Exploring Timing Properties Using VDM++*

Paul Mukherjee[1], Fabien Bousquet[2], Jérôme Delabre[2], Stephen Paynter[3], and Peter Gorm Larsen[1]

[1] IFAD,Forskerparken 10, DK-5230,Odense M, Denmark
[2] Matra BAe Dynamics France, 20-22 rue Grange Dame Rose, 78141 Velizy-Villacoublay Cedex, France
[3] Matra BAe Dynamics (UK) Ltd, FPC 450, P.O. Box 5, Filton, Bristol, BS34 7QW

Abstract. A new and unique method of timing property exploration based on the formal specification notation VDM++ is presented. It is explained how the VDM++ notation and tool support has been adapted to enable a pragmatic approach to detect potential timing bottlenecks with a software design before expensive commitment to an efficient implementation is made. The approach is explained both informally and semantically, and an example is presented.

1 Introduction

A key problem when developing software for real-time systems is judging whether a design has the real-time behaviour required by the overall system. That is, judging whether the design will be able to meet the physical deadlines imposed by the system and its environment. In practice it is often only possible to make this judgement once an implementation is available: if during testing the implementation fails to meet any deadlines, then the design must be revisited and implementation repeated. Thus, *exploration* of timing properties occurs late in the development process.

This paper describes an approach intended to allow exploration of timing properties at an earlier stage in the development process than implementation. The approach described extends the existing pragmatic approach to formal modelling advocated by IFAD [2], in which formal models are validated by execution. The extension described in this paper allows accumulation of information concerning real-time behaviour during execution of models. This information may then be analyzed separately to allow analysis of the cumulative real-time behaviour of the model. The approach may be customized according to the execution environment (processor, real-time kernel) intended for use by the actual system.

This paper is organized as follows: in the next section the notation VDM++ and its supporting tools are introduced. After that the details of the approach to timing property exploration are presented. In Section 4 an example is given,

* The approach described in this paper was developed as part of the ESPRIT project number 27618 "VICE: VDM++ In a Constrained Environment".

illustrating the approach. Following this a comparison to other approaches is given, and finally some conclusions are presented.

Note that when talking in general about quantities of time, the term *time unit* is used. However for specific examples, units of time relevant to the example are used (e.g. milliseconds). Throughout the text, portions of formal specification are presented with grey background to distinguish from the main body of the text.

2 The Formal Notation VDM++

The formal notation VDM++ is an object-oriented, model-based specification language, and is largely a superset of the ISO standardized notation VDM-SL [11]. VDM++ was originally developed in the ESPRIT project called AFRODITE [1] and subsequently improved by IFAD. This notation is supported by the IFAD VDM++ Toolbox [6]. It provides a precise, unambiguous basis for analysis of requirements and allows early validation through testing and debugging. In this way it is possible to bring testing activities forward to the specification phase of the development life-cycle.

2.1 The VDM++ Notation

In VDM++ a complete formal specification consists of a collection of class specifications. A class specification has the following components:

Class header: This contains the class name declaration and inheritance information (single or multiple).

Types: Definitions of any types used in the class.

Values: Definitions of constant values.

Instance variables: The state of an object consists of variables which can be of simple types, VDM-SL types such as sets, sequences and maps, and object references (the clientship relation). Instance variables can have invariant and initial expressions.

Operations: Class methods that may be defined implicitly, explicitly (through imperative statements), or as a mixture of both. The implicit style uses pre and post condition expressions in the VDM-SL syntax.

Functions: Functions are similar to operations except that the body of a function is an expression rather than a statement. Also, functions are not allowed to refer to instance variables.

Synchronization: Operation invocation is defined with the Rendez-Vous semantics. It is possible to specify the circumstances in which an operation may be executed using a *permission predicate* for the operation. This predicate is over the instance variables of the object.

Thread: In VDM++ active objects are considered to model active world entities. An object can be made active by the specification of a thread. A thread is a sequence of statements which are executed to completion, at which point the thread dies.

2.2 The IFAD VDM++ Toolbox

The IFAD VDM++ Toolbox is a comprehensive suite of tools for the analysis and validation of formal models described in VDM++. Currently the following features are supported: syntax and type checking of models; execution of models using an integrated symbolic interpreter; debugging of models using breakpoints and stepping through the model; execution of thread-based models; automatic generation of UML models in Rational Rose from VDM++ models, and vice versa, as well as merging of such heterogeneous UML/VDM++ models; automatic generation of code from models, into C++ or Java; pretty-printing using Microsoft Word or LaTeX; output of formatted document, with colouring based on test coverage, incorporation of tables containing percentage coverage; CORBA-compliant API allowing interaction with tools from other applications.

