Design Model

At this stage real-time information is added to the model. The 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. An example of a concurrent real-time model is given in Section 3.7.

Section 3.7,  
Page 62

### 2.4.1 Default Duration Information

If the target processor is not one of the standard ones (currently the Motorola 68040 is the only supported processor), then a file containing default duration information should be defined for the target processor. In principle this is straightforward to define, based on the data book for that processor. However, with modern processors using caching or a pipeline architecture it may be more difficult to be able to achieve precise estimates because the time for different basic instructions depends upon the instruction context.

### 2.4.2 Duration 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 the execution times.

For those parts of the model which are effectively modelling the environment (e.g. a thread which generates input data), execution should be enclosed with a **duration** (0) statement. This is because although those actions consume real time, they do not consume any time on the target processor, which is the time that we are modelling.

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.

### 2.4.3 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 the Toolbox during execution of the model.

#### 2.4.4 Validation of 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 number of time trace files which may be analyzed subsequently (examples of this are shown in Chapter 8). Analysis should establish the following:

- No periodic threads miss their deadlines.
- All real-time response requirements are satisfied.

#### 2.4.6 Criteria for Completion

1. The model is syntax and type correct.
2. The UML class diagram and VDM++ 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. All periodic threads make their deadlines for the tests conducted.
7. Any real-time response requirements are satisfied for the tests conducted.
8. 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 **VDMTools®** C++ Code Generator is appropriate. This implies a massive reduction in effort, since the implementation is obtained by one mouse click. Similarly if Java is the implementation language, **VDMTools®** Java Code Generator can be used.

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++ model into code. For instance [\[Bousquet00\]](#) gives a list of rules that can be applied to translate VDM++ models into an Ada 95 implementation.

## 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 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++ design model with information about the timing behaviour of the intended 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.

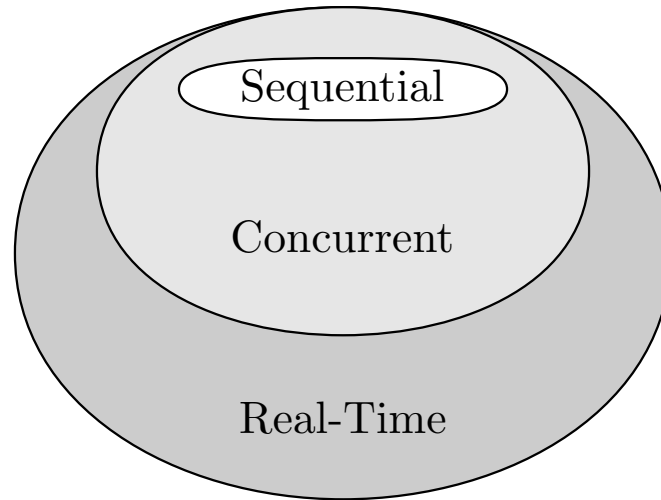


Figure 2.6: Relationship Between VDM++ Models

## 2.7 Discussion

In this chapter we have described the evolution of one VDM-SL model and three VDM++ models: a sequential model, a concurrent model and a concurrent real-time model. There is no formal relationship between them in the sense that no formal refinement or reification 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 2.6. That is to say, the concurrent model is an extension of the sequential one, and the real-time model is an extension of the concurrent one.

## Chapter 3

# Example Development

In this chapter an example development process is presented. This development goes through each of the steps in 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 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. The threat sensor relays the ID of the threat to the controller. For each different kind of ID 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. 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



Figure 3.1: Context Diagram for the Counter Measures System

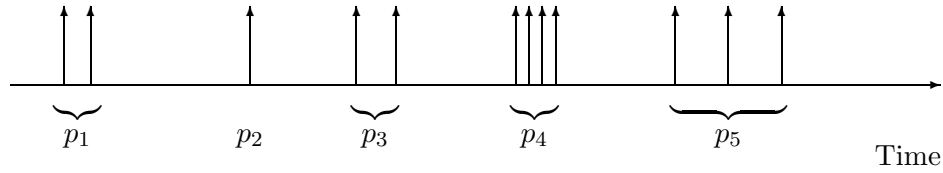


Figure 3.2: Example Firing Sequence

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 to respond to the different missile types as shown by the firing sequences in Figure 3.3.

The following requirements apply to this system:

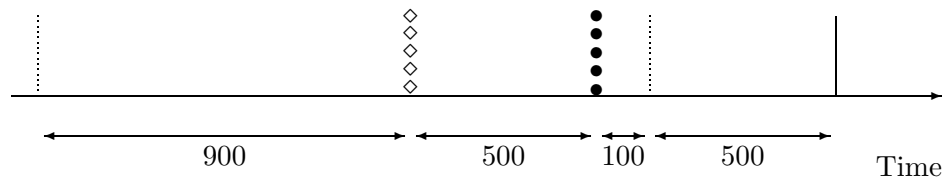
1. If while computing the firing sequence for a given threat, another threat is sensed, the controller should check the priority of the more recent threat and, if greater than the previous one, should abort computation of the current firing sequence. Computation of the new firing sequence should then take place.
2. The controller should be capable of sending the first flare release command within 250 milliseconds of receiving threat information from the sensor.
3. The controller should be able to abort a firing sequence within 130 milliseconds.

## 3.2 UML Use Cases for Requirements Capture

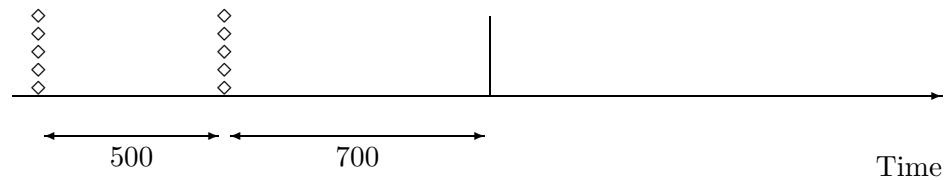
If use cases are used for capturing the requirements for the counter measures system we produce a use case diagram like the one shown in Figure 3.4. For each of the ovals (the graphical representation of a use case) a textual description of the use case must be made. This is typically done in an itemized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



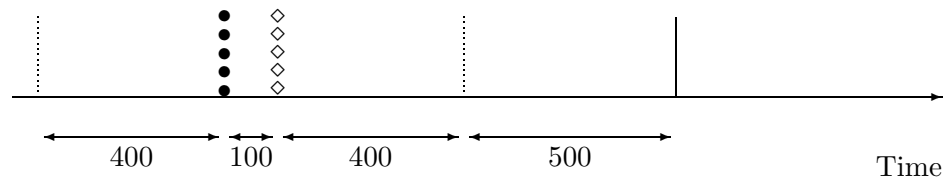
### Response for Missile A



### Response for Missile B



### Response for Missile C



### Key

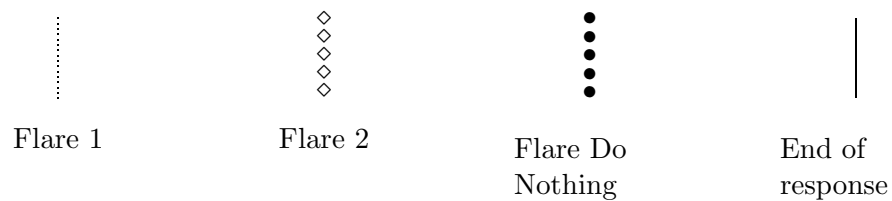


Figure 3.3: Example Missile Responses Used in the Models

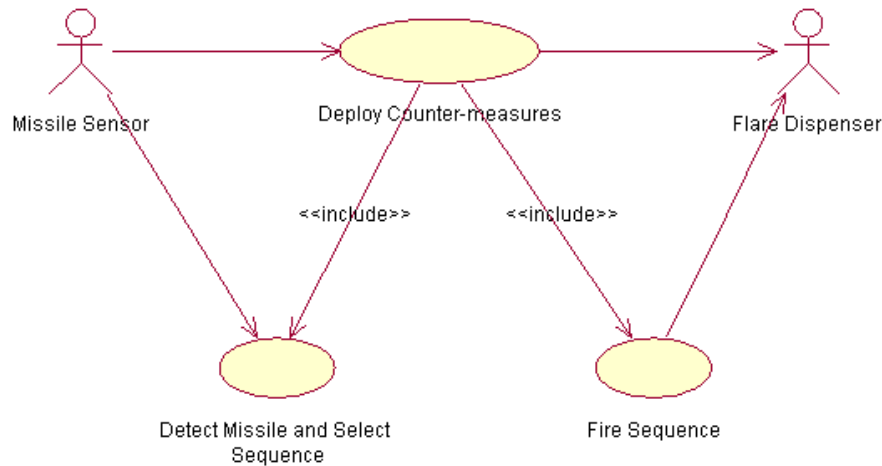


Figure 3.4: Use Case Diagram for the Counter Measures System

### 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 Sequence” and “Fire Sequence”

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

**Course of action:** When a new threatening missile has been detected by the missile sensor it should be determined if another threat is already being handled. 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 the flare dispenser must interrupt what it is currently doing and start firing this new sequence. If the priority is 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 the 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 Sequence

**Primary Actor(s):** Missile Sensor

**Secondary Actor(s):** None

**Intent:** This use case is responsible for taking the given threat identification and based on this determine what sequence of flares 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 is received the responding flare sequence must follow the identification from Figure 3.3.

**Post-conditions:** The correct flare sequence has been identified.

### 3.2.3 Fire Sequence

**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 flares have been released.

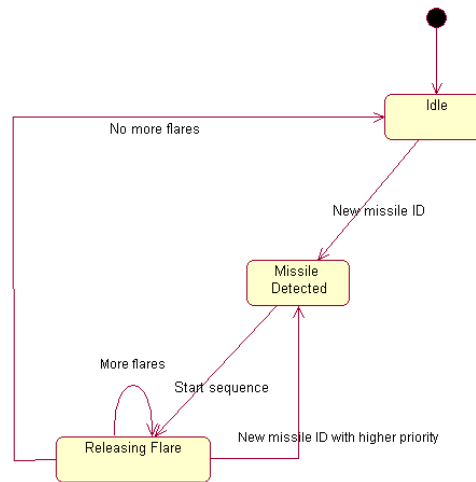


Figure 3.5: State Diagram for Deploying Counter Measures

### 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 2.1.2.

### 3.3.1 Definition of Types

The counter measures system takes **MissileInputs** as input. This is a sequence of values of **MissileType** - 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.

```
types
  MissileInputs = seq of MissileType;

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

The type **Output** represents 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**).

```
Output = seq of OutputStep;

OutputStep = FlareType * AbsTime;

AbsTime = nat;
```

Note that it is assumed that there only two kinds of physical flare (**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
  = {<MissileA> |-> 1,
     <MissileB> |-> 2,
     <MissileC> |-> 3,
     <None> |-> 0};
```

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 : nat = 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, [], <None>, 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 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 the last missile (if there has been none before the **<None>** value is used) which has had effect on the output so far. 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 * [MissileType] * nat -> Output
CM( missileInputs, outputSoFar, lastMissile, curTime) ==
  if missileInputs = []
  then outputSoFar
  else let curMis = hd missileInputs
       in
         if missilePriority(curMis) > missilePriority(lastMissile)
         then let newOutput = InterruptPlan(curTime,outputSoFar,
                                             responseDB(curMis))
              in CM(tl missileInputs, newOutput, curMis,
                    curTime + stepLength)
         else CM(tl missileInputs, outputSoFar, lastMissile,
                 curTime + stepLength);

```

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**.<sup>1</sup> Thus, conversion needs to take place here; this is performed by **MakeOutputFromPlan**.

<sup>1</sup>Note that **Output** is in terms of absolute time, whereas **Plan** is in terms of relative time.

```
InterruptPlan: nat * Output * Plan -> Output
InterruptPlan(curTime, expOutput, plan) ==
  LeavePrefixUnchanged(expOutput, curTime) ^
  MakeOutputFromPlan(curTime, plan);
```

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

```
LeavePrefixUnchanged: Output * nat -> Output
LeavePrefixUnchanged(expOutput, curTime) ==
  [expOutput(i) | i in set inds expOutput
    & let mk_(-,t) = expOutput(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.

```
MakeOutputFromResponse : nat * seq of Response -> Output
MakeOutputFromResponse(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 -> Output
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}]];
```

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];

```

### 3.3.4 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.

## 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 2.2.1.2. 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 three candidate classes, which we might respectively call `MissileDetector`, `FlareController` 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:

**Sensor** A class for modelling the hardware used to sense the arrival of missiles.

**MissileDetector** A class which takes information from the **Sensor** and passes it to the **FlareController**.

**FlareController** A class which outputs flares for a given detected missile.

**Timer** A timer class used to step time throughout the model.

**World** The main class, used to create the class topology and allow execution.

**IO** A VDM++ standard library class.

**SensorIO** A specialization of the **IO** class to read data from a file which is used by the **Sensor**.

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

### 3.5.1 The Sensor Class

The **Sensor** class is used to model the hardware environment. It uses the **SensorIO** class to read missile values from a file (a specific scenario for testing) and makes these

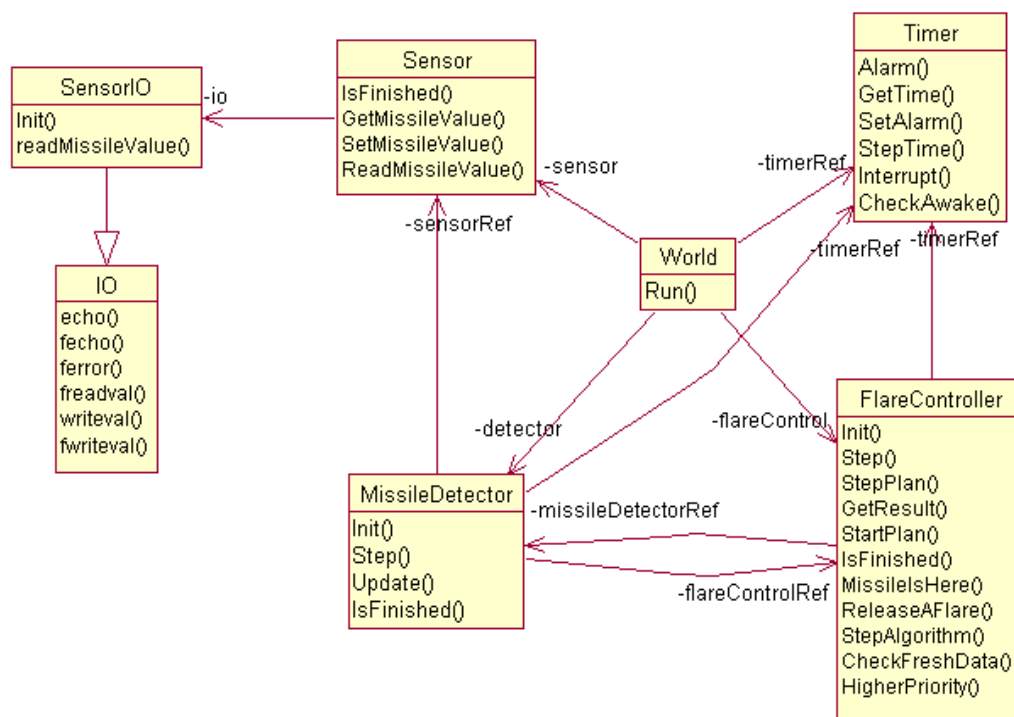


Figure 3.6: Class Diagram for the Sequential Counter Measures Model

missile values available to the `MissileDetector`.

The type `MissileTypes` corresponds to the type of the same name from the VDM-SL model.

