Exploring Timing Properties Using VDM++ on an Industrial Application

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. In this paper we present a new and unique method of timing property exploration based on the formal specification notation called VDM++. We explain how the VDM++ notation and tool support is being adapted to enable a pragmatic approach to detect potential timing bottlenecks with a software design before expensive commitment to an efficient implementation is made. Finally, an industrial trial project used to validate this approach is described. The application deals with a guidance and control system for a missile.

1 VDM++ And VICE

VICE (VDM++ Specification In a Constrained Environment) is a European ESPRIT Project (no. 27618) aimed at demonstrating the applicability of the formal technology VDM++ in highly constrained real-time systems. The objective of VICE is to demonstrate the suitability and viability of using the formal method VDM++ [2] for the development of highly constrained critical applications.

VICE proposes to address this issue through the use of formal methods and within a trial application where the formal notation VDM++ and associated tools are used for the specification and design of an airborne control system within a missile. This application was selected as it demonstrates performance in an extreme environment, and addresses the continuous trend of the military industry to use Commercial-Of-The-Shelf (COTS) tools from civilian environments to significantly reduce costs.

The formal notation VDM++ was originally developed in the ESPRIT project called AFRODITE [1] and subsequently improved by IFAD. This notation is supported by the VDM++ Toolbox [3]. 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. The current customer base for the VDM++ technology consists of organizations who are interested in VDM++ because of its object-oriented extensions to the ISO standardized VDM-SL [5]. This includes developers of sequential safety- or mission-critical applications which

have a requirement for correctness and reliability. However prior to the VICE project neither the notation or tools provided significant support for concurrent, real-time systems.

Matra BAe Dynamics (MBD) is Europe's leading enterprise in the field of guided weapons. Matra BAe Dynamics France (MBDF) is playing the role of end-user and the overall coordinator of the VICE project. IFAD is playing the role of technology provider and adapting the tools to the specific requirements that are arising during the project.

The VICE project is scheduled to run for 24 months, and is currently at month 18. Therefore this paper provides a snapshot of the project's progress so far and the direction in which the project will go. Thus this paper is organized as follows: the remainder of this section gives a description of the unique characteristics of the defence avionics industry, and also describes the VDM++ technology. In Section 2 an overview of the approach to modelling of timing behaviour is described. The trial application being used in the project is described in Section 3, together with an insight into how the timing properties of the target application are being explored. Finally in Section 4 some concluding remarks are given.

1.1 The Defence Avionics Industry

Missile production is a highly competitive global industry. Software used in missiles is extremely complex due to both the highly constrained environment in which it must execute and the required integrity of the software. Of itself this renders development of such software problematic, but this is exacerbated by uncertainty about the physical environment with which the software must interact. That is, the system specifications are never fully available at the beginning of the software development: they are completed or modified as system issues are further investigated. For example the algorithms can depend on aerodynamic configuration which is well known early in the development cycle, but also on structural stress that can only be known by fire trials which can only be performed late in the development process. This can result in requirements that will be modified and completed throughout the system development. Using traditional approaches to software engineering this can prove to be time and cost consuming.

At MBDF, software is developed for both ground-based systems and embedded systems. In embedded systems there are often safety- or mission-critical requirements including hard real-time requirements. A key question is: How can we be sure that the dynamic architecture satisfies the timing requirements? Here, by dynamic architecture we mean the allocation of computations to schedulable threads. A major objective of the project is to demonstrate that the use of VDM++ and the VDM++ Technology can help to answer such questions.

1.2 VDM++

In this section we describe the VDM++ notation. VDM++ is an object-oriented specification 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).

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.

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, and also *history variables* for that object. A history variable can be used to count the number of requests, activations and completions for an operation on that 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.

1.3 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++. In this section we describe the features currently supported by the tools.

Syntax and Type Checking: Static analysis of models.

Execution: Execution of models using an integrated symbolic interpreter. Debugging of models using breakpoints and single/multiple stepping. Execution of thread-based models, with cooperative round-robin scheduling.

Code Generation: Automatic generation of code from models, into C++ or Java.