2.3 Extensions to the VDM++ Technology for Concurrent Real-Time Systems

As well as the extension described in Section 3, a few additions to the VDM++ notation have been implemented: a *threadid* expression, denoting the unique identifier number of the thread currently executing; a *mutex* directive for use in permission predicates; the introduction of periodic threads i.e. threads that perform some operation with fixed frequency; introduction of access modifiers (public, private and protected) for class members.

All of these extensions to the VDM++ notation are supported by the VDM++ Toolbox. Note that in the case of periodic threads, a notion of time is required, and thus the extensions described in Section 3 are a prerequisite.

3 Timing Approach

In this section we describe the approach to timing analysis supported using the VDM++ Technology.

3.1 Objectives

The main objective of the approach described here is to get early feedback on the suitability of a particular dynamic architecture. Here the term dynamic architecture refers to the mapping of computations to threads. Suitability refers to the absence of the following properties:

- Periodic threads missing deadlines;
- Deadlocks in the model;
- Missed system deadlines (hard or soft).

(Note the distinction here between missed system deadlines and periodic threads missing their deadlines: the former is an externally visible property, whereas the latter is an internal property of the design, which may or may not lead to missed system deadlines.)

The approach is pragmatic because the objective is to provide feedback rather than formal guarantees of the kind usually obtained by formal verification. In particular, it is desired that timing-related problems be identified during the specification and design phases, rather than during target integration testing, where such problems are currently found.

Note that currently the approach assumes a single processor target. Therefore for the remainder of this paper, it is assumed that the target consists of a single processor (referred to as the target processor) controlled by a real-time kernel (referred to as the target kernel). However there is no reason in principal why the approach could not be extended to multiprocessor targets.

3.2 Overview of Approach

The basic idea of the approach is to simulate the timing behaviour of the target processor within the IFAD VDM++ Toolbox interpreter. To achieve this the interpreter maintains an internal variable which corresponds to the clock of the target processor i.e. the clock of the target processor is simulated. The interpreter adopts the same scheduling algorithm as that used by the target kernel. During execution of the model a number of events will occur: swapping in and out of threads; and operation requests, activations and completions. We call such events, trace events. For the purposes of this paper we restrict our interest to the swapping in and out of threads.

Each trace event is logged in a trace file, with the *simulated time* at which the event occurred. The simulated time is the reading of the clock on the target processor as recorded by the interpreter when the event occurred. The simulated time is incremented during execution and thread swapping by the interpreter. There are three ways in which this is performed:

- Selected portions of the model may be enclosed within *duration statements* (described in detail in Section 3.5). These give the execution duration in simulated time for that portion of the model. This duration is used to increment simulated time.
- Worst-case analysis is used for portions of the model that are not covered by duration statements. This is calculated specifically for the target processor, using the default duration information. The default duration information is a file containing the execution duration in simulated time for elementary assembly instructions on the target processor. The values yielded by this worst-case analysis are then used to increment simulated time.
- When a thread is switched out and another thread is switched in, simulated time is incremented by the task switching overhead. This is a user-definable value, allowing simulation of the time taken to switch tasks in the target kernel.

An overview of the approach is given in Figure 1.

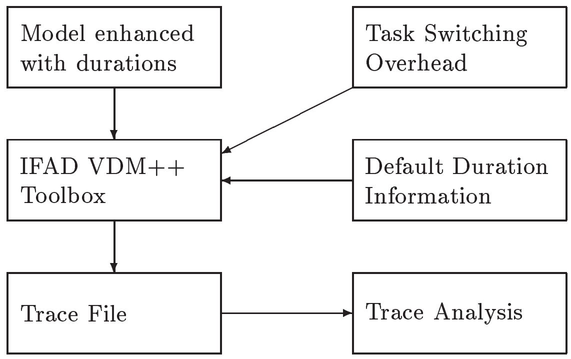


Fig. 1. Overview of Approach

3.3 Semantics

In this section we give a semantic description of the basic timing approach (i.e. we do not consider duration statements here). The semantics presented are at a high level of abstraction, and are presented in VDM-SL. The presentation is described independent of the scheduling algorithm used, so information relevant for scheduling (e.g. thread priority) is not needed here.