```
class Sensor
types

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

The class has two instance variables: a reference to a `SensorIO` object called `io`, and a variable representing the missile most recently read from the `SensorIO` object called `missileValue`. When `missileValue` is consumed by the `MissileDetector` it is set to the special value `<Consumed>`. When there are no more values to be read from `io` it takes the value `nil`.

```
instance variables

io          : SensorIO          := new SensorIO().Init();
missileValue : [MissileType|<Consumed>] := io.readMissileValue();
```

The operation `SetMissileValue` reads the next missile value from the `SensorIO` object.

```
operations

public SetMissileValue : () ==> ()
SetMissileValue() ==
    missileValue := io.readMissileValue();
```

`ReadMissileValue` is used by `World` for debugging purposes, to allow the output to be shown with the input.

```
public ReadMissileValue : () ==> [MissileType]
ReadMissileValue() ==
    return missileValue;
```

`IsFinished` is used to check whether the scenario has finished or not i.e. whether all of the missile values have been read.

```
public IsFinished : () ==> bool
IsFinished() == return missileValue = nil;
```

The `MissileDetector` class uses `GetMissileValue` to acquire the most recent missile value. If a valid (i.e. non `nil`) value is available, then `missileValue` is set to be `<Consumed>` to indicate that this value has been acquired by the missile detector.

```

public GetMissileValue : () ==> [MissileType]
GetMissileValue() ==
  let orgMissileValue = missileValue in
  (if missileValue <> nil
   then missileValue := <Consumed>;
   return orgMissileValue);

end Sensor

```

### 3.5.2 The Missile Detector Class

The **MissileDetector** class takes information concerning arriving missiles from the **Sensor** class and passes this information on to the **FlareController** class.

```

class MissileDetector

instance variables

```

The class has the following instance variables:

**sensorRef** A reference to the sensor.

**flareControlRef** A reference to the flare controller.

**missileValue** The most recent missile value read from the sensor (or <None> if no value has been read).

**timerRef** A reference to a timer object.

```

sensorRef      : Sensor;
flareControlRef : FlareController;
missileValue    : [Sensor'MissileType] := <None>;
timerRef       : Timer

```

The **Init** operation is used to initialize the object references in the instance variables.

```

operations

public Init : Sensor * FlareController * Timer ==> ()
Init (newSensor, newFlareController, newTimer) ==
  (sensorRef := newSensor;
   flareControlRef := newFlareController;
   timerRef := newTimer
  );

```

The operation **Step** is used to “step” the algorithm: it takes a missile value from the sensor and uses this to update the **MissileDetector**’s instance variables.

```
public Step : () ==> ()
Step() ==
  let newMissileValue = sensorRef.GetMissileValue() in
  Update(newMissileValue);
```

**Update** updates the **MissileDetector**’s instance variables and, if necessary, interrupts the flare controller to alert it to the arrival of a new missile. This interruption occurs via the shared **Timer** object.

```
Update : [Sensor'MissileType] ==> ()
Update(newMissileType) ==
  (if newMissileType <> <None>
   then
     (missileValue := newMissileType;
      flareControlRef.MissileIsHere(missileValue);
      if missileValue <> nil
      then timerRef.Interrupt();
     )
  );
```

**IsFinished** is used to check whether the **MissileDetector** has read all of the values available in this scenario.

```
public IsFinished : () ==> bool
IsFinished() == return missileValue = nil;

end MissileDetector
```

### 3.5.3 The Flare Controller Class

The job of the **FlareController** class is to release a sequence of flares (a *plan*) corresponding to the highest priority missile most recently detected.

First some types are defined. A **Plan** is a sequence of **PlanSteps**. A **PlanStep** is a pair consisting of a flare to be released, and the amount of time between this release and the next one.

```

class FlareController

types
  Plan = seq of PlanStep;
  public PlanStep = FlareType * nat;
  public FlareType = <FlareOneA> | <FlareTwoA> | <FlareOneB> |
                    <FlareTwoB> | <FlareOneC> | <FlareTwoC> |
                    <DoNothingA> | <DoNothingB> | <DoNothingC>;

```

A number of instance variables are defined:

**missileDetectorRef** a reference to the **MissileDetector** object.

**timerRef** a reference to a **Timer**, which is shared with the **MissileDetector**. This allows the **MissileDetector** to wake the **FlareController**.

**currentMissileValue** this reflects the missile for which the a plan is currently being executed. If no missile has been detected, the **<None>** value is used.

**currentStep** the index of the last **PlanStep** executed for the current **Plan**.

**latestMissileValue** the missile most recently read by the **MissileDetector**.

**fresh** is used to indicate whether **latestMissileValue** has only just been set, or whether it was set some time previously.

Two instance variables are included just for modelling: **outputSequence** records the actual flares that have been released; and **noMoreMissiles** indicates when all of the missiles in the scenario have been detected.

```

instance variables
  missileDetectorRef : MissileDetector;
  timerRef           : Timer;
  currentMissileValue : [Sensor'MissileType] := <None>;
  currentStep        : nat                  := 0;
  fresh              : bool                 := false;
  latestMissileValue : Sensor'MissileType   := <None>;
  outputSequence     : seq of (FlareType * nat) := [];
  numberOfFreshValues : nat                 := 0;
  noMoreMissiles     : bool                 := false;

```

Two values are defined: **responseDB** gives the **Plan** for each missile that can be detected; and **missilePriority** gives the relative priority of each missile. Note that since no **Plan** would be executed if **currentMissileValue** is **<None>**, this value is not in the domain of **responseDB**.

```

values
  responseDB : map Sensor'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 Sensor'MissileType to nat
    = {<MissileA> |-> 1,
      <MissileB> |-> 2,
      <MissileC> |-> 3,
      <None> |-> 0}

```

The `missileDetectorRef` and `timerRef` are initialized using the operation `Init`:

```

operations

public Init : MissileDetector * Timer ==> ()
Init(initMissileDetector, initTimerRef) ==
  (missileDetectorRef := initMissileDetector;
   timerRef := initTimerRef;
  );

```

The operation `Step` represents the behaviour of the algorithm at each time step. If the timer is not forcing the `FlareController` to sleep then it performs as follows: it uses the operation `StepAlgorithm` to find the next `PlanStep` (if any) to execute and release the flare in this plan step; then, if a plan *is* being executed, `timerRef` is used to wait for the delay corresponding to this `PlanStep`.

```

public Step : () ==> ()
Step() ==
  ( if timerRef.CheckAwake()
    then (
      StepAlgorithm();
      if currentMissileValue = nil
      then noMoreMissiles := true
      elseif currentMissileValue <> <None>
      then let mk_(-, delay_val) =
          responseDB(currentMissileValue)(currentStep-1)
        in timerRef.Alarm(delay_val)
    )
  );

```

`StepAlgorithm` is used to update `currentMissileValue` if necessary (using `CheckFreshData`) and release the next flare in the current plan (using `StepPlan`). It blocks if there is no fresh missile detected and no plan is currently being executed.



```

StepAlgorithm : () ==> ()
StepAlgorithm() ==
  (if fresh
    then (
      fresh := false;
      CheckFreshData();
    );
    StepPlan()
  );

```

**CheckFreshData** checks whether the most recently detected missile (**latestMissileValue**) has higher priority than the one for which a plan is currently being executed. If so, then the plan for this new missile is started and the previous one abandoned.

```

operations

CheckFreshData : () ==> ()
CheckFreshData() ==
  (if HigherPriority(latestMissileValue,
                    currentMissileValue)
    then StartPlan(latestMissileValue);
    latestMissileValue := <None>;
    numberOfFreshValues := numberOfFreshValues + 1;
  );

HigherPriority : Sensor'MissileType *
               Sensor'MissileType ==> bool
HigherPriority(latest, current) ==
  return missilePriority(latest) > missilePriority(current);

```

```

StartPlan : Sensor'MissileType ==> ()
StartPlan(newMissileValue) ==
  (currentMissileValue := newMissileValue;
   currentStep := 1
  );

```

The operation **ReleaseAFlare** corresponds to the physical action of releasing a flare.

```

ReleaseAFlare : FlareType ==> ()
ReleaseAFlare(ft) ==
  outputSequence := outputSequence ^ [mk_(ft, timerRef.GetTime())];

```

The operation **StepPlan** is used to execute the next **PlanStep**. If we have reached the end of the plan then **currentMissileValue** and **currentStep** are reset; otherwise the flare to be released in this **PlanStep** is released and **currentStep** is incremented.

```
StepPlan : () ==> ()
StepPlan() ==
  if currentStep <= len responseDB(currentMissileValue)
  then
    (let mk_(flare, -) = responseDB(currentMissileValue)(currentStep)
    in ReleaseAFIare(fIare);
    currentStep := currentStep + 1
  )
  else (currentMissileValue := <None>;
        currentStep := 0
       );
```

IsFinished is used to check whether the current plan has completed execution, and there are no more missiles left in this scenario.

```
public IsFinished : () ==> bool
IsFinished() == return currentStep = 0 and noMoreMissiles;
```

GetResult is used for testing to extract the result of the algorithm.

```
public GetResult : () ==> seq of (FlareType * nat)
GetResult() == return outputSequence;
```

The operation MissileIsHere is used by the MissileDetector to pass detected missile values to the FlareController.

```
public MissileIsHere : [Sensor'MissileType] ==> ()
MissileIsHere(newMissileValue) ==
  ( if newMissileValue not in set {<None>, nil}
  then fresh := true;
    if newMissileValue = nil
    then noMoreMissiles := true
    else latestMissileValue := newMissileValue;
  );

end FlareController
```

### 3.5.4 The Timer Class

The Timer class is used to control evolution of time throughout the system.

```
class Timer

instance variables
```

The **Timer** has the two instance variables: **currentTime** representing the current time in the system; and **currentAlarm**, which is the time of the next alarm i.e. in the period **currentTime** to **currentAlarm** the **FlareController** will sleep. This variable takes the value **nil** if no alarm has been set.

```
currentTime : nat := 0;
currentAlarm : [nat] := nil;
```

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

```
values

stepLength : nat = 100;
```

**Alarm** is used by clients to specify the time until the next alarm.

```
operations

public Alarm : nat ==> ()
Alarm(n) ==
  SetAlarm(n);
```

**CheckAwake** is used by clients to check whether the next alarm (if any) has been reached or not.

```
public CheckAwake : () ==> bool
CheckAwake() ==
  return currentAlarm = nil or
    currentAlarm <= currentTime;
```

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;
```

`SetAlarm` is a private operation used to set the `currentAlarm` instance variable.

```
SetAlarm : nat ==> ()
SetAlarm(n) ==
  currentAlarm := currentTime + n;

public Interrupt : () ==> ()
Interrupt() == currentAlarm := nil;

end Timer
```

### 3.5.5 The World Class

The `World` class consists of just one operation, which is used to set up the object topology and then execute the model. It repeatedly steps the algorithm and then steps the current time, until the `Sensor`, `MissileDetector` and `FlareController` have finished.

```
class World

instance variables
  sensor : Sensor := new Sensor();
  detector : MissileDetector := new MissileDetector();
  flareControl : FlareController := new FlareController();
  timerRef : Timer := new Timer();
  inputVals : seq of ([Sensor'MissileType] * nat) := [];

operations
public Run : () ==> (seq of (FlareController'FlareType * nat)) *
  (seq of ([Sensor'MissileType] * nat))
Run() ==
  (detector.Init(sensor,flareControl,timerRef);
   flareControl.Init(detector,timerRef);

   while not (sensor.IsFinished() and detector.IsFinished() and
              flareControl.IsFinished()) do
     ( inputVals := inputVals ^
        [mk_(sensor.ReadMissileValue(), timerRef.GetTime())];
       if not detector.IsFinished() then detector.Step();
       if not flareControl.IsFinished() then flareControl.Step();
       timerRef.StepTime();
       if not sensor.IsFinished() then sensor.SetMissileValue();
     );
     return mk_(flareControl.GetResult(), inputVals)
   )

end World
```

### 3.5.6 The IO Class

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

```
class IO

-- IFAD VDMTools STANDARD LIBRARY: INPUT/OUTPUT
-- -----
--
-- Standard library for the VDMTools 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 (IFAD VDM-SL) or class (VDM++). Pre/post conditions is
-- fully user customisable.
--
-- Note VDM++: Polymorphic functions are protected. In order to call
-- these functions you should:
-- 1: Either make the standard library class called IO superclass of all
--    classes, or
-- 2: Make a subclass of the standard library class and define
--    an operation that calls the polymorphic function. This operation
--    should then be called elsewhere in the specification.
--
-- 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
end IO
```

### 3.5.7 The SensorIO Class

The class **SensorIO** is a subclass of **IO** specialized for the counter measures example. It is a straightforward class that reads a sequence of missile values from a file, and stores them in the instance variable **mvList**. The instance variable **curIndex** then keeps track of the value of the missile most recently read.

```

class SensorIO is subclass of IO

instance variables

    curIndex : nat := 0;
    mvList : seq of Sensor'MissileType := [];

operations

public Init : () ==> SensorIO
Init() ==
( let mk_(-,list) = freadval[seq1 of Sensor'MissileType]("scenario.txt")
  in
    mvList := list;
    curIndex := 1;
    return self
);

```

```

public readMissileValue : () ==> [Sensor'MissileType]
readMissileValue() ==
    if curIndex <= len mvList
    then (curIndex := curIndex + 1;
          return mvList(curIndex-1))
    else return nil;

end SensorIO

```

### 3.5.8 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 the test cases 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.

The lessons which 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. This validation was conducted using traditional testing techniques

and a large amount of reuse of the test cases from the first VDM-SL model was possible.



### 3.6 Concurrent VDM++ Design Model

From the sequential counter measures model we can model three threads: one each for the sensor, the missile detector and the flare controller. We consider these to be separate threads because they all represent independent activities. Note that the sensor is actually part of the environment, so this thread is used to inhabit the model with test data.

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 **SensorIO** and **IO** are unchanged from the previous model, so are not given again here.

The main change to the sensor class is that a thread has been added. To facilitate this, a number of other minor changes are necessary.

