Modeling and Validating Distributed Embedded Real-Time Systems with VDM++

(working draft – *Revision* : 1.3 – November 6, 2005)

Marcel Verhoef [1], Peter Gorm Larsen [2], and Jozef Hooman [3]

[1] Chess Information Technology B.V., P.O. Box 5021, 2000 CA Haarlem, The Netherlands.*
http://www.chess.nl email: Marcel.Verhoef@chess.nl
[2] University College of Aarhus, Dalgas Avenue 2, DK-8000 Århus C, Denmark.
http://www.iha.dk email: pgl@iha.dk
[3] Embedded Systems Institute, Den Dolech 2, 5612 AZ Eindhoven, The Netherlands.*
http://www.esi.nl email: Jozef.Hooman@esi.nl

Abstract. To Be Written.

1 Introduction

To Be Written.

1. Introducing VDM++ (move from start of Section 2?).
2. Explain the typical properties of distributed embedded software development, use [1]. Typically SFM are used but they are limited when complex algorithms come into play (for example higher levels of control and error handling). Also introduce commonly accepted RTE notions such as jitter, clock drift.
3. Aim of modeling is to increase the insight into the system under construction at minimal cost.
4. Make minimal changes to both VDM++ syntax, static and dynamic semantics. Keep the application specification as much as possible free from (hardware) architecture specific knowledge.
5. Concrete ASCII syntax for case study in VDM++ and mathematical syntax for semantics in VDM-SL.
6. Make clear to the reader the distinction between models at the VDM++ level (application models?) and at the meta-level (semantic models?). Align all sections accordingly.

1.1 Contribution of this paper

To Be Written.

1. Enabling (bridging the gap between) formal software engineering and hardware / networking disciplines.
2. Properly deal with the notion of deployment on the computation and communication level.
3. Improvement of VDM++ technology to better support distributed and embedded real-time systems. Ability to describe a whole new class of systems.

1.2 Related work

To Be Written.

1. Certainly we have to reflect on TrueTime, Ptolemy and Giotto and show why we are different.
2. Check the dan.bib file for more related work, just like [2] and [3] – product families.
3. Discussion notion of deployment in UML and AADL.

* and Radboud University Nijmegen, Institute for Computer and Information Sciences, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands. *http://www.cs.ru.nl/ita/*. This work has been carried out as part of the Boderc project under the responsibility of the Embedded Systems Institute. This project was partially supported by the Netherlands Ministry of Economic Affairs under the Senter TS program.

2 The VDM++ approach to real-time

The formal notation VDM++ is an object-oriented, model-based specification language, and is largely a superset of the ISO standardized notation VDM-SL [4]. VDM++ was originally developed in the ESPRIT project AFRODITE and was subsequently improved by the Danish company IFAD. The real-time extensions to VDM++ where developed as part of the ESPRIT project VICE: "VDM++ In a Constrained Environment". Both the VDM++ and VICE notations are supported by the industry strength VDMTools, which is now owned and maintained by the Japanese company CSK [5].

VDM++ provides a precise and unambiguous notation for analysis of requirements and allows for early validation through testing and debugging using VDMTools. In this way it is possible to bring testing activities forward to the specification phase of the development life-cycle. This pragmatic approach to applying formal techniques for systems development has been instrumental for a number of very successful applications in industry, for example [6], [7] and Felicia Networks. In Section 2.1 we present a short overview of the VDM++ language and its real-time extensions, both in terms of capabilities and limitations. In Section 2.2 we will investigate how to overcome the limitations by introducing some extensions to the notation and its associated tool support.

2.1 The existing VDM++ capabilities and limitations

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, types such as sets, sequences, tuples, records 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.

Synchronization: Operation invocation is defined with synchronuous semantics (rendez-vous). 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.

VDMTools is a comprehensive suite of tools for the analysis and validation of formal models described in VDM++. Different variants of VDMTools exists but in this article we limit our efforts to the version resulted from the VICE project and an auxiliary tool called `ShowVICE` [8]. The only part of VDMTools important for this article is the interpreter. It is able to perform execution of VDM++ models. It has functionality enabling debugging using breakpoints and single/multiple stepping directly at the VDM++ level. Execution of thread-based models, with different scheduling policies selected by the user e.g. cooperative round-robin scheduling.

Specifically in the VICE version of VDMTools the notion of time and periodic threads is available. In general the approach inside the interpreter is that the time of the previous instruction 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. However, there is a notion of a duration 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 basic idea of the approach is to simulate the timing behaviour of the target processor within the 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 [4].

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. After the completing of an execution it is then possible to make use of the `ShowVICE` tool to automatically get information about the trace produced in the form of an extended sequence diagram with time annotations and thread context.

Limitations

1. VDM++ is in fact a synchronous language, there is no notion of events
2. VDMTools only supports uni-processor multi-threading
3. TR: No access to the "global time" (if space available also new permission predicates).
4. No notion of distribution in the language. Describe 5 basic communication patterns that we want to support.
5. Fixed format for trace file.
6. Only absolute time progress (`duration`) is supported, no relative notion of time (depending on CPU context or BUS properties).