At an abstract level we can think of a thread as a sequence of statements:

```
Thread = seq of TimedStmt;
```

A timed statement consists of a constituent statement, and its execution duration (in simulated time on the target processor).

```
TimedStmt :: stmt : Stmt
             dur  : nat;
```

The execution duration is calculated statically using worst-case analysis on the text of the statement. The issue of the coarseness of this calculation is addressed in Section 3.5.

For our current purposes the only property we require of statements is being able to distinguish them. Thus:

```
Stmt = token;
```

The objective of a semantic description is to capture the possible traces that might occur when several threads are executed concurrently on a single processor. It is sufficient to consider the case in which two threads are executed concurrently; the general case follows from this case. Note that it is not possible to statically ascertain the *specific* trace that would be generated.

A trace is a sequence of trace elements; a trace element represents an uninterrupted execution on the processor, with the times at which execution began and ended:

```
Trace = seq1 of TraceElement
inv tr == forall i, j in set inds tr &
            i < j => tr(i).switchedOut < tr(j).switchedIn;

TraceElement :: stmts       : seq1 of Stmt
                switchedIn  : nat
                switchedOut : nat
inv mk_TraceElement(-, si, so) == si <= so;
```

The semantics of a particular pairwise interleaving is then given by the function `Interleave`. This takes a pair of threads and the task switching overhead, and returns the set of possible traces for this pair.

This function works recursively:

If either thread has no statements left to execute the other thread is executed continuously to make exactly one trace;

Otherwise the first statement is taken from each thread, and the interleavings for the remaining parts of the threads are recursively calculated. The first statement is then added to each trace so generated, with times offset accordingly.

```
Interleave : Thread * Thread * nat -> set of Trace
Interleave(t1, t2, ts) ==
  if t1 = [] then MakeOneTrace(t2)
  elseif t2 = [] then MakeOneTrace(t1)
  else let i1 = Interleave(tl t1, t2, ts),
           i2 = Interleave(t1, tl t2, ts) in
       let i1_start_t1 = { t | t in set i1 & FromThread(t1, t(1)) },
           i1_start_t2 = { t | t in set i1 & FromThread(t2, t(1)) },
           i2_start_t1 = { t | t in set i2 & FromThread(t1, t(1)) },
           i2_start_t2 = { t | t in set i2 & FromThread(t2, t(1)) } in
       AddPrefixToTraceSet(hd t1, i1_start_t1, ts) union
       AddPrefixToTraceSet(hd t2, i2_start_t2, ts) union
       AddOnePrefixToTraceSet(hd t1, i1_start_t2, ts) union
       AddOnePrefixToTraceSet(hd t2, i2_start_t1, ts);
```

The function `MakeOneTrace` takes a thread and constructs a trace which executes continuously on the processor:

```
MakeOneTrace : Thread -> set of Trace
MakeOneTrace(t) ==
  { [mk_TraceElement([t(i).stmt | i in set inds t], 0,
                     Sum( [t(i).dur | i in set inds t]))] };
```

Here the function `Sum` returns the sum of a sequence of natural numbers; its definition is omitted for brevity.

The predicate `FromThread` delivers true iff a particular trace event corresponds to a particular thread:

```
FromThread : Thread * TraceElement -> bool
FromThread(thread, te) ==
  forall stmt in set elems te.stmts &
    exists tstmt in set elems thread & stmt = tstmt.stmt
```

The function `AddPrefixToTraceSet` takes a timed statement `tstmt`, a set of traces `traces` and a task switching overhead, and computes all possible traces that can be made by executing `tstmt` first, and then continuing with a trace from `traces`:

```
AddPrefixToTraceSet : TimedStmt * set of Trace * nat -> set of Trace
AddPrefixToTraceSet(tstmt, traces, ts) ==
  AddOnePrefixToTraceSet(tstmt, traces, ts) union
  { AddPrefixToTrace(tstmt, trace) | trace in set traces };
```

The function `AddOnePrefixToTraceSet` adds a single trace element to all the traces in the given set.

```
AddOnePrefixToTraceSet : TimedStmt * set of Trace * nat -> set of Trace
AddOnePrefixToTraceSet(tstmt, traces, ts) ==
  let te = mk_TraceElement([tstmt.stmt], 0, tstmt.dur) in
  { [te] ^ Offset(trace, te.switchedOut + ts) | trace in set traces };
```

Here, the function `Offset` is used to perform a time shift in a trace:

```
Offset : Trace * nat -> Trace
Offset (trace, offset) ==
  [ mk_TraceElement(trace(i).stmts, trace(i).switchedIn + offset,
                    trace(i).switchedOut + offset)
  | i in set inds trace];
```

The function `AddPrefixToTrace` adds the given statement to the first trace element in the given trace:

```
AddPrefixToTrace : TimedStmt * Trace -> Trace
AddPrefixToTrace(tstmt, trace) ==
  let te = mk_TraceElement([tstmt.stmt] ^ trace(1).stmts,
                           trace(1).switchedIn,
                           trace(1).switchedOut + tstmt.dur) in
  [te] ^ Offset(tl trace, tstmt.dur);
```

Having specified all possible traces, it would then be possible to specify different scheduling algorithms as subsets of all possible traces. This is not presented here for brevity.