Pretty Printing and Test Coverage: Generation of test coverage information from model execution. Output of formatted document, with colouring based on test coverage, incorporation of tables containing percentage coverage.

API: CORBA-compliant API allowing interaction with tools from other applications.

Link with Rational Rose: Automatic generation of UML models in Rose from VDM++ models, and vice versa. Automatic merging of heterogeneous models.

2 Modelling Timing Behaviour

2.1 Background

MBDF develops many real-time systems, with either hard, soft, or a mixture of hard and soft real-time requirements. Such systems typically consist of a number of concurrent tasks executing on a single processor with a real-time operating system. The scheduling algorithm used varies from system to system. MBDF's main objective when performing timing analysis is the validation of dynamic architectures before starting the software integration phase. Ideally feedback on the feasibility of a particular dynamic architecture should be provided at the earliest possible point in the design process. Specifically, it would be desirable to receive feedback before performing schedulability analysis on the final production code.

2.2 Existing Approach

Currently timing analysis is restricted to using the PERTS tool [4]. This tool provides limited feedback about the feasibility of dynamic architectures. The user provides as input the tasks to be scheduled. For instance with periodic tasks, the (constant) time between two consecutive ready times should be provided; for aperiodic tasks the time between two consecutive ready times can be modelled as either uniform or exponential distributions. Priorities may also be assigned to the various tasks. Using this input PERTS calculates whether the tasks are schedulable, giving a yes/no response for each task. There are a number of drawbacks to the current approach: the level of granularity at which analysis is performed is quite coarse; and no feedback is provided on those parts of a design which might be causing timing bottlenecks.

2.3 The VICE Approach

The underlying philosophy for the VICE project is the practical use of precise models. For timing analysis this means using the models constructed during the specification and design phases to improve the quality of feedback concerning the appropriateness of particular dynamic architectures. The approach is pragmatic because the objective is to provide feedback rather than formal guarantees of the kind usually obtained by formal validation or verification.

The basic idea of the approach is to simulate the timing behaviour of the target processor within the 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 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. 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 time at which the event occurred. This time is the reading of the clock on the target processor as recorded by the interpreter when the event occurred. To maintain the internal variable representing the target processor's clock, selected portions of the VDM++ model are enhanced with duration information, a file of default duration information is utilized and the user provides a default task switching overhead. The intention is that for those parts of the model whose timing properties are known from previous experience the designer should specify duration information. Elsewhere worst case analysis based on the default duration information is used. Further details of this are given below. An overview of the approach is given in Figure 1.

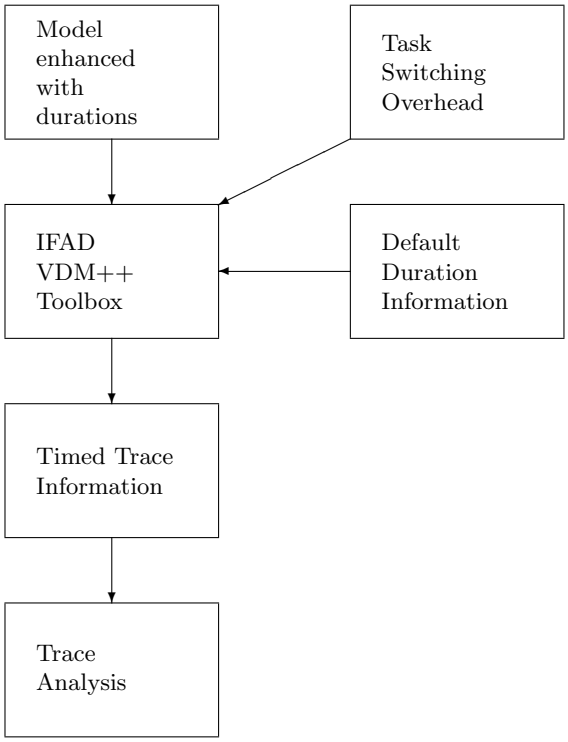


Fig. 1. Overview

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.