```
class Sensor

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

A reference to the common **Timer** is added as an instance variable; this is because the sensor thread will drive time.

```
instance variables
  io          : SensorIO          := new SensorIO().Init();
  missileValue : [MissileType|<Consumed>] := io.readMissileValue();
  timerRef    : Timer;
```

The sensor thread executes until all of the values in the scenario have been read (i.e. until the most recently read **missileValue** is **nil**). It repeatedly waits until the missile detector has consumed the last missile value read, and then reads a fresh missile value. Following this it steps the time in the shared timer. Before completion of the thread the timer is informed that the sensor has finished. This modelling of time follows the principles presented in Section 5.1.

```
thread
  (while missileValue <> nil do
    ( MissileValueConsumed();
      SetMissileValue();
      timerRef.StepTime());
    timerRef.Finished()
  )
```

Note here that the thread uses the operation `MissileValueConsumed` to wait until the missile detector has read the missile value. This operation is a pure synchronization point; it has no functionality of its own, but instead calls to it block until its permission predicate is satisfied. Its permission predicate specifies that the missile value has been read.

```
operations

MissileValueConsumed : () ==> ()
MissileValueConsumed() ==
    skip

sync

per MissileValueConsumed => missileValue = <Consumed>;
```

The operation `Init` is used to initialize the timer reference.

```
operations

public Init : Timer ==> ()
Init(newTimer) ==
    timerRef := newTimer;
```

The remaining operations have bodies unchanged from the sequential model; however in the presence of concurrency it is necessary to consider issues such as mutual exclusion.

The operation `SetMissileValue` writes to the `missileValue` instance variable, so therefore it must execute mutually exclusively with any readers of `missileValue`. The only such operation is `GetMissileValue`.

```
SetMissileValue : () ==> ()
SetMissileValue() ==
    missileValue := io.readMissileValue();

sync

mutex(SetMissileValue, GetMissileValue);
```

The operation `IsFinished` is used to indicate when the sensor thread has finished. It is purely a synchronization point and blocks until the thread has finished.

```

operations

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

sync

per IsFinished => missileValue = nil;

```

The operation `GetMissileValue` is used by the missile detector thread to read the latest missile value. Calls to it will block until a new value has been read.

```

operations

public GetMissileValue : () ==> [MissileType]
GetMissileValue() ==
  let orgMissileValue = missileValue in
  (if missileValue <> nil
   then missileValue := <Consumed>;
   return orgMissileValue);

sync

per GetMissileValue => missileValue <> <Consumed>;

end Sensor

```

This `MissileDetector` class communicates with the sensor thread via its `sensorRef` instance variable, and with the flare controller thread via its `flareControlRef` instance variable.

```

class MissileDetector
instance variables

sensorRef      : Sensor;
flareControlRef : FlareController;
missileValue   : [Sensor'MissileType] := <None>;
timerRef       : Timer

```

The missile detector thread repeatedly reads values from the sensor thread, and uses this value to call the operation `Update`. The body of the thread corresponds to the operation `Step` from the sequential model.

```
thread

while missileValue <> nil do
  let newMissileValue = sensorRef.GetMissileValue() in
  Update(newMissileValue);
```

The operation `IsFinished` is different from the sequential model, since it uses a permission predicate to block until the thread has finished.

```
operations

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

sync
per IsFinished => missileValue = nil;
```

The operation `Update` is similar to the sequential version, except that the timer reference is always interrupted now (for the sequential version it was only interrupted if `missileValue <> nil`).

```
operations

Update : [Sensor'MissileType] ==> ()
Update(newMissileValue) ==
  (if newMissileValue <> <None>
  then
    (missileValue := newMissileValue;
     flareControlRef.MissileIsHere(missileValue);
     if missileValue <> nil
     then timerRef.Interrupt();
    )
  );
```

The remaining operation is unchanged from the sequential model.

```
public Init : Sensor * FlareController * Timer ==> ()
Init (newSensor, newFlareController, newTimer) ==
  (sensorRef := newSensor;
   flareControlRef := newFlareController;
   timerRef := newTimer
  );

end MissileDetector
```

For the flare controller, the types, values and instance variables are unchanged from the sequential version.

```
class FlareController

types
  Plan = seq of PlanStep;
  public PlanStep = FlareType * nat;
  public FlareType = <FlareOneA> | <FlareTwoA> | <FlareOneB> |
                    <FlareTwoB> | <FlareOneC> | <FlareTwoC> |
                    <DoNothingA> | <DoNothingB> | <DoNothingC>;
```

```
instance variables
  missileDetectorRef : MissileDetector;
  timerRef           : Timer;
  currentMissileValue : [Sensor'MissileType] := <None>;
  currentStep         : nat                 := 0;
  fresh              : bool                 := false;
  latestMissileValue  : Sensor'MissileType := <None>;
  outputSequence      : seq of (FlareType * nat) := [];
  noMoreMissiles      : bool                 := false;
```

```
values
  responseDB : map Sensor'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 Sensor'MissileType to nat
    = {<MissileA> |-> 1,
        <MissileB> |-> 2,
        <MissileC> |-> 3,
        <None> |-> 0}
```

The thread for the flare controller is similar to the operation **Step** from the sequential model. The difference between the two is that whereas in the sequential model an explicit check had to be made to see whether the flare controller should be awake or not (if `timerRef.CheckAwake()`), in this version this check is not needed as the call to `timerRef.Alarm` blocks until either the alarm is reached, or an interruption is received from the missile detector.

```

thread
  while true do
    ( StepAlgorithm();
      if currentMissileValue = nil
      then noMoreMissiles := true
      elseif currentMissileValue <> <None>
      then let mk_(-, delay_val) =
            responseDB(currentMissileValue)(currentStep-1)
            in timerRef.Alarm(delay_val);
    )

```

The operation `StepAlgorithm` has the same body as before, but now has a permission predicate which states that the operation may only be executed if a fresh value has been detected, or the `currentMissileValue` is an actual missile (i.e. a plan is currently being executed).

```

operations

StepAlgorithm : () ==> ()
StepAlgorithm() ==
  (if fresh
   then (
     fresh := false;
     CheckFreshData();
   );
   StepPlan()
  );

sync

per StepAlgorithm => fresh = true or currentMissileValue <> <None>;

```

The operation `IsFinished` blocks until all of the missiles in the scenario have been detected (`noMoreMissiles`) and the flare controller has finished executing its last plan (`currentStep = 0`).

```

operations

public IsFinished : () ==> seq of (FlareType * nat)
IsFinished() == return outputSequence;

sync

per IsFinished => currentStep = 0 and noMoreMissiles;

```

The operation `MissileIsHere` has an unchanged body from the sequential version, but since it writes to `latestMissileValue`, it must be executed mutually exclusively with `CheckFreshData`, which writes to the same instance variable.

```
operations

public MissileIsHere : [Sensor'MissileType] ==> ()
MissileIsHere(newMissileValue) ==
  ( if newMissileValue not in set {<None>, nil}
    then fresh := true;
    if newMissileValue = nil
    then noMoreMissiles := true
    else latestMissileValue := newMissileValue;
  );

sync

mutex(MissileIsHere, CheckFreshData);
```

The remaining operations are unchanged from the sequential version.

```
operations

public Init : MissileDetector * Timer ==> ()
Init(initMissileDetector, initTimerRef) ==
  (missileDetectorRef := initMissileDetector;
   timerRef := initTimerRef;
  );
```

```
CheckFreshData : () ==> ()
CheckFreshData() ==
  (if HigherPriority(latestMissileValue,
                    currentMissileValue)
   then StartPlan(latestMissileValue);
   latestMissileValue := <None>;
  );
```

```
HigherPriority : Sensor'MissileType *
                Sensor'MissileType ==> bool
HigherPriority(latest, current) ==
  return missilePriority(latest) > missilePriority(current);

StartPlan : Sensor'MissileType ==> ()
StartPlan(newMissileValue) ==
  (currentMissileValue := newMissileValue;
   currentStep := 1
  );
```

```
ReleaseAFlare : FlareType ==> ()
ReleaseAFlare(ps) ==
  outputSequence := outputSequence ^ [mk_(ps, timerRef.GetTime())];
```

```
StepPlan : () ==> ()
StepPlan() ==
  if currentStep <= len responseDB(currentMissileValue)
  then
    (let mk_(flare, -) = responseDB(currentMissileValue)(currentStep)
    in ReleaseAFlare(flare);
    currentStep := currentStep + 1
  )
  else (currentMissileValue := <None>;
        currentStep := 0
        );
end FlareController
```

The **Timer** class has an additional instance variable - **finished** - that is used to indicate when the sensor thread has finished. This is because the sensor thread is used to drive time, so when it finishes, time should automatically be stepped to the next requested alarm (since no more missiles will arrive, this is safe).

```
class Timer

instance variables

  currentTime : nat := 0;
  currentAlarm : [nat] := nil;
  finished : bool := false;
```

The **Finished** operation is used to set the instance variable **finished**.

```
operations

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

A new operation call is added to the body of the **Alarm** operation. This is a call to the **WakeUp** operation.

```
operations

public Alarm : nat ==> ()
Alarm(n) ==
  ( SetAlarm(n);
    WakeUp());
```



The `WakeUp` operation blocks until time advances up to or beyond, the alarm, or the alarm is set to nil (an interruption), or the `finished` flag is set. In the latter case, since the sensor thread has finished, time can be automatically advanced to the next alarm time. This is specified in the body of the operation. The `sync` clause also specifies mutual exclusion, to protect the integrity of `currentTime`.

```
WakeUp : () ==> ()
WakeUp() ==
  if currentAlarm <> nil
  then if currentTime < currentAlarm
        then currentTime := currentAlarm;
sync

per WakeUp => finished or
              currentAlarm = nil or
              currentAlarm <= currentTime;
mutex(WakeUp, StepTime, GetTime);
```

The only other difference from the sequential model is that the operations `Interrupt` and `SetAlarm` must be executed mutually exclusively, since they both write to the instance variable `currentAlarm`.

```
operations

public Interrupt : () ==> ()
Interrupt() ==
  currentAlarm := nil;

sync

mutex(SetAlarm, Interrupt);
```

The remainder is unchanged from the sequential version.

```
operations

SetAlarm : nat ==> ()
SetAlarm(n) ==
  currentAlarm := currentTime + n;
```

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

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

values

  stepLength : nat = 100;

end Timer
```

The World class is simplified, as instead of having a loop which increments time and steps the algorithm, it instead creates a number of threads and then releases them.

```
class World

instance variables

  sensor : Sensor := new Sensor();
  detector : MissileDetector := new MissileDetector();
  flareControl : FlareController := new FlareController();
  timerRef : Timer := new Timer();

operations

public Run : () ==> seq of (FlareController'FlareType * nat)
Run() ==
  ( sensor.Init(timerRef);
    detector.Init(sensor,flareControl,timerRef);
    flareControl.Init(detector,timerRef);
    startlist({sensor, detector, flareControl});
    sensor.IsFinished();
    detector.IsFinished();
    return flareControl.IsFinished()
  )

end World
```

### 3.6.1 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.

The lessons which 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 first VDM-SL model was possible.

### 3.7 Real-Time Concurrent VDM++ Design Model

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

- The **Timer** has its own thread to drive time. This means that the **Sensor** no longer needs to drive time in its thread.
- Duration statements are used to indicate portions of the model whose execution time is known from previous experience.
- Duration statements with **duration (0)** are used to prevent environmental behaviour from influencing the timing measurements made.

Otherwise the model is largely unchanged from the concurrent one.

For the **Sensor** class, the types and instance variables are virtually unchanged. The only difference is that the **timer** instance variable is now a privately owned reference to a **Timer** object, rather than a shared one.

```
class Sensor
types

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

instance variables

io          : SensorIO          := new SensorIO().Init();
missileValue : [MissileType|<Consumed>] := io.readMissileValue();
timer       : Timer             := new Timer();
```

The sensor thread shows some changes. First of all, the thread is enclosed with a **duration(0)** statement since it is part of the environment and does not represent execution on the target CPU. The second change is that the constant **stepLength** has been moved from the **Timer** class to this class, and is used in the thread. It is used to specify the delay to be made between reading different values in the scenario. Finally, the private **timer** is started at the start of the thread.

```
values

public stepLength : nat = 100;
```

```

thread

duration(0)
( start(timer);
  while missileValue <> nil do
    ( SkipNum(stepLength);
      SetMissileValue();
    )
  )
)

```

The operation `SkipNum` is used to set an alarm in the local `timer` for the specified period.

```

operations

SkipNum : nat ==> ()
SkipNum(n) ==
(
  timer.Alarm(n);
);

```

`GetMissileValue` is unchanged from the concurrent version, except for the addition of a `duration(0)` statement. This is because arrival of new missiles would normally be indicated to the missile detector thread by a hardware interruption, which would not consume processor time.

```

operations

public GetMissileValue : () ==> [MissileType]
GetMissileValue() ==
duration(0)
let orgMissileValue = missileValue in
(if missileValue <> nil
  then missileValue := <Consumed>;
  return orgMissileValue);

sync

per GetMissileValue => missileValue not in set {<Consumed>, <None>};

```

The remainder of the class is unchanged from the concurrent version.

```

operations

SetMissileValue : () ==> ()
SetMissileValue() ==
  missileValue := io.readMissileValue();

```

```
sync

mutex(SetMissileValue, GetMissileValue);
operations

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

sync

per IsFinished => missileValue = nil;

end Sensor
```

For the `MissileDetector` class, the only change is that the timer is always interrupted, rather than just being interrupted if `missileValue <> nil`. This is for technical reasons, due to the fact that the `Timer` has its own thread now.

```
class MissileDetector

operations

Update : [Sensor'MissileType] ==> ()
Update(newMissileValue) ==
  if newMissileValue <> <None>
  then
    (missileValue := newMissileValue;
     flareControlRef.MissileIsHere(missileValue);
     timerRef.Interruption();
    );
```

```
instance variables

sensorRef      : Sensor;
flareControlRef : FlareController;
missileValue   : [Sensor'MissileType] := <None>;
timerRef       : Timer;
```

```
operations

public Init : Sensor * FlareController * Timer ==> ()
Init (newSensor, newFlareController, newTimer) ==
  (sensorRef := newSensor;
   flareControlRef := newFlareController;
   timerRef := newTimer);
```

```

thread

  while missileValue <> nil do
    let newMissileValue = sensorRef.GetMissileValue() in
    Update(newMissileValue);

```

```

operations

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

sync
  per IsFinished => missileValue = nil;

end MissileDetector

```

The types and values defined in `FlareController` are unchanged.

```

class FlareController

types

  Plan = seq of PlanStep;
  PlanStep = FlareType * nat;
  public FlareType = <FlareOneA> | <FlareTwoA> | <FlareOneB> |
                    <FlareTwoB> | <FlareOneC> | <FlareTwoC> |
                    <DoNothingA> | <DoNothingB> | <DoNothingC>;