3.4 Example

To illustrate the approach and the semantics presented in the preceding section, consider a model in which two threads are executing concurrently:

Thread 1

```
(s1;
 s2)
```

Thread 2

```
(t1;
 t2;
 t3)
```

Here, `s1`, `s2`, `t1`,`t2` and `t3` are VDM++ statements. According to the model and the scheduling policy a number of different interleavings are possible. We consider the the interleaving shown in Table 1.

`s1;`
Thread 2 switched in
`t1;`
`t2;`
Thread 1 switched in
`s2;`
Thread 2 switched in
`t3;`

Table 1. Example Interleaving

Statement	Time
`s1`	10
`s2`	20
`t1`	30
`t2`	20
`t3`	40

Table 2. Worst-case Execution Times

For the purposes of this example, suppose that using worst case analysis based on default timing behaviour for the target architecture, the execution times shown in Table 2 are computed for the statements. Suppose further that the user specifies the task switching overhead to be 5 time units.

The trace unfolds as follows. Following execution of statement `t1`, thread 1 is switched in. It has execution time of 30, so including the task switching overhead, statement `s1` is executed at time 35. After executing `s1` thread 2 is switched in again. Now the time is 50 time units. When `t2` has been completed the time is 70. When the execution of `s2` then starts 75 time units have passed. Finally when `s2` has completed and `t3` is ready to start in thread 2 the time will be 100 and after the execution of `t3` the time will be 140 time units.

We can represent this interleaving diagrammatically:

Event time		35	50	75	100	140
Execution	`t1;` (30) →	`s1;` (10) →	`t2;` (20) →	`s2;` (20) →	`t3;` (40) →	

In this diagram, arrows represent statements executed sequentially from a single thread before an event occurs, and the figure in brackets following the statement is the calculated execution time for that statement. Note that in the trace file only the events and the times at which they occurred will be logged. Recall that an event is a switch in or out of a thread, or a request, activation or completion of an operation call.

3.5 Duration Statements

The approach described in Sections 3.2, 3.3 and 3.4 allows exploration and analysis of the timing behaviour of a VDM++ model. However the use of worst-case analysis for statically computing statement execution durations gives crude, coarse values for simulated time. Moreover in many situations, from existing knowledge and experience it is known how long a particular algorithm takes to execute on a particular processor.

To remedy this situation, to the existing statements of VDM++, a new statement has been added: the duration statement.

$$\text{duration statement} = \mathsf{duration}(\text{numeral})\ \text{statement}$$

A duration is an estimate of how much time a particular portion of a VDM model will take to execute, in the implementation, on the target processor. The information provided by a duration statement is used to override the default execution time calculated for that portion.

The duration statement offers a number of benefits: timing analysis can be made more precise; different durations for portions of a model can be experimented with interactively; and portions of the model which would not actually consume processor time (e.g. any parts of the environment which have been modelled) can be enclosed within a `duration(0)` statement.

3.6 Semantics of Durations

The essential difference introduced by durations relates to the simulated execution time of portions of the model; in Section 3.3 this time is statically calculated for each statement. With the introduction of durations, this execution time can no longer be statically ascertained (for instance, an operation call could occur within a duration statement). A fully formal semantic description is beyond the scope of this paper (though such a semantic description does exist: see Section 3.8). Instead in this section we give an informal description of the semantics of durations.

The execution time for a statement is ascertained dynamically, as follows:

If the statement is not in the scope of a duration statement
The default value for that particular target architecture, as calculated using worst case analysis using default timing behaviour for the target architecture.

If the statement is in the scope of a duration statement
If this is statement s_i from a block $s_1, \ldots, s_n$ which is bound by the duration statement duration t, then the execution time for this statement is zero if i is less than n. If i equals n the execution time is the time of the entire duration statement. That is, the time will not be incremented before the entire duration statement is completed. This is a coarse approximation if a thread is interrupted in the middle of execution of the body of a duration statement. This will only pose a problem if a pre-emptive scheduling algorithm is used; if such problems arise, finer-grained duration statements may be used.