In general the approach inside the interpreter is that the time of the previous event is recorded in an internal variable. The task switching overhead is a constant defined by the user. The execution time for statements executed since the previous event, is the sum of the execution times for each such statement. The execution time for an individual statement is:

If the statement does not fall under 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 does fall under 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 is equal to n the time is incremented with 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. However, to date this has not lead to any problems with the VICE trial application.

2.4 Example

Consider a model in which two threads are executing concurrently:

Thread 1	**Thread 2**
`(s1;`	`duration (20)(`
` s2)`	` t1;`
	` t2);`
	`duration (10) t3`

Here, `s1`, `s2` and `t1`,. . .,`t3` are VDM++ statements. According to the model and the scheduling policy a number of different interleavings are possible. We consider two of these interleavings:

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

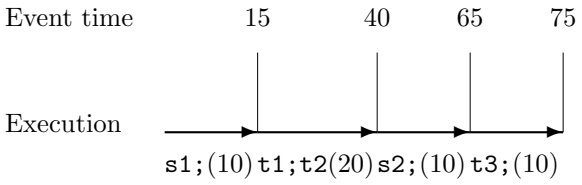
For the purposes of this example, suppose that using worst case analysis based on default timing behaviour for the target architecture, the following execution times are computed for the statements:

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

Suppose further that the user specifies the task switching overhead to be 5 time units. We now consider each interleaving in turn.

Interleaving 1 Three events will be logged to the trace file for this interleaving. After statement `s1` has been executed thread 1 is switched out. The act of switching the task requires 5 time units. So this event is recorded in the log file as having occurred at time point 15. After this statements `t1` and `t2` are executed. The time taken to execute these is 20 time units (by the duration statement). Thread 1 is switched back in now, so including the overhead for task switching this event is logged to the trace file to have occurred at time point 40. The `s2` statement is now executed and the time is incremented with 25 time units. Finally thread 2 is switched back in, statement `t3` is executed and its execution will terminate at time point 75 (by the duration statement). No other trace events occur in this interleaving.

We can represent this interleaving diagrammatically:

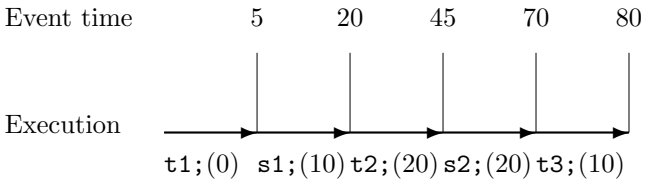


In this diagram, arrows represent statements executed sequentially from a single thread before an event occurs, and the figure in brackets following the

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.

Interleaving 2 Interleaving 2 represents a maximal interleaving. 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 A Guidance And Control Application

In this section we describe the trial application being used to validate the project's proposed approach. The trial application is a simplified autopilot system for a missile (see Figure 2).

The system performs the functions of navigation (which inform the missile about its position in space), of guidance (which drives the missile along the desired path and time schedule which it must follow) and control (which controls the movements of the missile around its centre of gravity). The missile which the autopilot must control is equipped with an engine whose thrust axis is parallel to the axis Gx. It has two horizontal fins (left and right-hand side) and one vertical fin, for which movement is limited to $\pm 5^\circ$ in increments of 0.5°.

The missile flight must be controlled according to three functions: its centre of gravity must follow the correct path (guidance function); the missile movements around its centre of gravity must be stabilized and controlled (control function); and the missile's instantaneous situation in space must be known (navigation function).

3.1 Simplified UML description

At the most basic level we can think of the autopilot as a closed-loop system which reads information from sensors and generates commands to control surfaces based on this information and the desired trajectory. At a deeper level,