```

```

values

  responseDB : map Sensor'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 Sensor'MissileType to nat
    = {<MissileA> |-> 1,
       <MissileB> |-> 2,
       <MissileC> |-> 3,
       <None> |-> 0}

```

The only change in the instance variables are that the type of `outputSequence` is just `seq of FlareType`. This is because the times of the flare releases will be obtained from the time trace file generated during execution.

```
instance variables
  missileDetectorRef : MissileDetector;
  timerRef           : Timer;
  currentMissileValue : [Sensor'MissileType] := <None>;
  currentStep        : nat                  := 0;
  fresh              : bool                 := false;
  latestMissileValue : Sensor'MissileType := <None>;
  outputSequence     : seq of FlareType    := [];
  noMoreMissiles     : bool                := false;
```

Due to the different type of `outputSequence`, the type of `IsFinished` differs also.

```
operations

public IsFinished : () ==> seq of FlareType
IsFinished() == return outputSequence;

sync
  per IsFinished => currentStep = 0 and noMoreMissiles;
```

The only other change is in the definition of `ReleaseAFlare`. This exhibits two changes:

- The body is enclosed within a duration statement, since we know in advance the time taken to release a flare, from previous experience.
- For each different kind of flare, there is an operation to release it. This is so that the different flare release actions can be distinguished in the logfile. The operations themselves have no functional effect.

```
operations

ReleaseAFlare : FlareType ==> ()
ReleaseAFlare(ps) ==
  duration(10)
  ( cases ps:
    <FlareOneA> -> ReleaseFlareOneA(),
    <FlareTwoA> -> ReleaseFlareTwoA(),
    <FlareOneB> -> ReleaseFlareOneB(),
    <FlareTwoB> -> ReleaseFlareTwoB(),
    <FlareOneC> -> ReleaseFlareOneC(),
    <FlareTwoC> -> ReleaseFlareTwoC(),
    <DoNothingA> -> ReleaseFlareDoNothingA(),
    <DoNothingB> -> ReleaseFlareDoNothingB(),
    <DoNothingC> -> ReleaseFlareDoNothingC()
  end;
  outputSequence := outputSequence ^ [ps]
);
```



```
ReleaseFlareOneA : () ==> ()
ReleaseFlareOneA() == skip;
```

```
ReleaseFlareTwoA : () ==> ()
ReleaseFlareTwoA() == skip;
```

```
ReleaseFlareOneB : () ==> ()
ReleaseFlareOneB() == skip;
```

```
ReleaseFlareTwoB : () ==> ()
ReleaseFlareTwoB() == skip;
```

```
ReleaseFlareOneC : () ==> ()
ReleaseFlareOneC() == skip;
```

```
ReleaseFlareTwoC : () ==> ()
ReleaseFlareTwoC() == skip;
```

```
ReleaseFlareDoNothingA : () ==> ()
ReleaseFlareDoNothingA() == skip;
```

```
ReleaseFlareDoNothingB : () ==> ()
ReleaseFlareDoNothingB() == skip;
```

```
ReleaseFlareDoNothingC : () ==> ()
ReleaseFlareDoNothingC() == skip;
```

The remainder of the class is unchanged from the concurrent version.

```
public Init : MissileDetector * Timer ==> ()
Init(initMissileDetector, initTimer) ==
  (missileDetectorRef := initMissileDetector;
   timerRef := initTimer);
```

```
CheckFreshData : () ==> ()
CheckFreshData() ==
  (if HigherPriority(latestMissileValue,
                    currentMissileValue)
   then StartPlan(latestMissileValue);
   latestMissileValue := <None>;
  );
```

```

StepAlgorithm : () ==> ()
StepAlgorithm() ==
  (if fresh
   then (
     fresh := false;
     CheckFreshData();
   );
   StepPlan()
  );

sync

per StepAlgorithm => fresh = true or currentMissileValue <> <None>;

```

```

thread
  while true do
    ( StepAlgorithm();
      if currentMissileValue = nil
      then noMoreMissiles := true
      elseif currentMissileValue <> <None>
      then let mk_(-, delay_val) =
           responseDB(currentMissileValue)(currentStep-1)
           in timerRef.Alarm(delay_val))

```

```

operations

HigherPriority : Sensor'MissileType *
                Sensor'MissileType ==> bool
HigherPriority(latest, current) ==
  return missilePriority(latest) > missilePriority(current);

```

```

StartPlan : Sensor'MissileType ==> ()
StartPlan(newMissileValue) ==
  (currentMissileValue := newMissileValue;
   currentStep := 1
  );

```

```

StepPlan : () ==> ()
StepPlan() ==
  if currentStep <= len responseDB(currentMissileValue)
  then
    (let mk_(flare, -) = responseDB(currentMissileValue)(currentStep)
     in ReleaseAFlare(flare);
     currentStep := currentStep + 1
    )
  else (currentMissileValue := <None>;
        currentStep := 0
       );

```

```

public MissileIsHere : [Sensor'MissileType] ==> ()
MissileIsHere(newMissileValue) ==
  ( if newMissileValue not in set {<None>, nil}
    then fresh := true;
    if newMissileValue = nil
    then noMoreMissiles := true
    else latestMissileValue := newMissileValue;
  );

sync

  mutex(MissileIsHere, CheckFreshData);

end FlareController

```

The **Timer** class differs significantly from the concurrent one since it has its own *periodic* thread. Rather than acting as a device for driving time throughout the model, the **Timer** now acts as a device just for specifying periods of inactivity in the client thread.

It works as follows: the timer may be active or inactive (specified by the **active** instance variable). When inactive the periodic thread has no effect, and the client(s) is(are) engaged in meaningful activity. The timer becomes active when a client requests an alarm. This resets the instance variable **currentTime** to 0; thereafter this instance variable records the time that has passed since the alarm was requested, via the periodic thread. When this equals or exceeds the specified alarm period (recorded in the instance variable **currentAlarm**), the client is released. At any point another client may interrupt a waiting client using the **Interruption** operation, which advances **currentTime** direct to the next alarm time. This periodic thread follows the principles presented in Section 5.2.

```

class Timer

instance variables
  currentTime : nat := 0;
  active : bool := false;
  currentAlarm : nat := 0;

thread
  periodic(100)(IncTime)

```

**IncTime** advances time whenever the timer is active.

```
operations

IncTime: () ==> ()
IncTime() ==
  duration(0)
  if active
  then currentTime := currentTime + Sensor'stepLength;
```

The operation **Alarm** is used by clients to request an alarm.

```
public Alarm : nat ==> ()
Alarm(wakeup) ==
  ( currentTime := wakeup;
    Start();
    WaitAlarm());
```

The operation **WaitAlarm** is used internally to block until the time of the alarm has been reached, and then set the timer to be inactive.

```
WaitAlarm : () ==> ()
WaitAlarm() == Stop();

sync
  per WaitAlarm => currentTime >= currentAlarm;
```

Clients may interrupt an alarm using the **Interruption** operation.

```
operations

public Interruption : () ==> ()
Interruption() ==
  currentTime := currentAlarm;
```

The operations **Start** and **Stop** are used to make the timer active and inactive respectively.

```
Start : () ==> ()
Start() ==
  ( active := true;
    currentTime := 0);

Stop : () ==> ()
Stop() ==
  active := false;
```

The following `sync` clause protects instance variables against concurrent writing.

```
sync
    mutex(Start, Interruption, IncTime);
    mutex(Start, Stop);
end Timer
```

For the `World` class, the only change is that the shared timer object must have its thread started.

```
class World

instance variables

    sensor : Sensor := new Sensor();
    detector : MissileDetector := new MissileDetector();
    flareControl : FlareController := new FlareController();
    timerRef : Timer := new Timer();

operations

public Run : () ==> seq of FlareController'FlareType
Run() ==
duration(0)
( start(timer);
  detector.Init(sensor,flareControl,timer);
  flareControl.Init(detector,timer);
  startlist({sensor, detector, flareControl});
  sensor.IsFinished();
  detector.IsFinished();
  return flareControl.IsFinished()
)

end World
```

As for the concurrent model, the `IO` and `SensorIO` classes are unchanged, and are therefore not repeated here.

### 3.7.1 Summary

In this section we have presented a model of the counter measures system that incorporates timing information. It should be pointed out that to some extent this model is 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.

## 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:

<pre> class Buffer  instance variables   data : [nat] := nil  operations   public Put : nat ==&gt; ()   Put(v) == data := v;  sync   per Put =&gt; data = nil; </pre>	<pre> public Get : () ==&gt; nat Get() ==   let n = data in   (data := nil;    return n) pre data &lt;&gt; nil  sync   per Get =&gt; data &lt;&gt; nil  end Buffer </pre>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------

#### 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:

<pre> class BufferN  values   N : nat = 5  instance variables   data : seq of nat := [];   inv len data &lt;= N  operations   public Put : nat ==&gt; ()   Put(v) ==     data := data ^ [v]; </pre>	<pre> public Get : () ==&gt; nat Get() ==   let n = hd data in   (data := tl data;    return n) pre data &lt;&gt; []  sync   per Put =&gt;     #fin(Put) - #fin(Get) &lt; N;   per Get =&gt;     #fin(Get) &lt; #fin(Put);  end BufferN </pre>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

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

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

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:

<pre>class UnsyncBuffer  instance variables   data : [nat] := nil  operations   public Put : nat ==&gt; ()   Put(v) ==     data := v;</pre>	<pre>public Get : () ==&gt; nat Get() ==   let n = data in     (data := nil;      return n)   pre data &lt;&gt; nil  end UnsyncBuffer</pre>
---------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------

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

<pre>-- Producer thread while true do   (let v = Produce() in    b.Put(v))</pre>	<pre>-- Consumer thread while true do   Consume(b.Get())</pre>
----------------------------------------------------------------------------------	----------------------------------------------------------------

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.

<pre>-- Producer thread while true do   (let v = Produce() in    b.Put(v)    o.Notify();    o.Wait())</pre>	<pre>-- Consumer thread while true do   (o.Wait();    Consume(b.Get());    o.Notify())</pre>
-------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------

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.

Note that Wait-Notify is not a primitive in VDM++; instead it is a user-defined class. Its definition may be found in [Appendix C.5](#).

### 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.

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:

<pre> class Shared  instance variables   data : seq of nat := []  operations    public Put : nat ==&gt; ()   Put(n) ==     data := data ^ [n];    public Get : () ==&gt;     seq of nat   Get() == return data  sync   mutex(Put);   mutex(Put,Get)  end Shared  class SubThread  instance variables   s : Shared  operations </pre>	<pre>     public Init : Shared ==&gt; ()     Init(ns) == s := ns  thread   for i = 1 to 100 do     s.Put(i * i)   end SubThread  class MainThread  operations    public Main :     () ==&gt; seq of nat   Main() ==     ( dcl s : Shared :=       new Shared(),       t : SubThread :=         new SubThread();       t.Init(s);       start(t);       return s.Get()     )  end MainThread </pre>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Figure 4.1: Not Waiting For Thread Completion

<pre> class Shared  instance variables   data : seq of nat := []  operations    public Put : nat ==&gt; ()   Put(n) ==     data := data ^ [n];    public Get :     () ==&gt; seq of nat   Get() == return data; </pre>	<pre>   public IsFinished :     () ==&gt; ()   IsFinished() == skip;  sync   mutex(Put);   mutex(Put,Get);    per IsFinished =&gt;     #fin(Put) = 100 end Shared </pre>
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------

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

<pre> class Shared  instance variables   data : seq of nat := [];   finished : bool := false;  operations    public Put : nat ==&gt; ()   Put(n) == data := data ^ [n];    public Get :     () ==&gt; seq of nat   Get() == return data; </pre>	<pre>   public IsFinished :     () ==&gt; ()   IsFinished() == skip;    public Finished : () ==&gt; ()   Finished() ==     finished := true  sync   mutex(Put);   mutex(Put,Get);   mutex(Finished);   per IsFinished =&gt; finished end Shared </pre>
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

## 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<sup>1</sup> – breakpoints can be placed within the operations of the wait-notify class to help identify any problems.

<sup>1</sup>Alternatively the trace file produced by the VDM++ Toolbox 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) (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 every 100 time units following execution of the statement `start(sp)`.

Note that in a simple system such as the one above, `PollSensor` will be invoked exactly every 100 time units. However 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 message 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:

- Under time limited scheduling, a periodic thread which has not completed by the end of its time slice will be descheduled and later rescheduled in the same manner as a procedural thread. Until it has completed, another periodic invocation will not be made.
- 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:

<pre>class Clock  instance variables   curtime : nat := 0  operations   public GetTime : () ==&gt; nat   GetTime() == return curtime;</pre>	<pre>    Tick : () ==&gt; ()     Tick() == curtime := curtime + 1  thread   periodic (100)(Tick)  end Clock</pre>
---------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------

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.



### 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) (OpA)
end A

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

class C
...
thread
  periodic (110) (OpC)
end C

class Main
operations
  public Run : () ==> ()
  Run() ==
    ( dcl a : A := new A(),
      b : B := new B(),
      c : C := new C();
      ...
      duration (0)
        startlist({a,b,c});
      ...
    )
end Main

```

Note here that the `duration(0) startlist(...)` statement is used to ensure that the time at which all three threads start is the same.

### 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

# Interrupts

### 6.1 Introduction

In the previous chapter statically schedulable systems were described, and the manner in which they can be modelled in VDM++ was demonstrated. In such systems, interaction with the environment and external actors occurs via *polling* - the system checks on the status of the environment and external actors by periodically reading values from sensors. In many systems this polling approach is unsuitable, because of the inherent performance limitations of the approach. Therefore such systems use interrupts - when an external event is sensed by hardware, the CPU is interrupted from its current program so that the event can if necessary be dealt with.

From a VDM++ point of view, interrupts represent a challenge since concurrency is simulated via multithreading. In this chapter the way in which interrupts can be modelled is described. First an overview of the approach is given, then an example is presented.

### 6.2 Modelling Interrupts in VDM++

Since VDM++ simulates concurrency using multithreading, pure interrupt-driven systems can not directly be modelled. Instead, an interrupt is simulated as a thread; specific test scenarios can choose to execute a specified number of interrupts at particular times. Interrupt handlers will exist in another thread which is blocked, waiting for an interrupt to occur. The interrupt simulator thread will have strictly higher priority than any other thread. In this way when executed under priority-based scheduling, whenever a simulated interrupt occurs, it is immediately reported to the VDM++ Toolbox scheduler, since the interrupt simulator thread has highest priority. Handling

of the interrupt can occur via the interrupt handling thread, but this need not be a high priority thread; according to the system and the kind of interrupts being modelled, handling of the interrupt could have high or low priority. However this is a design decision that is possible within this framework. When interrupts are not being generated or handled, normal system activity may occur.

## 6.3 Example

In this section we describe an abstract example of a system containing interrupts and an interrupt handler. A UML class diagram showing the main classes is shown in Figure 6.1.

The main classes in the model are the `InterruptSimulator` and `InterruptHandler`. Note that the former will not actually be implemented - it exists purely for modelling purposes. In this model we have one object for each of these classes, that communicate via a shared `InterruptBuffer`. We begin by describing the `InterruptSimulator`

### 6.3.1 The Interrupt Simulator

The interrupt simulator simulates the effect of interrupts occurring. In this case 10 sporadic interrupts are simulated, with a random delay between them of between 1000 and 5000 milliseconds.

There are three instance variables in the class: the shared interrupt buffer, a boolean representing completion of the interrupt simulation, and a local timer. Since `Timer` is a standard class, it is not presented here.