3.7 Example with Duration Statements

Consider again the model from Section 3.4, this time endowed with duration statements in thread 2.

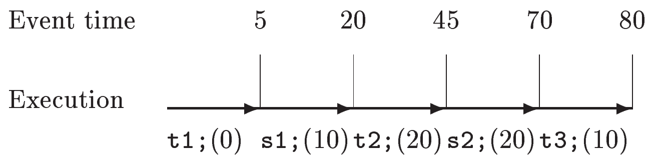
Thread 1		**Thread 2**	

	`(s1;`
	` s2)`
			`duration (20)(`
			` t1;`
			` t2);`
			`duration (10) t3`

Using the same interleaving, default times, and task switching overhead as previously the trace would evolve as follows:

Following execution of statement `t1`, thread 1 is switched in. However, since `t1` is inside the body of a duration statement the time is not yet incremented. Thus including the task switching overhead, statement `s1` is executed at time 5. After executing `s1` thread 2 is switched in again. Now the time is 20 time units. When `t2` has been completed the entire body of the duration statement in thread 2 is completed and thus the 20 time units from the duration statement is added. When the execution of `s2` then starts 45 time units has passed. Finally when `s2` has completed and `t3` is ready to start in thread 2 the time will be 70 and after the execution of `t3` the time will be 80 time units.

We can represent this interleaving diagrammatically:



3.8 Tool Support for Approach

As outlined in Section 3.2, an integral part of the approach described in this paper is the tool support. To support this approach, the IFAD VDM++ Toolbox has been extended in three ways:

- Support for duration statements within the syntax checker, type checker and interpreter;
- Generation of timed trace files by the interpreter during model execution;
- Support for a variety of real-time features within the interpreter.

The real-time features provided include the following:

- A selection of scheduling algorithms (priority-based, cooperative and time sliced amongst others).
- Customizable timing behaviour, in the sense that the static analysis used to calculate worst-case execution times, is parametrized in terms of the execution duration for standard assembly language instructions. The default values for these durations can be overridden by providing a user-defined file of these values;

– Customizable task switching overhead, allowing modelling of a variety of real-time kernels, which typically have different task switching overheads to each other;
– Time factor, allowing definition of a scalar which is used to multiply the statically calculated execution times (duration statements are unaffected by this). This allows investigation of different processor speeds etc.

Note that since the interpreter used within the IFAD VDM++ Toolbox is formally specified in VDM-SL, the dynamic semantics of all of these features have been formally specified.

3.9 Trace Analysis

The trace file generated during execution of a model is a plain text file, and is therefore straightforward to manipulate and analyze. This means that standard tools can be used to analyse runtime behaviour of the model. Two kinds of analysis may be performed: generic and specific.

Generic trace analysis allows compilation of statistics that are meaningful for all VDM++ models. This includes generating statistics about function calls (average, minimum and maximum durations for each function), statistics about processor utilization etc. Currently a collection of Perl scripts has been written which take as input a trace file and generate Microsoft Excel spreadsheets showing such statistics.

Specific trace analysis refers to analysis intrinsic to the model being executed. This might be visualization of the trace file using symbols or icons meaningful in the context of the system being modelled. An example of this is given in Section 4.3. An alternative kind of specific trace analysis is to frame requirements on traces as predicates on sequences in VDM++. These can then be executed against generated traces to check satisfaction of such requirements.

4 Example - A Counter Measures System

In this section an example application is presented illustrating the proposed approach. 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 2.

Fig. 2. Overview of Counter Measures System

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 communication.

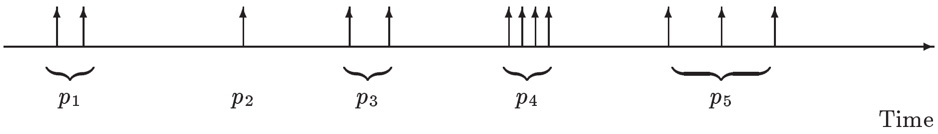


Fig. 3. Example Firing Sequence

An example firing sequence is shown in Figure 3. Flare release commands are represented by the vertical arrows. Five actions are depicted in this figure. 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.

4.1 VDM++ Model

At the top level, the counter measures application consists of a MissileDetector thread and a FlareController thread. In the model presented, there is also a Sensor thread used for simulation purposes. The MissileDetector thread takes information from the Sensor thread concerning missiles that have been sensed, and passes instructions on to the FlareController thread. The FlareController thread computes commands to be sent to flare control hardware. An overview of this arrangement can be seen in the UML class diagram shown in Figure 4.