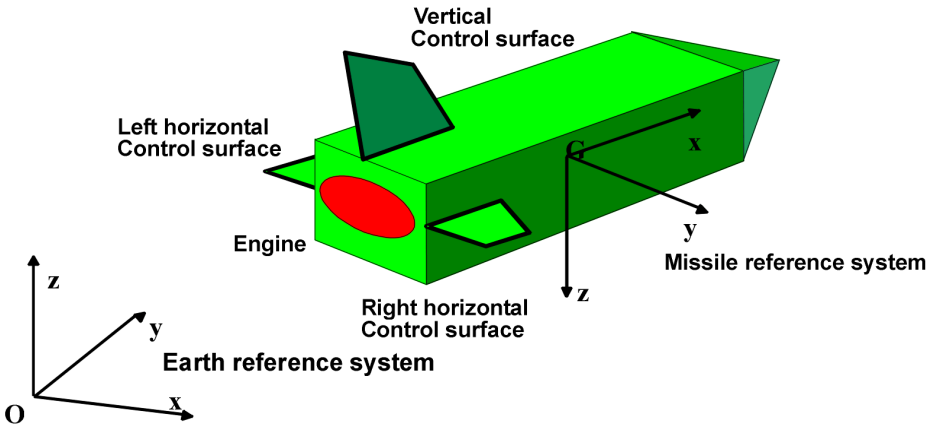


Fig. 2. Missile's Control Surfaces

an autopilot system has an inherent object structure, corresponding to the tasks performed by the system, and information stored and used in the system. In this section we give an overview of this structure.

The Autopilot: is the principal object of the system; it coordinates all the computations needed to pilot the missile.

The InertialMeasurementUnit : is the interface between acceleration and angular rate sensors and the system. This object corresponds to a physical module called inertial measurement unit (IMU).

The Route: is the desired trajectory which the missile has to follow as closely as possible.

The Navigation: uses the flight parameters already delivered by the IMU and computes the current missile attitude. This information is set in the flight parameters object.

The Guidance: compares the current missile attitude with the desire trajectory and generates commands to make the missile follow the trajectory.

The ControlSurfaces: is the interface with all the physical control surface.

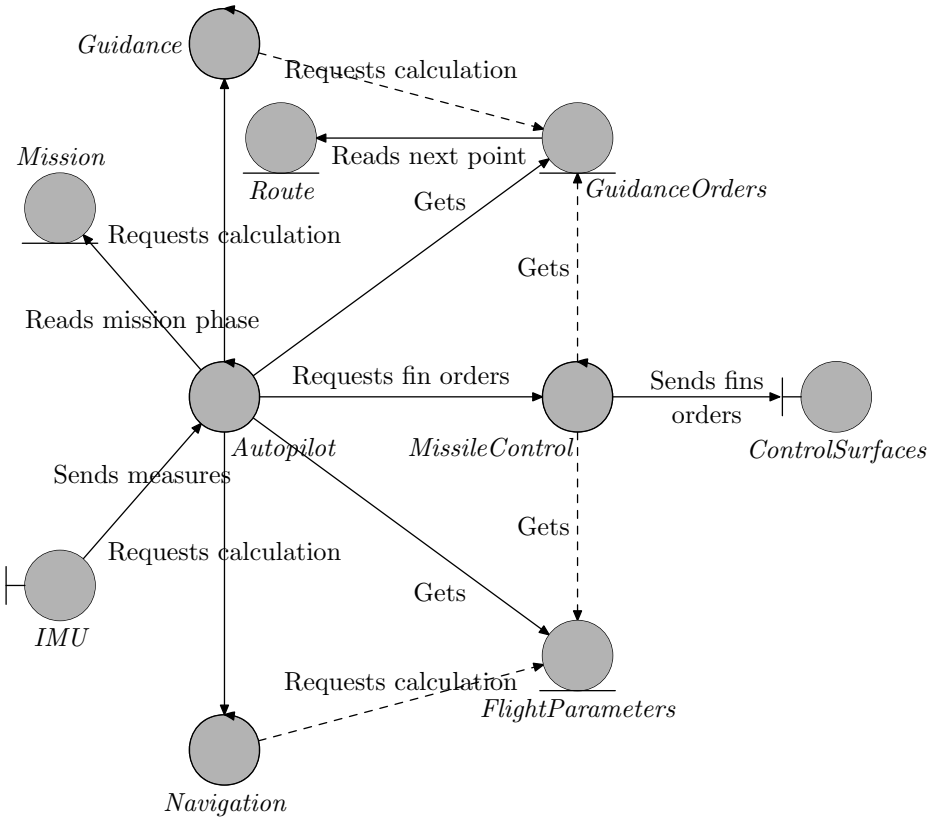
The FlightParameters: represents the current missile attitude based on the most recent IMU value.