```
class InterruptSimulator

instance variables
  ib : InterruptBuffer;
  isFinished : bool := false;
  t : Timer := new Timer()
```

An initialization operation is provided to allow initialization of the shared interrupt buffer.

```
operations

public Init : InterruptBuffer ==> ()
Init(b) == ib := b;
```

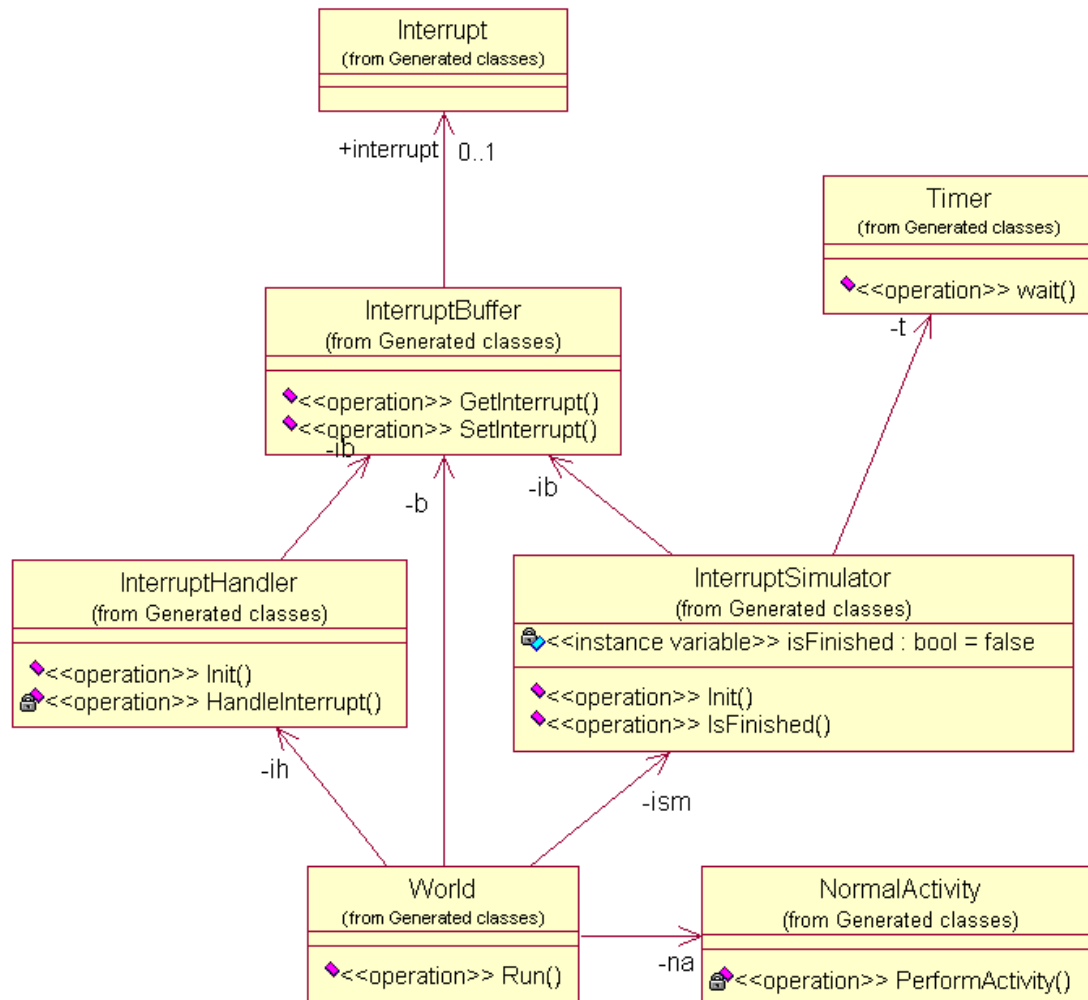


Figure 6.1: Class Diagram of Interrupt Model

The `IsFinished` operation is a simple synchronization point with no functionality.

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

sync
  per IsFinished => isFinished
```

The simulation of interrupts occurs in the interrupt simulator's thread. There the simulator repeatedly waits for a random period, and then executes an interrupt.

```
thread

  ( start(t);
    for count = 1 to 10 do
      ( let del in set {1000,...,5000} in
        ( t.Alarm(del);
          ib.SetInterrupt(new Interrupt())
        );
        isFinished := true;
      )
    )
  )

end InterruptSimulator
```

Since this is an abstract model, the nature of the interrupt is not elaborated. In real models, subclasses of the `Interrupt` class could be used to represent different kinds of interrupt. For instance a missile arrival interrupt in a counter measures system might consist of an identifier for the interrupt.

```
class Interrupt

end Interrupt
```

### 6.3.2 The Interrupt Handler

The interrupt handler class shown here is a skeleton which illustrates how interrupts can be handled. Interrupts are communicated to the handler via the interrupt buffer that is shared with the interrupt simulator.

```
class InterruptHandler

instance variables
  ib : InterruptBuffer
```

The `Init` operation is used to initialize the `ib` instance variable.

```
operations

public Init : InterruptBuffer ==> ()
Init(b) == ib := b;
```

`HandleInterrupt` is used to handle specific interrupts. Since this is just a skeleton class, its definition is left undefined. The key observation is that this operation blocks until an interrupt is available. This is made possible by accessing the public instance variable `interrupt` of `ib` in the permission predicate. This is necessary since operation calls are not allowed in permission predicates.

```
HandleInterrupt : Interrupt ==> ()
HandleInterrupt(interrupt) == is not yet specified;

sync
  per HandleInterrupt => ib.interrupt <> nil
```

The interrupt handler thread then repeatedly handles interrupts as they become available.

```
thread
  while true do
    HandleInterrupt(ib.GetInterrupt())
  end InterruptHandler
```

### 6.3.3 The Interrupt Buffer

The interrupt buffer is used by the interrupt simulator and interrupt handler to communicate. It is a one-place buffer, storing an interrupt object or a nil value, in the instance variable `interrupt`.

```
class InterruptBuffer

instance variables
  public interrupt : [Interrupt] := nil
```

Note that this instance variable is public since it must be readable from within the interrupt handler.

The only operations available are get and set operations for this single instance variable.

```
operations

public SetInterrupt : Interrupt ==> ()
SetInterrupt(ni) ==
  interrupt := ni;

public GetInterrupt : () ==> Interrupt
GetInterrupt() ==
  return interrupt

end InterruptBuffer
```

### 6.3.4 Other Classes

The remaining classes consist of the `NormalActivity` class and the `World` class.

The `NormalActivity` class is an abstract representation of the system's normal behaviour when interrupts are not being handled. Again, since this is an abstract model, no real activity is described.

```
class NormalActivity

operations

PerformActivity : () ==> ()
PerformActivity () == skip

thread
  while true do
    PerformActivity();
  end

end NormalActivity
```

The `World` class creates the various objects in the system, connects the shared buffer between the interrupt simulator and the interrupt handler, and starts the threads.

```
class World

instance variables
  ism : InterruptSimulator := new InterruptSimulator();
  ih  : InterruptHandler   := new InterruptHandler();
  na  : NormalActivity     := new NormalActivity();
  b   : InterruptBuffer    := new InterruptBuffer();

operations
```



```
public Run : () ==> ()
Run() ==
( ism.Init(b);
  ih.Init(b);
  startlist({ism,ih,na});
  ism.IsFinished();
)

end World
```

## 6.4 Summary

In this chapter an approach to modelling interrupts has been described. It uses a thread to model the occurrence of interrupts, and uses priority-based scheduling to ensure that interrupts can be handled as soon as possible after they occur.

An alternative approach is to continually poll for interrupts with high frequency. This has the advantage of being able to use the static scheduling techniques described in the previous chapter. The drawback of this is that polling with high frequency can lead to severe performance deterioration in the model. Therefore unless the implementation is expected to use a polling approach, it is not recommended that this approach is used during the modelling phase.



## Chapter 7

# Scheduling Policies

The VDM++ Toolbox 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 the VDM++ Toolbox 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.

### 7.1 Primary Scheduling Algorithm

The VDM++ Toolbox 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 duration 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.

## 7.2 Secondary Scheduling Algorithm

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

### 7.2.1 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.

### 7.2.2 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.

### 7.2.3 Priority of Default Thread

Consider the following model shown in Figure 7.1

Suppose that in the Toolbox 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 Toolbox 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

<pre> class A  instance variables   data : seq of nat := []  operations    public IsFinished:() ==&gt; ()   IsFinished() == skip;    public Get: () ==&gt;     seq of nat   Get() == return data  sync   per IsFinished =&gt;     len data &gt; 10 </pre>	<pre> thread   (dcl i : nat := 0;    while true do      ( data := data ^ [i];        i := i + 1 )) end A  class B operations   public Main:() ==&gt; seq of nat   Main() ==     ( dcl a : A := new A();       start(a);       a.IsFinished();       a.Get() ) end B </pre>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Figure 7.1: Priority of Default Thread

terminate, since B will never be scheduled due to having a lower priority than A, which is always schedulable.

To avoid this problem, the Toolbox 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.

## Chapter 8

# 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.

### 8.1 Timed Trace Files

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

#### 8.1.1 Example

An extract from a trace file is shown below as an example. White space has been added to improve legibility.

```

fin ->                               Op: FlareController'ReleaseAFlare  Obj: 13
                                      Class: FlareController   @ 188
ThreadSwapOut ->                     ThreadId: 5  Obj: 13  Class: FlareController   @ 204
DelayedThreadSwapIn -> ThreadId: 6  Obj: 11  Class: Timer    @ 204  Delayed: 2
req ->                               Op: Timer'IncTime    Obj: 11  Class: Timer    @ 204
act ->                               Op: Timer'IncTime    Obj: 11  Class: Timer    @ 204
fin ->                               Op: Timer'IncTime    Obj: 11  Class: Timer    @ 204

```

Each entry in the time trace file consists of an event name, and event information, separated by ->. Events fall into two categories, object history 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; the object reference 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.

Thread events correspond to a thread being 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.

## 8.2 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.

### 8.2.1 Generic Analysis Tools

Since the time trace file is simply an ASCII file, manipulation of such files is straightforward (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, as shown in Figure [8.1](#). 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.

### 8.2.2 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.

#### 8.2.2.1 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 [8.2](#).

Given the preponderance of GUI toolkits such as Java Swing [[JavaSwing](#)], Qt [[Qt](#)] and Visual Basic [[VisualBasic](#)], such visualizations are relatively easy to develop. Moreover



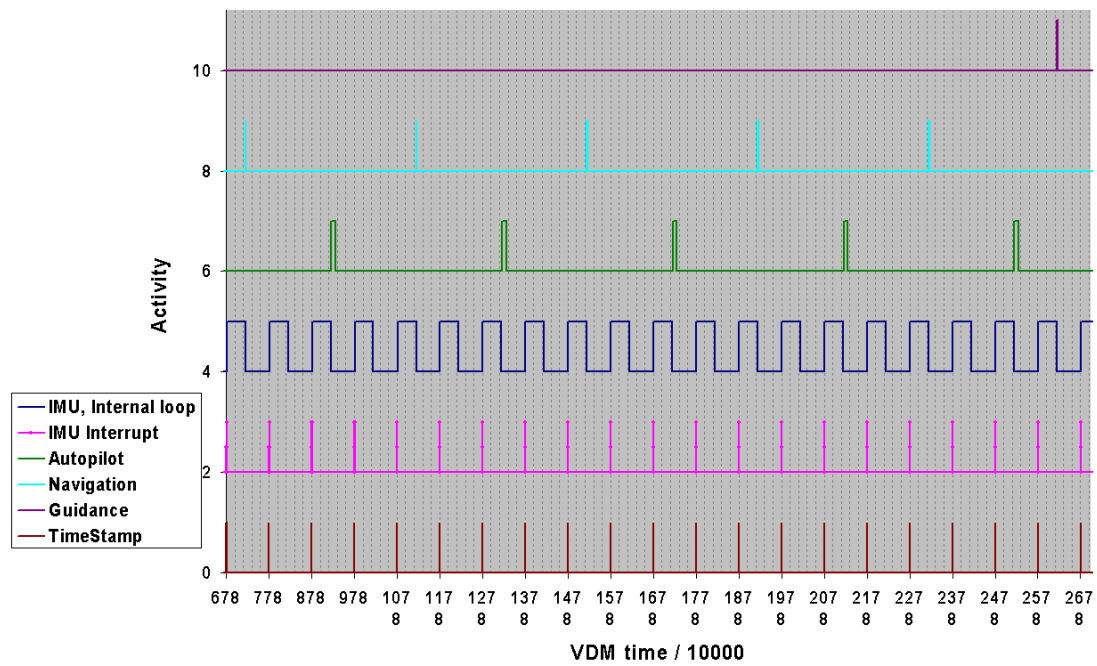


Figure 8.1: A chart in Microsoft Excel generated from a time trace file

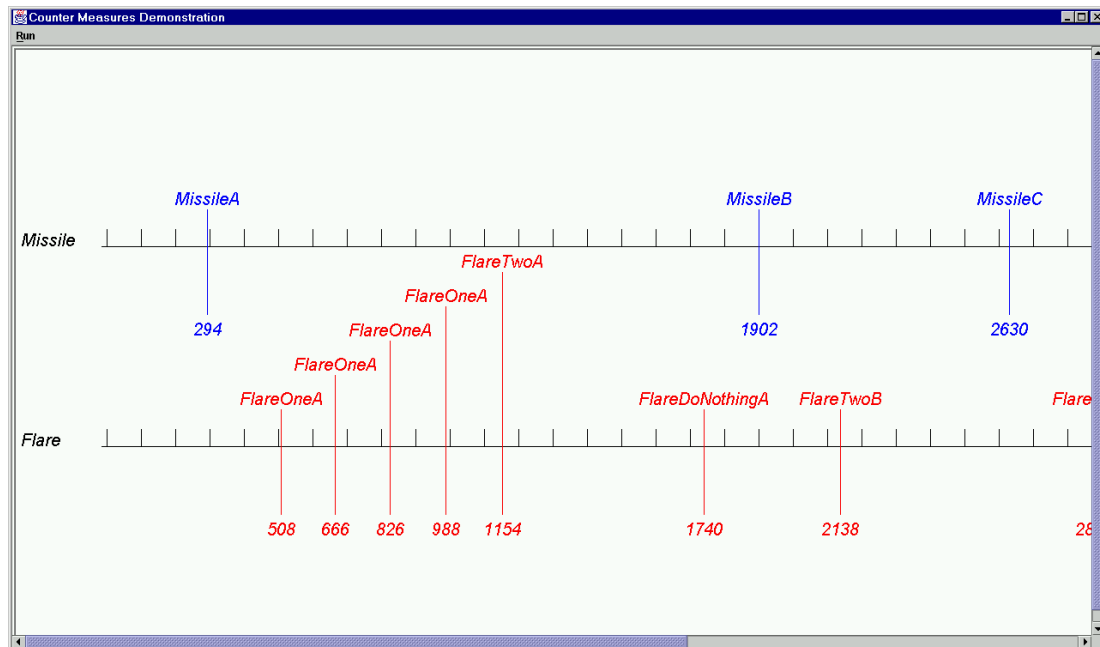


Figure 8.2: Visualization of a time trace file

they can be extremely powerful in communicating the behaviour of a model to a domain expert.