For brevity we only describe one class specification - the FlareController class - in detail in this paper. The remaining classes are now described briefly:

SensorIO - a class used to read test data from a file. This file contains a specific scenario i.e. a specific sequence of missiles arriving at different times.

Timer - a class representing a timer which can be used to block for a period of time using its `Alarm` operation. The timer may be interrupted using its `Interrupt` operation.

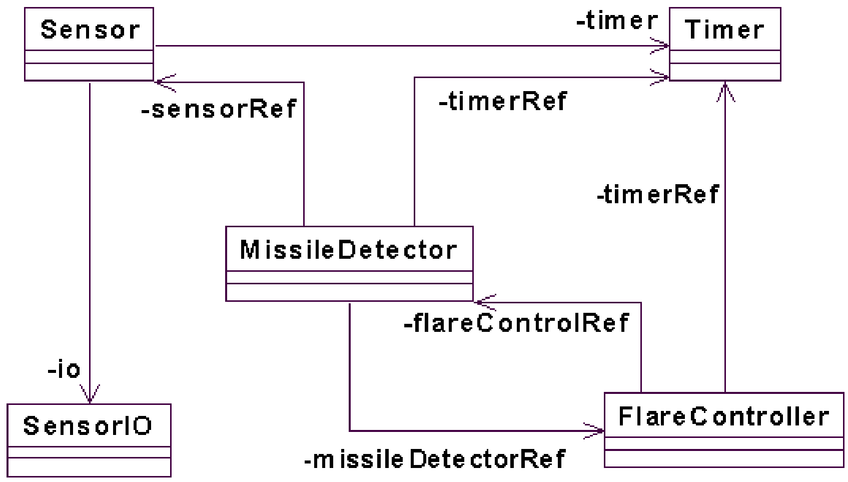


Fig. 4. Class Diagram for Counter Measures System

Sensor This contains the Sensor thread. It is used to model hardware which would sense a missile. It uses the SensorIO class to acquire values for a particular scenario. It uses an instance of the Timer class to simulate the different arrival times of missiles.

MissileDetector This class contains the MissileDetector thread. It reads values yielded by the Sensor thread, and passes these on to the FlareController thread. It shares a Timer with the FlareController thread, which it uses to interrupt the FlareController thread whenever a new missile is detected.

4.2 Specification of The Flare Controller

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

```
class FlareController
```

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. `FlareType` consists of some example flare values used for testing.

```
types

  Plan = seq of PlanStep;
  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 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;
  latestMissileValue  : Sensor`MissileType   := <None>;
  fresh               : bool                 := false;
  outputSequence      : seq of FlareType     := [];
  noMoreMissiles      : bool                 := false;
```

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

```
operations

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

Two values are defined: `responseDB` gives the `Plan` for each missile that can be detected, with values for testing purposes; 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 algorithm for the `FlareController` is described in its thread. This repeatedly does the following: 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`.

```
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))
```

`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.

```
operations

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

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

`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>);

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 `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 ReleaseAFlare(flare);
        currentStep := currentStep + 1)
  else (currentMissileValue := <None>;
        currentStep := 0);
```

The operation `ReleaseAFlare` corresponds to the physical action of releasing a flare. It is known that 10 milliseconds are required for this action, so this is specified using a duration statement.

```
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]);
```

The specific `ReleaseFlare...` operations all have the same body as `ReleaseFlareOneA` given below, and are therefore omitted for brevity. They are used purely to allow identification of the different flares released in the trace file.

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

The operation `MissileIsHere` is used by the `MissileDetector` thread to indicate detection of a new missile. Since it is executed by another thread, it may execute concurrently with operations used by the `FlareController` thread. Therefore to ensure integrity of instance variables, it is executed mutually exclusively with `CheckFreshData`.