The GuidanceOrders: contains the different values computed by the guidance to change the current missile trajectory. They are used with the flight parameters to compute the missile orders.

The Mission: represents the current state of the missile e.g. before launch, during flight etc.

The MissileControl: represents the last phase of the missile order computations. It combines the current missile attitude and current missile orders to deduce the new missile commands.

A diagram showing the inter-class relationships can be seen below.



This diagram shows the different permanent links between classes. It has to be read from left to right. When an arrow (an association for UML) is traced from left to right, the left class provides some information. When the arrow is traced from the right to the left then the right class is sending some information to the left class.

3.2 Dynamic Structure

The autopilot system is an embedded dynamic system. An overview of the dynamic structure can be seen in Figure 3. In this diagram, each box (except *Flight*) corresponds to an active object in the system (that is, an object with its own thread).

Thus we can see that the system consists of a number of periodic and aperiodic threads. Details of periodic frequencies are not available for reasons of confidentiality. However there is a hard real-time requirement on the delay between IMU readings becoming available, and commands being sent to control surfaces.

The job then of the autopilot system is to synchronize the different threads, ensuring that on arrival of the IMU readings the different threads are released in the correct order.

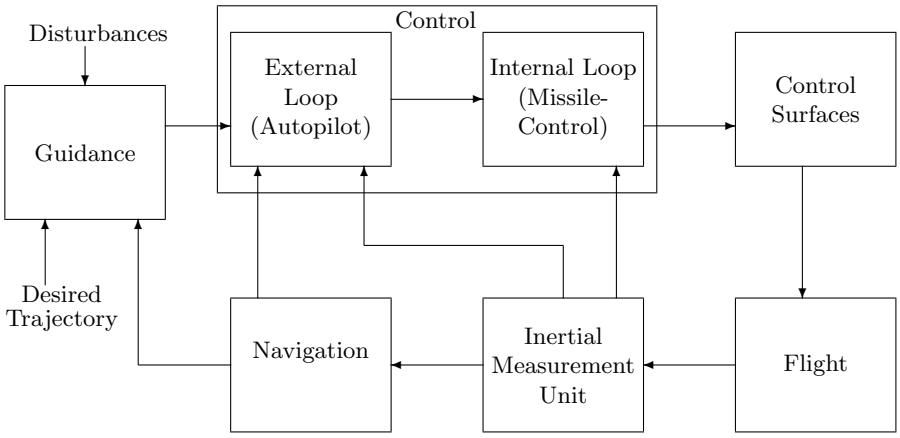


Fig. 3. Overview of Dynamic Structure

3.3 Testing approach

One of the objectives of this project is to allow early testing of the application. The VDM++ technology provides four important tests tools: a debugger that is useful for unitary testing; a batch mode that is useful for large scale testing; an IO class that can be used to read input and write output; and the log file described in Section 2.

To test the model some input data (IMU readings) that has been generated by a guidance missile model is used. After execution with a batch file the computed results are compared with the results from the same model.

Note that the unique aspect of the testing is the use of the log file. This can be used in three ways: for diagnosing deadlocks; for solving behavioural problems relating to incorrect sequencing; and for checking that hard real-time requirements have been satisfied by the model. The first two of these uses are in fact related, as deadlocks typically occur due to incorrect sequencing of threads. By analysing the log file it is possible to see precisely which thread was released when it should not have been, and what triggered its release. In this way the cause can be traced backwards until its root is found and remedied.

Checking of hard real-time requirements is also an example of a more general technique we can use: since the log file is essentially a sequence, it is possible to read it back into the VDM++ Toolbox using the VDM++ IO class, and then analyze it using arbitrary VDM++ predicates. This allows an extremely powerful mechanism for validating run-time properties of the system.

3.4 Benefits with the VICE approach

Using the approach described above is very useful as it allows the debugging of real-time software architectures as soon as possible in order to detect unsatisfied requirements. It should be a good way to support and to simplify software test phases such as the integration one. Generally, when this activity is performed,

the first step starts with testing together hardware and software. At this stage, when a problem occurs, developers encounter great difficulty in finding the real cause of the failure (Is it a dynamic or a scheduling failure of the software? Is it a hardware failure? etc). Separating these problems would give developers much more confidence in their application if it has already been tested statically and dynamically.