### 8.2.2.2 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 item consists of a trace event, defined below, and an occurrence time.

```
TraceItem :: event : TraceEvent
           time : nat;
```

A trace is an ordered sequence of trace items.

```
Trace = seq of TraceItem
inv t == forall i,j in set inds t &
    i < j => t(i).time <= t(j).time;
```

A trace event corresponds to a thread swap in (normal or delayed), thread swap out, or an operation request activation or completion.

```
TraceEvent = ThreadSwapIn | ThreadSwapOut | DelayedThreadSwapIn |
             OpRequest | OpActivate | OpCompleted;
```

A thread swap in event consists of the id of the thread being swapped in, the object reference owning the thread, and the class name in which the thread is defined.

```
ThreadSwapIn :: id : ThreadId
              objref: [OBJ_Ref]
              clnm : String;
```

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

```
DelayedThreadSwapIn :: id : ThreadId
                    objref: [OBJ_Ref]
                    clnm : [String]
                    delay : real;
```

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

```
ThreadSwapOut :: id : ThreadId
               objref: [OBJ_Ref]
               clnm : [String];
```

An operation request contains the name of the operation, a reference to the object on which the request occurred, and the name of the class which this object is an instance of.

```
OpRequest :: opname : String
            objref : OBJ_Ref
            clnm : String;
```

Operation activations and completions contain the same information as operation requests.

```
OpActivate :: opname : String
             objref : OBJ_Ref
             clnm : String;

OpCompleted :: opname : String
              objref : OBJ_Ref
              clnm : String
```

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.event) => ti.event.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).event) =>
      trace(i).event.opname = opname =>
        let opcompleteIndex = NextOpComplete(opname, i
                                              trace) in
          trace(opcompleteIndex).time - trace(i).time <= 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).event) and trace(j).event.opname = opname]
pre exists j in set inds trace & is_OpCompleted(trace(j).event) and
  j > i and trace(j).event.opname = opname
```

### 8.3 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 machine 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 processor.

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 9

# Postscript

In this document we have described how reactive real-time systems can be developed using **VDMTools®**. We have focussed on how key features of reactive real-time systems can be modelled and analyzed using **VDMTools®**. 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. 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.





## Chapter 10

# 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.
- [Boehm88] B. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21(5):61–72, 1988.
- [Bousquet00] Fabian Bousquet. *VDM++ to Ada95 Translation Rules*. Technical Report VICE-MBDF-17, Matra BAe Dynamics, Rue Grange Dame Rose 20/22, 78141 Velizy-Villacoublay, France, 2000.
- [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, IFAD, June 2000. <ftp://ftp.ifad.dk/pub/vdmtools/doc/cashdispenser.zip>.
- [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&98] 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.
- [Guidelines] The VDM Tool Group. *VDM++ Method Guidelines*. Technical Report, IFAD, October 2000. [ftp://ftp.ifad.dk/pub/vdmttools/doc/guidelines\\_letter.pdf](ftp://ftp.ifad.dk/pub/vdmttools/doc/guidelines_letter.pdf).
- [JavaSwing] Java Swing. December 2000. <http://java.sun.com/products/jfc/tsc/index.html>.
- [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, IFAD, October 2000. [ftp://ftp.ifad.dk/pub/vdmttools/doc/langmanpp\\_letter.pdf](ftp://ftp.ifad.dk/pub/vdmttools/doc/langmanpp_letter.pdf).
- [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.
- [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.

- 
- [UML&97] UML Revision Task Force. *The Unified Modelling Language, version 1.3*. Technical Report, Object Management Group, June 1999.
- [UserManPP] The VDM Tool Group. *VDM++ Toolbox User Manual*. Technical Report, IFAD, October 2000. [ftp://ftp.ifad.dk/pub/vdmttools/doc/usermanpp\\_letter.pdf](ftp://ftp.ifad.dk/pub/vdmttools/doc/usermanpp_letter.pdf).
- [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.



## 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 VDM++ Design Model** A model which defines a particular dynamic architecture including its real-time behaviour.

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

**Default Duration 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 Toolbox, 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 the VDM++ Toolbox, 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 some 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**.

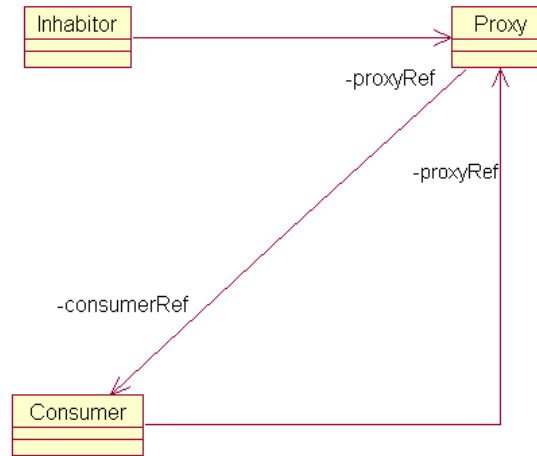


Figure B.1: Class Diagram for Fresh Data Pattern

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.

```

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 operation **Init** is used to initialize the reference to the **Proxy**.

```

operations

  public Init : Proxy ==> ()
  Init (proxy) ==
    proxyRef := proxy;
  
```



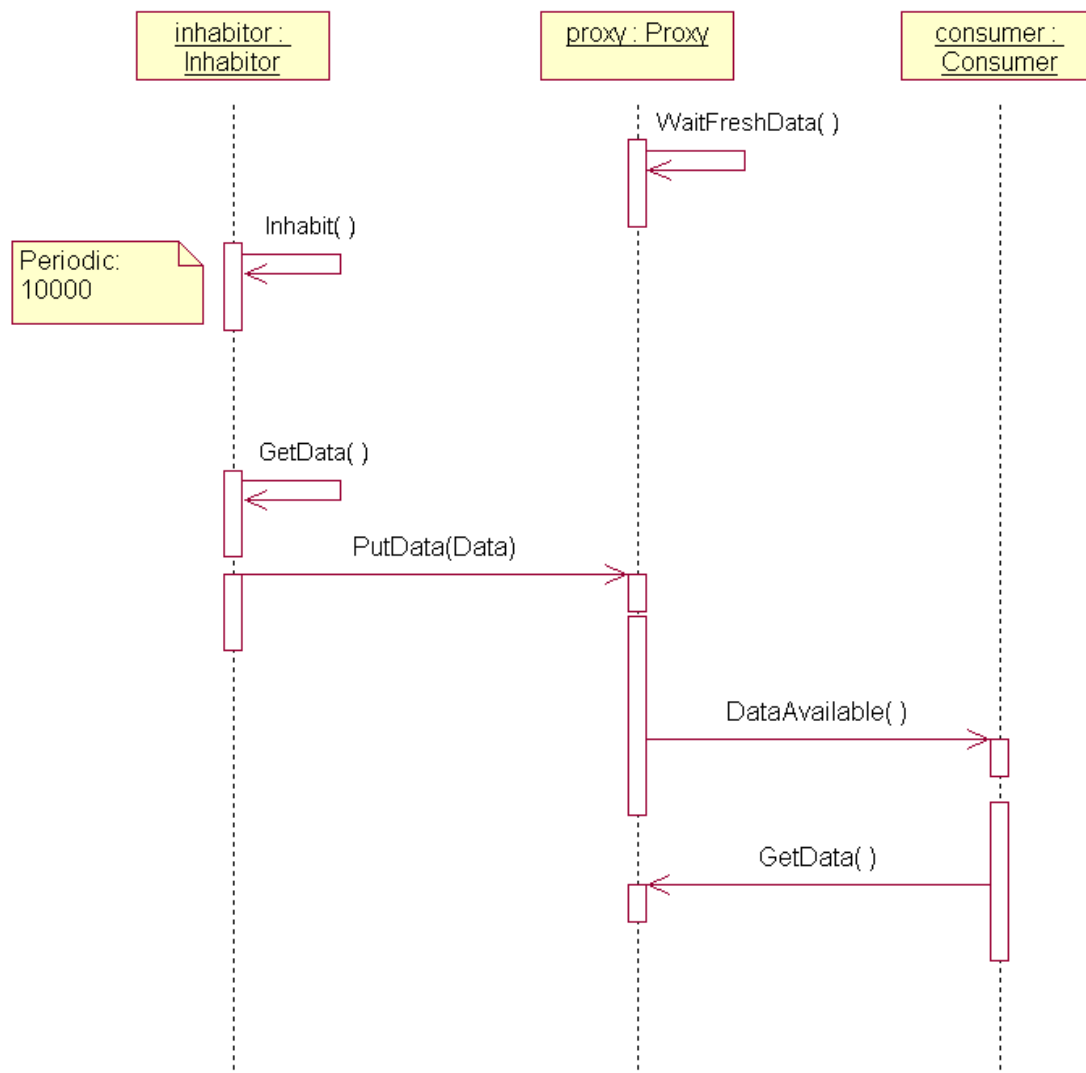


Figure B.2: Sequence Diagram for Fresh Data Pattern

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
```

The `Inhabitor` thread calls the `Inhabit` operation with some fixed period, here arbitrarily chosen to be 10000.

```
thread

  periodic (10000)(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 `Init` operation is used to initialize the `Consumer` reference.

```
operations
```

```
public Init : Consumer ==> ()
Init(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 operation **Init** is used to initialize the reference to the **Proxy**.

```
operations

  public Init : Proxy ==> ()
  Init(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
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*. The main class in the Time Stamp pattern is the `TimeStamp` class, which extends the `WaitNotify` class described in Section 4.2. 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 B.3 with two client classes included for illustrative purposes.

A sample execution is shown in Figure B.4. In this example three clients exist: two “fast” clients and one “slow” client. The two fast clients each have execution period every 10 time steps, while the slow client has execution period every 30 time steps. Each client calls `TimeStamp.WaitRelative` with its execution period, and is then awoken at that time, or as soon as possible thereafter. When awoken each client executes its `ComputationPhase` - that computation it is intended to perform periodically. When it completes its `ComputationPhase` it calls `TimeStamp.Notify` to release any waiting clients. 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 extends the `WaitNotify` class. It maintains a map from thread ids to time (`wakeUpMap`), representing when a particular thread should be woken. It therefore overrides the `WaitNotify` interface to ensure that this map is correctly maintained. 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.

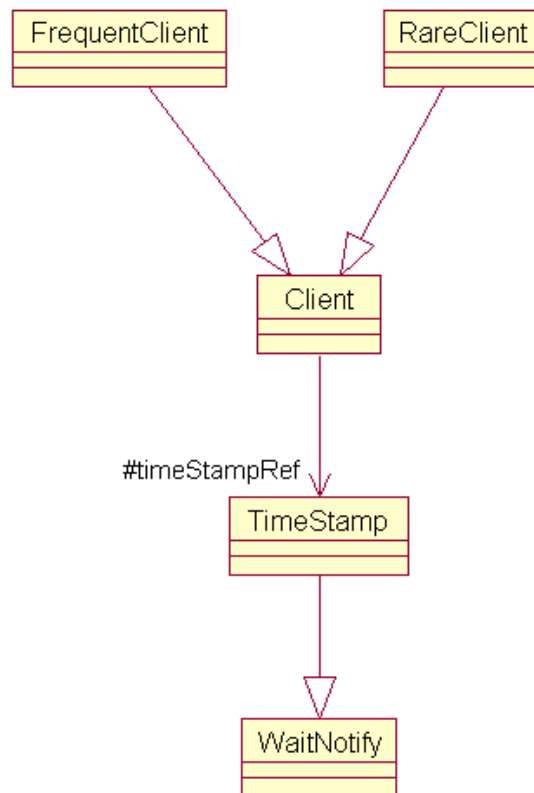


Figure B.3: Class Diagram for Time Stamp Pattern

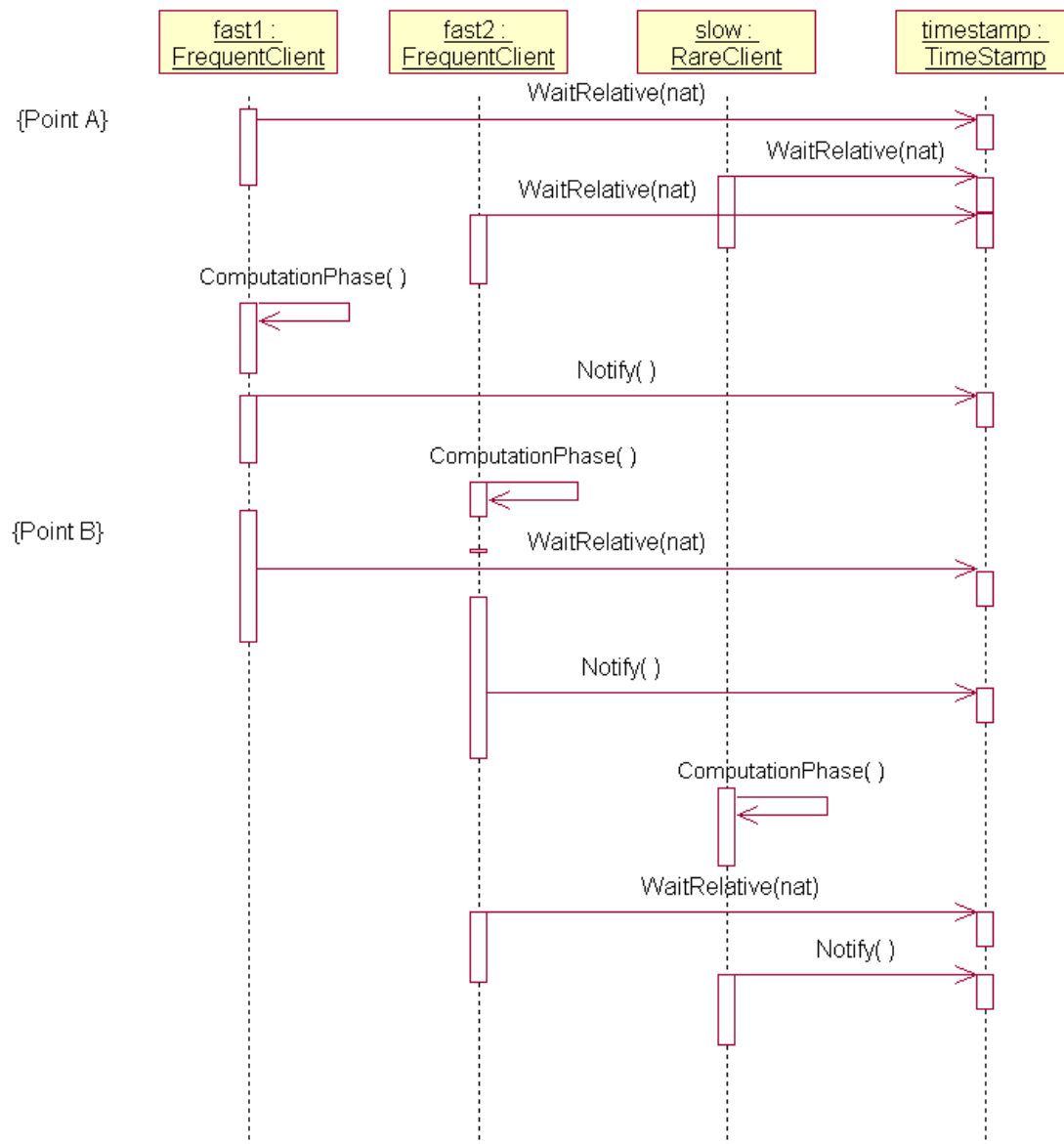


Figure B.4: Sequence Diagram for Time Stamp Pattern

```
class TimeStamp is subclass of WaitNotify

instance variables

  currentTime : nat := 0;
  wakeUpMap : map nat to nat := {|->};
```