```
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);

end FlareController
```

4.3 Execution Of The Specification

The counter measures model has been executed by the IFAD VDM++ Toolbox using a number of different scenarios, and the resulting trace files analyzed. To analyze the trace files, a program was written that takes as input a trace file and visualizes the external events: in particular, two time lines were constructed, one showing the times at which missiles were identified, and another showing the times at which flares were released. An example is shown in Figure 5.

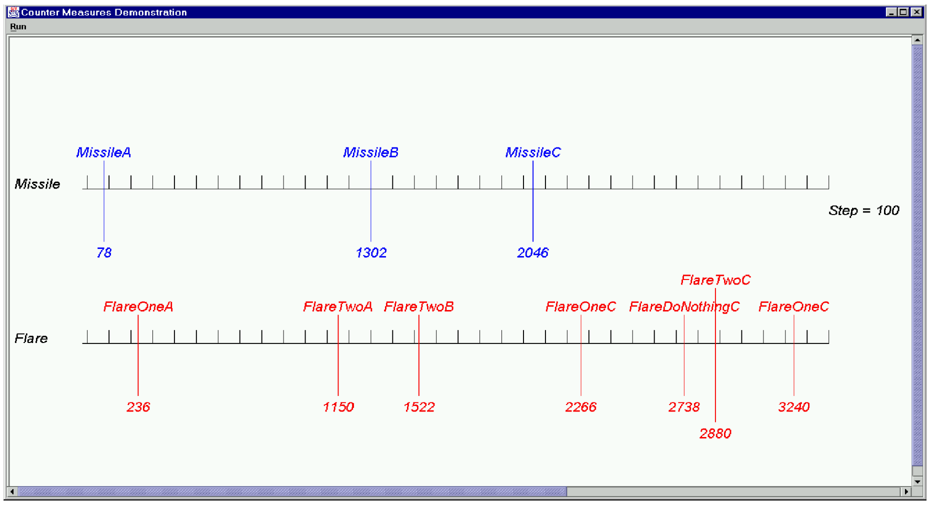


Fig. 5. Visualization of Counter Measures Trace

In the light of this visualization it is interesting to revisit the requirements given in Section 4:

1. The mechanism for dealing with higher priority missiles is clearly working, since when missile B arrives treatment of missile A is aborted, and when missile C arrives treatment of missile B is aborted.
2. The controller sends out its first flare 158 milliseconds after the identification of the first missile. This comfortably meets the deadline of 250 milliseconds.
3. Consideration of the third requirement - "The controller should be able to abort a firing sequence within 130 milliseconds" reveals an ambiguity in the requirements. From this statement, in the situation where a plan is being executed and a higher priority missile arrives, it is not clear whether the first flare in the new plan should be released within 130 milliseconds, or whether it should be released within 130 + 250 milliseconds (including Requirement 2). If the former is the case, then the model does not meet the requirements; if the latter then for this scenario the model satisfies these requirements.

This kind of visualization is particularly useful for communicating model behaviour with domain experts, who do not necessarily have knowledge of VDM++.

5 Comparison

Timed formalisms can be categorized into five broad categories. Those that are: model-based (e.g. VDM++); process algebras (e.g. Timed CSP, [13]); based on logics (e.g. RTL, [8] and DC, [18]); state-based (e.g. Timed Transition Systems, [7], and Timed Automata, [3]); or net-based (e.g. HLTPNs, [4]). Each category has it own characteristics, with attendant strengths and weaknesses. Logic-based formalisms, for example, typically make few assumptions about a computational model, while model-based formalisms such as VDM++ are often based upon a specific model. As stated previously, VDM++'s computational model is a single processor with threads which synchronize when they communicate (via method invocations).

The advantage of generality must be traded against the complexity of having to specify the desired model "from scratch" in each specification, along with the attendant risk of failing to specifying the desired model correctly.

Here VDM++ is only briefly compared with three other model-based timed formalisms: Real-Time Object-Z (RT-OZ); Timed RAISE Specification Language (TRSL); and the Activity Description Language (ADL).

RT-OZ, [16], is a 'dual language' formalism which combines Object-Z with the timed refinement calculus, [9]. RT-OZ extends Object-Z classes with an extra part in which timed refinement calculus expressions can be placed. These expression may refer to timed versions of the class attributes. Timed attributes can also be defined and manipulated within the class itself, and the current time can be accessed.

The declarative nature of RT-OZ makes it easier to specify directly the required timing properties than it is in VDM++. This, however, has the cost of making animation of arbitrary RT-OZ specifications difficult, effectively denying an important analysis technique that VDM++ supports.

TRSL, [17], is a minimal (but not conservative) extension to RSL, [5]: it adds a *wait* expression, which takes a non-negative real as a parameter. Time in TRSL is hence based on points, is continuous (actually, it is super-dense, [10]), and linear, whereas time in VDM++ is based on points, discrete, and linear. TRSL, like Timed CSP, adopts the maximal progress assumption. This can be used to *model* systems which use scheduling, but is less convenient for directly specifying the behaviour required, c.f. TAM, [14].

Time only elapses in I/O and wait expressions in TRSL. This is in contrast with VDM++, where default timing values and duration statements define the time elapse of any particular statement. The VDM++ approach is aimed at being able to explore the temporal effects of engaging in complex computations. The insertion of enough Wait statements in an TRSL specification could be used to achieve this effect, but not as flexibly as in VDM++, where, for example, the time of an assignment could be changed systematically across a specification.