Another point from our previous experience is that the production of an operational hardware platform can be delayed in time while the project milestones are not. In that case, software people could continue performing deep testing with a simulated behaviour of the application, and of its environment, that is close to the real one.

These two points are key issues for project managers who are concerned with the reduction of costs and development duration.

This approach should also be of interest to our customers, for it would make it possible for the software team to reply speedily to queries concerning the feasibility of adapting the system to respond to changed requirements. Determining the feasibility would be aided by the pre-existing model, and by its object-oriented architecture. In the case of a major problem, software specialists could also propose alternative solutions (either software or hardware) to satisfy these new requirements and help customers choose between them.

4 Concluding Remarks

4.1 Current Status

MBDF has already experienced the use of formal methods in different case studies in the field of embedded safety- or mission-critical systems. These studies involved a formal method approach for the specification and, if possible, for design and code generation.

Currently, the functional requirements included in our specifications contain no reference to safety- or mission-critical issues although, from a quality point of view, code must be produced according to best practices. In the future, the national and international customers of MBDF will probably enjoin us to develop new equipment in accordance with recognised safety standards. For this reason, MBDF has started to evaluate the benefit of specifying functional requirements with formal methods without real success so far due to limitations of the ones investigated. At the moment, VDM++ offers the same characteristics as the formal methods listed and criticised above.

MBDF wants now to explore new tracks and more precisely focus on the representation and validation of timing requirements – a major point in the safe dynamic behaviour of an embedded real-time application. VDM++ and the extensions specified so far by the VICE team represent the appropriate starting point to this work allowing precise description and testing, in the design phase, of the dynamic architecture of the trial application.

The modified VDM++ language will offer a completely integrated set of functionalities both for functional and timing requirements analysis. Thanks to

this, VDM++ will have a clear lead over other methods and give the ability to MBDF to reduce its investment and functioning costs. MBDF (and UK) places its hopes in this promising approach and has no doubt that this will help the company to improve its competitiveness in the international defence market.

4.2 Future Direction

The work described in this paper is an early experiment in using a formal specification language that has constructs for concurrency and time, and which is supported by a commercial tool that provides animation.

The ability to explore the temporal behaviour of a proposed system from its specification has the potential to become an important factor in the adoption by industry of formal specification techniques. It may even motivate their regular use on non-critical real-time applications.

The main benefits that this technology promises are: a systematic way of expressing temporal constraints, aiding system conception and definition; a user-friendly environment for exploring the consequences of the various elements of the system having particular temporal properties; and the ability, early in the system development, to allow performance requirements to influence intelligently the proposed hardware and software solution.

We believe that through the VICE project, the VDM++ notation and toolbox have taken a step in the right direction to realise these benefits. However, we can see further developments which would help. In particular, we have ideas about defining temporal deadlines, windows, and jitter constraints over the time trace. Also, we are aware that the accuracy of the scheduling model is crucial, and can imagine a future version of the toolbox either providing a number of options to enable a user to tailor it, or providing a mechanism for a user to override the default scheduler. A desirable, but more extensive development, would be the addition of the ability to model multi-processor systems with heterogenous scheduling policies. Future industrial feedback will clarify exactly which extensions will be needed to realise the desired benefits.

References

1. http://www.ifad.dk/projects/afrodite/afrodite.htm, 1995.
2. The VDM Tool Group. The IFAD VDM++ Language. Technical report, IFAD, February 2000.
3. The VDM Tool Group. VDM++ Toolbox User Manual. Technical report, IFAD, February 2000.
4. J.W.S. Liu, J. Redondo, Z. Deng, T. Tia, R. Bettati, A. Silberman, M. Storch, R. Ha, and W. Shih. PERTS: A Prototyping Environment for Real-Time Systems. Technical Report UIUCDCS-R-93-1802, University of Illinois at Urbana-Champaign, 1993.
5. 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.