A client may request a relative or an absolute wait. The former is performed using `WaitRelative`.

```
operations

  public WaitRelative : nat ==> ()
  WaitRelative(val) ==
    ( AddToWakeUpMap(threadid, currentTime + val);
      WaitNotify'Wait();
    );
```

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);
      WaitNotify'Wait();
    );
```

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

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

The following notify operations override those in the `WaitNotify` class. A specific thread can be immediately awoken using `NotifyThread`.

```
  public NotifyThread : nat ==> ()
  NotifyThread(tId) ==
    ( wakeUpMap := {tId} <-: wakeUpMap;
      WaitNotify'NotifyThread(tId)
    );
```

An arbitrary thread is immediately awoken using `Notify`.

```
  public Notify : () ==> ()
  Notify() ==
    let tId in set dom wakeUpMap in
    NotifyThread(tId);
```



All waiting threads are immediately awoken using `NotifyAll`.

```
public NotifyAll : () ==> ()
NotifyAll() ==
( wakeUpMap := {|->};
  WaitNotify'NotifyAll()
);
```

The operation `NotifyAndIncTime` is used to increment time and awake all threads waiting for the current time step.

```
NotifyAndIncTime : () ==> ()
NotifyAndIncTime() ==
( currentTime := currentTime + 1;
  for all t in set dom (wakeUpMap :> {currentTime}) do
    NotifyThread(t)
  );
```

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

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

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

```
sync

  mutex(AddToWakeUpMap, Notify, NotifyThread, NotifyAll);
```

The time stamp thread periodically calls `NotifyAndIncTime`. The period of the thread depends on the desired value of one time step.

```
thread

  periodic (1000)(NotifyAndIncTime)

end TimeStamp
```



## Appendix C

# Examples In Full

### C.1 VDM-SL Model for Counter Measures System

```
types
  MissileInputs = seq of MissileType;

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

```
Output = seq of OutputStep;

OutputStep = FlareType * AbsTime;

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
```

functions

```
CounterMeasures: MissileInputs -> Output
CounterMeasures(missileInputs) ==
  CM(missileInputs, [], <None>, 0);
```

```
CM: MissileInputs * Output * [MissileType] * nat -> Output
CM( missileInputs, outputSoFar, lastMissile, curTime) ==
  if missileInputs = []
  then outputSoFar
  else let curMis = hd missileInputs
       in
         if missilePriority(curMis) > missilePriority(lastMissile)
         then let newOutput = InterruptPlan(curTime, outputSoFar,
                                              responseDB(curMis))
              in CM(tl missileInputs, newOutput, curMis,
                   curTime + stepLength)
         else CM(tl missileInputs, outputSoFar, lastMissile,
                 curTime + stepLength);
```

```
InterruptPlan: nat * Output * Plan -> Output
InterruptPlan(curTime, expOutput, plan) ==
  LeavePrefixUnchanged(expOutput, curTime) ^
  MakeOutputFromPlan(curTime, plan);
```

```
LeavePrefixUnchanged: Output * nat -> Output
LeavePrefixUnchanged(expOutput, curTime) ==
  [expOutput(i) | i in set inds expOutput
   & let mk_(-,t) = expOutput(i) in t <= curTime]
```

```

MakeOutputFromResponse : nat * seq of Response -> Output
MakeOutputFromResponse(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 -> Output
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];

```

## C.2 Sequential VDM++ Model for Counter Measures System

```
class Sensor
types
public MissileType = <MissileA> | <MissileB> | <MissileC> | <None>;
```

```
instance variables

io          : SensorIO          := new SensorIO().Init();
missileValue : [MissileType|<Consumed>] := io.readMissileValue();
```

```
operations

public SetMissileValue : () ==> ()
SetMissileValue() ==
    missileValue := io.readMissileValue();
```

```
public ReadMissileValue : () ==> [MissileType]
ReadMissileValue() ==
    return missileValue;
```

```
public IsFinished : () ==> bool
IsFinished() == return missileValue = nil;
```

```
public GetMissileValue : () ==> [MissileType]
GetMissileValue() ==
    let orgMissileValue = missileValue in
    (if missileValue <> nil
     then missileValue := <Consumed>;
     return orgMissileValue);
```

```
end Sensor
```

```
class MissileDetector
```

```
instance variables
```

```
sensorRef      : Sensor;
flareControlRef : FlareController;
missileValue    : [Sensor'MissileType] := <None>;
timerRef       : Timer
```

```
operations
```

```
public Init : Sensor * FlareController * Timer ==> ()
Init (newSensor, newFlareController, newTimer) ==
  (sensorRef := newSensor;
   flareControlRef := newFlareController;
   timerRef := newTimer
  );
```

```
public Step : () ==> ()
Step() ==
  let newMissileValue = sensorRef.GetMissileValue() in
  Update(newMissileValue);
```

```
Update : [Sensor'MissileType] ==> ()
Update(newMissileType) ==
  (if newMissileType <> <None>
   then
     (missileValue := newMissileType;
      flareControlRef.MissileIsHere(missileValue);
      if missileValue <> nil
      then timerRef.Interrupt();
     )
  );
```

```
public IsFinished : () ==> bool
IsFinished() == return missileValue = nil;

end MissileDetector
```

```
class FlareController
```

```
types
  Plan = seq of PlanStep;
  public PlanStep = FlareType * nat;
  public FlareType = <FlareOneA> | <FlareTwoA> | <FlareOneB> |
                    <FlareTwoB> | <FlareOneC> | <FlareTwoC> |
                    <DoNothingA> | <DoNothingB> | <DoNothingC>;
```

```
instance variables
  missileDetectorRef : MissileDetector;
  timerRef           : Timer;
  currentMissileValue : [Sensor'MissileType] := <None>;
  currentStep        : nat                  := 0;
  fresh              : bool                 := false;
  latestMissileValue : Sensor'MissileType   := <None>;
  outputSequence     : seq of (FlareType * nat) := [];
  numberOfFreshValues : nat                 := 0;
  noMoreMissiles     : bool                 := false;
```

```
values
  responseDB : map Sensor'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 Sensor'MissileType to nat
    = {<MissileA> |-> 1,
      <MissileB> |-> 2,
      <MissileC> |-> 3,
      <None> |-> 0}
```

operations

```
public Init : MissileDetector * Timer ==> ()
Init(initMissileDetector, initTimerRef) ==
  (missileDetectorRef := initMissileDetector;
   timerRef := initTimerRef;
  );
```

```
public Step : () ==> ()
Step() ==
  ( if timerRef.CheckAwake()
    then (
      StepAlgorithm();
      if currentMissileValue = nil
      then noMoreMissiles := true
      elseif currentMissileValue <> <None>
      then let mk_(-, delay_val) =
            responseDB(currentMissileValue)(currentStep-1)
            in timerRef.Alarm(delay_val)
      )
  );
```



```

StepAlgorithm : () ==> ()
StepAlgorithm() ==
  (if fresh
    then (
      fresh := false;
      CheckFreshData();
    );
    StepPlan()
  );

```

operations

```

CheckFreshData : () ==> ()
CheckFreshData() ==
  (if HigherPriority(latestMissileValue,
                    currentMissileValue)
    then StartPlan(latestMissileValue);
    latestMissileValue := <None>;
    numberOfFreshValues := numberOfFreshValues + 1;
  );

HigherPriority : Sensor'MissileType *
               Sensor'MissileType ==> bool
HigherPriority(latest, current) ==
  return missilePriority(latest) > missilePriority(current);

```

```

StartPlan : Sensor'MissileType ==> ()
StartPlan(newMissileValue) ==
  (currentMissileValue := newMissileValue;
   currentStep := 1
  );

```

```

ReleaseAFlare : FlareType ==> ()
ReleaseAFlare(ft) ==
  outputSequence := outputSequence ^ [mk_(ft, timerRef.GetTime())];

```

```

StepPlan : () ==> ()
StepPlan() ==
  if currentStep <= len responseDB(currentMissileValue)
  then
    (let mk_(flare, -) = responseDB(currentMissileValue)(currentStep)
     in ReleaseAFlare(flare);
     currentStep := currentStep + 1
    )
  else (currentMissileValue := <None>;
        currentStep := 0
       );

```

```
public IsFinished : () ==> bool
IsFinished() == return currentStep = 0 and noMoreMissiles;
```

```
public GetResult : () ==> seq of (FlareType * nat)
GetResult() == return outputSequence;
```

```
public MissileIsHere : [Sensor'MissileType] ==> ()
MissileIsHere(newMissileValue) ==
  ( if newMissileValue not in set {<None>, nil}
    then fresh := true;
    if newMissileValue = nil
    then noMoreMissiles := true
    else latestMissileValue := newMissileValue;
  );

end FlareController
```

```
class Timer
```

```
instance variables
```

```
currentTime : nat := 0;
currentAlarm : [nat] := nil;
```

```
values
```

```
stepLength : nat = 100;
```

```
operations
```

```
public Alarm : nat ==> ()
Alarm(n) ==
  SetAlarm(n);
```

```
public CheckAwake : () ==> bool
CheckAwake() ==
  return currentAlarm = nil or
    currentAlarm <= currentTime;
```

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

```

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

```

```

SetAlarm : nat ==> ()
SetAlarm(n) ==
  currentAlarm := currentTime + n;

public Interrupt : () ==> ()
Interrupt() == currentAlarm := nil;

end Timer

```

```

class World

instance variables
  sensor : Sensor := new Sensor();
  detector : MissileDetector := new MissileDetector();
  flareControl : FlareController := new FlareController();
  timerRef : Timer := new Timer();
  inputVals : seq of ([Sensor'MissileType] * nat) := [];

operations
public Run : () ==> (seq of (FlareController'FlareType * nat)) *
  (seq of ([Sensor'MissileType] * nat))
Run() ==
  (detector.Init(sensor,flareControl,timerRef);
   flareControl.Init(detector,timerRef);

   while not (sensor.IsFinished() and detector.IsFinished() and
               flareControl.IsFinished()) do
     ( inputVals := inputVals ^
       [mk_(sensor.ReadMissileValue(), timerRef.GetTime())];
       if not detector.IsFinished() then detector.Step();
       if not flareControl.IsFinished() then flareControl.Step();
       timerRef.StepTime();
       if not sensor.IsFinished() then sensor.SetMissileValue();
       );
       return mk_(flareControl.GetResult(), inputVals)
     )
  )

end World

```

```
class IO

-- IFAD VDMTools STANDARD LIBRARY: INPUT/OUTPUT
-- -----
--
-- Standard library for the VDMTools 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 (IFAD VDM-SL) or class (VDM++). Pre/post conditions is
-- fully user customisable.
--
-- Note VDM++: Polymorphic functions are protected. In order to call
-- these functions you should:
-- 1: Either make the standard library class called IO superclass of all
--    classes, or
-- 2: Make a subclass of the standard library class and define
--    an operation that calls the polymorphic function. This operation
--    should then be called elsewhere in the specification.
--
-- 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
end IO
```

```
class SensorIO is subclass of IO
```

```
instance variables
```

```
  curIndex : nat := 0;
  mvList : seq of Sensor'MissileType := [];
```

```
operations
```

```
public Init : () ==> SensorIO
Init() ==
( let mk_(-,list) = freadval[seq1 of Sensor'MissileType]("scenario.txt")
  in
    mvList := list;
    curIndex := 1;
    return self
);
```

```
public readMissileValue : () ==> [Sensor'MissileType]
readMissileValue() ==
  if curIndex <= len mvList
  then (curIndex := curIndex + 1;
        return mvList(curIndex-1))
  else return nil;

end SensorIO
```

## C.3 Concurrent VDM++ Model for Counter Measures System

```
class Sensor

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

```
instance variables
  io          : SensorIO          := new SensorIO().Init();
  missileValue : [MissileType|<Consumed>] := io.readMissileValue();
  timerRef     : Timer;
```

```
thread
  (while missileValue <> nil do
    ( MissileValueConsumed();
      SetMissileValue();
      timerRef.StepTime();
      timerRef.Finished()
    )
  )
```

```
operations

MissileValueConsumed : () ==> ()
MissileValueConsumed() ==
  skip

sync

per MissileValueConsumed => missileValue = <Consumed>;
```

```
operations

public Init : Timer ==> ()
Init(newTimer) ==
  timerRef := newTimer;
```

```
SetMissileValue : () ==> ()
SetMissileValue() ==
  missileValue := io.readMissileValue();

sync

mutex(SetMissileValue, GetMissileValue);
```

```

operations

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

sync

per IsFinished => missileValue = nil;

```

```

operations

public GetMissileValue : () ==> [MissileType]
GetMissileValue() ==
  let orgMissileValue = missileValue in
    (if missileValue <> nil
     then missileValue := <Consumed>;
     return orgMissileValue);

sync

per GetMissileValue => missileValue <> <Consumed>;

end Sensor

```

```

class MissileDetector
instance variables

sensorRef      : Sensor;
flareControlRef : FlareController;
missileValue    : [Sensor'MissileType] := <None>;
timerRef       : Timer

```