ADL, [12], is a formal notation for defining the temporal and functional behaviour of processes in Real-Time Networks, [15]. The "dynamic states" of the ADL assign lower and upper time bounds to operations: effectively an extension to the VDM++ duration statement which instead of assigning a single (actual) time to a block of statement, assigns a *best case execution time* and a *worse-case response time*. The ADL, however, cannot define time-bounds around arbitrary sequences of statements.

The use of *worse-case response times* makes the ADL suitable for expressing temporal constraints directly, and unlike VDM++, means that the ADL need make no commitment to a particular scheduling or distribution approach. It has a penalty, however, in making ADL specifications harder than VDM++ specifications to animate realistically. The most significant differences between ADL and VDM++ are that one is compatible with real-time networks, and the other with an object-oriented view of systems; and that VDM++ has commercial tool support.

6 Concluding Remarks

In this paper we have presented a new approach for exploring timing properties for models written in VDM++. This is a pragmatic approach which enables the exploration of the timing characteristics in the design stage of the development process. We believe that in particular the animation capabilities will be applicable to many industrial real-time applications. In the VICE project we have used a trial study with the development of a missile guidance control system through the entire development process. Using the proposed approach a number of design errors have been discovered and solved, at a much earlier stage than would have been the case for a traditional development. Thus this study confirms our belief in the applicability of this new approach.

References

1. http://www.ifad.dk/projects/afrodite/afrodite.htm, 1995.

2. Sten Agerholm and Peter Gorm Larsen. A Lightweight Approach to Formal Methods. In *Proceedings of the International Workshop on Current Trends in Applied Formal Methods*, Boppard, Germany, October 1998. Springer-Verlag.
3. R. Alur and D.L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.
4. C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezze. A Unified High Level Petri Net Model for Time Critical Systems. *IEEE Transactions on Software Engineering*, 17(2), February 1991.
5. RAISE Language Group. *The RAISE Specification Language*. The BCS Practitioners Series. Prentice Hall, 1992.
6. The VDM Tool Group. VDM++ Toolbox User Manual. Technical report, IFAD, February 2000.
7. T.A. Henzinger, Z. Manna, and A. Pnueli. Timed Transition Systems. In *Real-Time Theory in Practice*, volume 600 of *LNCS*. Springer-Verlag, 1991.
8. F. Jahanian and A.K Mok. Safety Analysis of Timing Properties in Real-Time Systems. *IEEE Transactions on Software Engineering*, 12(9):890–904, 1986.
9. B.P. Mahony and I.J. Hayes. A Case Study in Timed Refinement: A Mine Pump. *IEEE Transactions on Software Engineering*, 18(9):817–826, 1992.
10. Z. Manna and A. Pnueli. Models of Reactivity. *Acta Informatica*, 30(7):609–678, 1993.
11. P. G. Larsen and B. S. Hansen and H. Brunn N. Plat and H. Toetenel and D. J. Andrews and J. Dawes and G. Parkin and others. Information technology — Programming languages, their environments and system software interfaces — Vienna Development Method — Specification Language — Part 1: Base language, December 1996.
12. S.E. Paynter, J.M. Armstrong, and J. Haveman. ADL: An Activity Description Language for Real-Time Networks. *Formal Aspects of Computing*, 2000. To Appear.
13. S. Schneider. *Concurrent and Real-Time Systems: The CSP Approach*. Wiley, 2000.
14. D. Scholefield and H.S.M. Zedan. TAM: A Formal Framework for the Development of Distributed Real-Time Systems. In *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 571 of *LNCS*. Springer-Verlag, 1992.
15. H.R. Simpson. The MASCOT Method. *Software Engineering Journal*, 1(3):103–120, 1986.
16. G. Smith and I. Hayes. Towards Real-Time Object-Z. In *Integrated Formal Methods*, pages 49–55. Springer-Verlag, June 1999.
17. Y. Xia and C. George. An Operational Semantics for Timed RAISE. In *World Congress on Formal Methods (FM '99)*, volume 1709 of *LNCS*. Springer, 1999.
18. C. Zhou, C.A.R. Hoare, and A.P. Ravn. A Calculus of Durations. *Information Processing Letters*, 40:269–276, 1991.