```

thread

  while missileValue <> nil do
    let newMissileValue = sensorRef.GetMissileValue() in
      Update(newMissileValue);

```

```

operations

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

sync

per IsFinished => missileValue = nil;

```

operations

```
Update : [Sensor'MissileType] ==> ()
Update(newMissileValue) ==
  (if newMissileValue <> <None>
  then
    (missileValue := newMissileValue;
     flareControlRef.MissileIsHere(missileValue);
     if missileValue <> nil
     then timerRef.Interrupt();
    )
  );
```

```
public Init : Sensor * FlareController * Timer ==> ()
Init (newSensor, newFlareController, newTimer) ==
  (sensorRef := newSensor;
   flareControlRef := newFlareController;
   timerRef := newTimer
  );

end MissileDetector
```

class FlareController

types

```
Plan = seq of PlanStep;
public PlanStep = FlareType * nat;
public FlareType = <FlareOneA> | <FlareTwoA> | <FlareOneB> |
                  <FlareTwoB> | <FlareOneC> | <FlareTwoC> |
                  <DoNothingA> | <DoNothingB> | <DoNothingC>;
```

instance variables

```
missileDetectorRef : MissileDetector;
timerRef           : Timer;
currentMissileValue : [Sensor'MissileType] := <None>;
currentStep        : nat := 0;
fresh              : bool := false;
latestMissileValue : Sensor'MissileType := <None>;
outputSequence     : seq of (FlareType * nat) := [];
noMoreMissiles     : bool := false;
```



```

values
  responseDB : map Sensor'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 Sensor'MissileType to nat
    = {<MissileA> |-> 1,
      <MissileB> |-> 2,
      <MissileC> |-> 3,
      <None> |-> 0}

```

```

thread
  while true do
    ( StepAlgorithm();
      if currentMissileValue = nil
      then noMoreMissiles := true
      elseif currentMissileValue <> <None>
      then let mk_(-, delay_val) =
           responseDB(currentMissileValue)(currentStep-1)
           in timerRef.Alarm(delay_val);
    )

```

```

operations

StepAlgorithm : () ==> ()
StepAlgorithm() ==
  (if fresh
   then (
     fresh := false;
     CheckFreshData();
   );
   StepPlan()
  );

sync

per StepAlgorithm => fresh = true or currentMissileValue <> <None>;

```

```
operations

public IsFinished : () ==> seq of (FlareType * nat)
IsFinished() == return outputSequence;

sync
  per IsFinished => currentStep = 0 and noMoreMissiles;
```

```
operations

public MissileIsHere : [Sensor'MissileType] ==> ()
MissileIsHere(newMissileValue) ==
  ( if newMissileValue not in set {<None>, nil}
    then fresh := true;
    if newMissileValue = nil
    then noMoreMissiles := true
    else latestMissileValue := newMissileValue;
  );

sync

  mutex(MissileIsHere, CheckFreshData);
```

```
operations

public Init : MissileDetector * Timer ==> ()
Init(initMissileDetector, initTimerRef) ==
  (missileDetectorRef := initMissileDetector;
   timerRef := initTimerRef;
  );
```

```
CheckFreshData : () ==> ()
CheckFreshData() ==
  (if HigherPriority(latestMissileValue,
                    currentMissileValue)
   then StartPlan(latestMissileValue);
   latestMissileValue := <None>;
  );
```

```

HigherPriority : Sensor'MissileType *
                Sensor'MissileType ==> bool
HigherPriority(latest, current) ==
    return missilePriority(latest) > missilePriority(current);

StartPlan : Sensor'MissileType ==> ()
StartPlan(newMissileValue) ==
    (currentMissileValue := newMissileValue;
     currentStep := 1
    );

```

```

ReleaseAFlare : FlareType ==> ()
ReleaseAFlare(ps) ==
    outputSequence := outputSequence ^ [mk_(ps, timerRef.GetTime())];

```

```

StepPlan : () ==> ()
StepPlan() ==
    if currentStep <= len responseDB(currentMissileValue)
    then
        (let mk_(flare, -) = responseDB(currentMissileValue)(currentStep)
         in ReleaseAFlare(flare);
         currentStep := currentStep + 1
        )
    else (currentMissileValue := <None>;
          currentStep := 0
         );

end FlareController

```

```

class Timer

instance variables

    currentTime : nat := 0;
    currentAlarm : [nat] := nil;
    finished : bool := false;

```

```

operations

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

```

```

operations

public Alarm : nat ==> ()
Alarm(n) ==
    ( SetAlarm(n);
      WakeUp());

```

```
WakeUp : () ==> ()
WakeUp() ==
  if currentAlarm <> nil
  then if currentTime < currentAlarm
        then currentTime := currentAlarm;
  sync

  per WakeUp => finished or
    currentAlarm = nil or
    currentAlarm <= currentTime;
  mutex(WakeUp, StepTime, GetTime);
```

```
operations

public Interrupt : () ==> ()
Interrupt() ==
  currentAlarm := nil;

sync

  mutex(SetAlarm, Interrupt);
```

```
operations

SetAlarm : nat ==> ()
SetAlarm(n) ==
  currentAlarm := currentTime + n;
```

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

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

values

  stepLength : nat = 100;

end Timer
```

```
class World

instance variables

    sensor : Sensor := new Sensor();
    detector : MissileDetector := new MissileDetector();
    flareControl : FlareController := new FlareController();
    timerRef : Timer := new Timer();

operations

public Run : () ==> seq of (FlareController'FlareType * nat)
Run() ==
( sensor.Init(timerRef);
  detector.Init(sensor,flareControl,timerRef);
  flareControl.Init(detector,timerRef);
  startlist({sensor, detector, flareControl});
  sensor.IsFinished();
  detector.IsFinished();
  return flareControl.IsFinished()
)

end World
```

## C.4 Real-Time Concurrent VDM++ Model for Counter Measures System

```
class Sensor
types

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

instance variables

io          : SensorIO          := new SensorIO().Init();
missileValue : [MissileType|<Consumed>] := io.readMissileValue();
timer       : Timer             := new Timer();
```

```
values

public stepLength : nat = 100;
```

```
thread

duration(0)
( start(timer);
  while missileValue <> nil do
    ( SkipNum(stepLength);
      SetMissileValue();
    )
  )
)
```

```
operations

SkipNum : nat ==> ()
SkipNum(n) ==
(
  timer.Alarm(n);
);
```

```

operations

public GetMissileValue : () ==> [MissileType]
GetMissileValue() ==
  duration(0)
  let orgMissileValue = missileValue in
    (if missileValue <> nil
      then missileValue := <Consumed>;
      return orgMissileValue);

sync

per GetMissileValue => missileValue not in set {<Consumed>, <None>};

```

```

operations

SetMissileValue : () ==> ()
SetMissileValue() ==
  missileValue := io.readMissileValue();

```

```

sync

mutex(SetMissileValue, GetMissileValue);
operations

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

sync

per IsFinished => missileValue = nil;

end Sensor

```

```

class MissileDetector

operations

Update : [Sensor'MissileType] ==> ()
Update(newMissileValue) ==
  if newMissileValue <> <None>
  then
    (missileValue := newMissileValue;
     flareControlRef.MissileIsHere(missileValue);
     timerRef.Interruption();
    );

```

```
instance variables
```

```
sensorRef      : Sensor;  
flareControlRef : FlareController;  
missileValue    : [Sensor'MissileType] := <None>;  
timerRef       : Timer;
```

```
operations
```

```
public Init : Sensor * FlareController * Timer ==> ()  
Init (newSensor, newFlareController, newTimer) ==  
  (sensorRef := newSensor;  
   flareControlRef := newFlareController;  
   timerRef := newTimer);
```

```
thread
```

```
while missileValue <> nil do  
  let newMissileValue = sensorRef.GetMissileValue() in  
  Update(newMissileValue);
```

```
operations
```

```
public IsFinished : () ==> ()  
IsFinished() == skip;  
  
sync  
per IsFinished => missileValue = nil;  
  
end MissileDetector
```

```
class FlareController
```

```
types
```

```
Plan = seq of PlanStep;  
PlanStep = FlareType * nat;  
public FlareType = <FlareOneA> | <FlareTwoA> | <FlareOneB> |  
                  <FlareTwoB> | <FlareOneC> | <FlareTwoC> |  
                  <DoNothingA> | <DoNothingB> | <DoNothingC>;
```



values

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

instance variables

```
missileDetectorRef : MissileDetector;
timerRef           : Timer;
currentMissileValue : [Sensor'MissileType] := <None>;
currentStep        : nat                    := 0;
fresh              : bool                  := false;
latestMissileValue : Sensor'MissileType    := <None>;
outputSequence     : seq of FlareType      := [];
noMoreMissiles     : bool                  := false;
```

operations

```
public IsFinished : () ==> seq of FlareType
IsFinished() == return outputSequence;

sync
  per IsFinished => currentStep = 0 and noMoreMissiles;
```

```
operations
```

```
ReleaseAFlare : FlareType ==> ()
ReleaseAFlare(ps) ==
  duration(10)
  ( cases ps:
    <FlareOneA> -> ReleaseFlareOneA(),
    <FlareTwoA> -> ReleaseFlareTwoA(),
    <FlareOneB> -> ReleaseFlareOneB(),
    <FlareTwoB> -> ReleaseFlareTwoB(),
    <FlareOneC> -> ReleaseFlareOneC(),
    <FlareTwoC> -> ReleaseFlareTwoC(),
    <DoNothingA> -> ReleaseFlareDoNothingA(),
    <DoNothingB> -> ReleaseFlareDoNothingB(),
    <DoNothingC> -> ReleaseFlareDoNothingC()
  end;
  outputSequence := outputSequence ^ [ps]
);
```

```
ReleaseFlareOneA : () ==> ()
ReleaseFlareOneA() == skip;
```

```
ReleaseFlareTwoA : () ==> ()
ReleaseFlareTwoA() == skip;
```

```
ReleaseFlareOneB : () ==> ()
ReleaseFlareOneB() == skip;
```

```
ReleaseFlareTwoB : () ==> ()
ReleaseFlareTwoB() == skip;
```

```
ReleaseFlareOneC : () ==> ()
ReleaseFlareOneC() == skip;
```

```
ReleaseFlareTwoC : () ==> ()
ReleaseFlareTwoC() == skip;
```

```
ReleaseFlareDoNothingA : () ==> ()
ReleaseFlareDoNothingA() == skip;
```

```
ReleaseFlareDoNothingB : () ==> ()
ReleaseFlareDoNothingB() == skip;
```

```
ReleaseFlareDoNothingC : () ==> ()
ReleaseFlareDoNothingC() == skip;
```

```
public Init : MissileDetector * Timer ==> ()
Init(initMissileDetector, initTimer) ==
  (missileDetectorRef := initMissileDetector;
   timerRef := initTimer);
```

```
CheckFreshData : () ==> ()
CheckFreshData() ==
  (if HigherPriority(latestMissileValue,
                    currentMissileValue)
   then StartPlan(latestMissileValue);
   latestMissileValue := <None>;
  );
```

```
StepAlgorithm : () ==> ()
StepAlgorithm() ==
  (if fresh
   then (
     fresh := false;
     CheckFreshData();
   );
   StepPlan()
  );

sync

per StepAlgorithm => fresh = true or currentMissileValue <> <None>;
```

```
thread
  while true do
    ( StepAlgorithm();
      if currentMissileValue = nil
      then noMoreMissiles := true
      elseif currentMissileValue <> <None>
      then let mk_(-, delay_val) =
           responseDB(currentMissileValue)(currentStep-1)
           in timerRef.Alarm(delay_val))
```

operations

```
HigherPriority : Sensor'MissileType *
               Sensor'MissileType ==> bool
HigherPriority(latest, current) ==
  return missilePriority(latest) > missilePriority(current);
```

```
StartPlan : Sensor'MissileType ==> ()
StartPlan(newMissileValue) ==
  (currentMissileValue := newMissileValue;
   currentStep := 1
  );
```

```
StepPlan : () ==> ()
StepPlan() ==
  if currentStep <= len responseDB(currentMissileValue)
  then
    (let mk_(flare, -) = responseDB(currentMissileValue)(currentStep)
     in ReleaseAFlare(flare);
     currentStep := currentStep + 1
    )
  else (currentMissileValue := <None>;
        currentStep := 0
       );
```

```
public MissileIsHere : [Sensor'MissileType] ==> ()
MissileIsHere(newMissileValue) ==
  ( if newMissileValue not in set {<None>, nil}
    then fresh := true;
    if newMissileValue = nil
    then noMoreMissiles := true
    else latestMissileValue := newMissileValue;
  );

sync

  mutex(MissileIsHere, CheckFreshData);

end FlareController
```

```
class Timer

instance variables
  currentTime : nat := 0;
  active : bool := false;
  currentAlarm : nat := 0;

thread
  periodic(100)(IncTime)
```

```
operations
```

```
IncTime: () ==> ()
IncTime() ==
  duration(0)
  if active
  then currentTime := currentTime + Sensor'stepLength;
```

```
public Alarm : nat ==> ()
Alarm(wakeup) ==
  ( currentAlarm := wakeup;
    Start();
    WaitAlarm());
```

```
WaitAlarm : () ==> ()
WaitAlarm() == Stop();

sync
  per WaitAlarm => currentTime >= currentAlarm;
```

```
operations
```

```
public Interruption : () ==> ()
Interruption() ==
  currentTime := currentAlarm;
```

```
Start : () ==> ()
Start() ==
  ( active := true;
    currentTime := 0);
```

```
Stop : () ==> ()
Stop() ==
  active := false;
```

```
sync
```

```
  mutex(Start, Interruption, IncTime);
  mutex(Start, Stop);
end Timer
```

```
class World

instance variables

    sensor : Sensor := new Sensor();
    detector : MissileDetector := new MissileDetector();
    flareControl : FlareController := new FlareController();
    timerRef : Timer := new Timer();

operations

public Run : () ==> seq of FlareController'FlareType
Run() ==
duration(0)
( start(timer);
  detector.Init(sensor,flareControl,timer);
  flareControl.Init(detector,timer);
  startlist({sensor, detector, flareControl});
  sensor.IsFinished();
  detector.IsFinished();
  return flareControl.IsFinished()
)

end World
```

## C.5 The Wait-Notify Mechanism

```
class WaitNotify

instance variables
  waitset : set of nat := {}

operations

  public Wait : () ==> ()
  Wait() ==
    ( AddToWaitSet (threadid);
      Awake()
    );

  public Notify : () ==> ()
  Notify() ==
    let p in set waitset in
      waitset := waitset \ {p};

  private AddToWaitSet :
    nat ==> ()
  AddToWaitSet(n) ==
    waitset := waitset
      union {n};

  private Awake : () ==> ()
  Awake() == skip

sync
  per Awake => threadid not
    in set waitset;
  mutex(AddToWaitSet)

end WaitNotify
```