2.2 Suggestions to overcome the limitations of VDM++

To Be Written.

1. Introduction of `async` operation calls.
2. Introduction of `system`, `CPU`, `BUS`.
3. Introduction of `cycles`.
4. Flexible trace file.

3 Demonstrating the strength of the suggested approach – a case study

The case study presented in this section, which was first published in [9] using other analysis techniques, is inspired by a system architecture definition study for a new distributed in-car radio navigation system. Such a system typically executes a number of concurrent software applications that share a common hardware platform. Each application might have hard individual performance requirements that need to be met. During the system definition phase, several candidate platform architectures are proposed by the engineers and the system architect needs to evaluate all of them and decide which one to implement. The system architect makes these decisions by building and analyzing models that support the design choices. From our own experience, the success of this activity is very much dependent on the amount of time needed to perform the work. Time-to-market targets are hard, especially so in the embedded systems domain and modelling and analysis

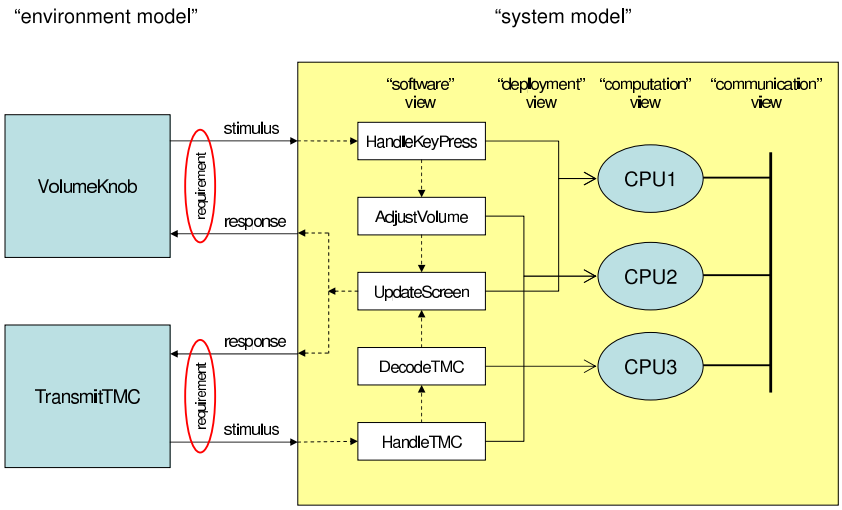


Fig. 1. An overview of the case study

techniques that do not meet this requirement are simply not used in practice, even despite their potential [DANA].

The in-car radio navigation system contains many applications, we will consider just two of them here because of space limitations; the full model is available in [TECHREP]. An informal overview of the case study is presented in Figure 1. An embedded system, as its name suggests, needs to be described in combination with its environment. In our model, we partition the *world* into an *environment model* and a *system model*.

In our case study, there are two tasks that represent the environment. They independently inject stimuli into the system and observe its response. The rate at which these events are generated can for example be described by the triple (p, j, d) where p represents the period, j the jitter and d the minimum time between two consequtive events. Typically system-level temporal and timing requirements can be specified at this interface. Informal examples of these requirements are respectively: *"The order of the input stimuli is preserved in the output response sequence."* and *"For each stimulus, the maximum response time shall be less than 100 msec."*. We will show later how to formalise these requirements in VDM++.

The *system model* is composed of a number of views. First of all, the *software view*. There are two applications in our example that consist of three tasks each. One of the tasks is actually shared by both applications. Tasks can either be triggered by external stimuli (interrupts) or by receiving messages from other tasks. A task can also actively acquire information by periodically checking for available data on an input source. All three notions of task activation are supported by our approach.

Secondly, the *deployment* and *computation* views. The deployment view describes on what computation resource each task is allocated. Although the notion of deployment is supported by other description techniques as well, such as for example UML 2.0, their value is often rather limited. In contrast, it is essential in our approach because the system behavior is determined by the allocation of tasks on resources. And in addition, our notion of deployment also supports dynamic task creation, which to our knowledge has not been done before.

Finally, the *communication* view. This view describes the internal connections between the computation and communication resources.

1. Finish describing case (network view)
2. Create UML diagrams to show old and new

[4] TR:For the purposes of this paper we restrict our interest to the swapping in and out of threads.

3. Explain new stuff in detail, with focus on new constructs
4. Show old and new trace files and post-analysis capabilities

4 Semantics of the VDM++ Real-time approach

In order to understand the semantics consequences of the suggestions made for improving VDM++ in its ability to describe and validate distributed real time systems. The two subsections below presents the existing semantics of the handling of threads and the changes resulting from the suggested improvements respectively.

4.1 Dynamic Semantics for the existing VDM++ Interpreter

The semantics of VDM++ descriptions is (just like for VDM-SL in its ISO standard) defined in terms of sets of the set of mathematical models satisfying the constraints given in the descriptions. This means that the interpreter from VDMTools essentially reflects one of these models.

At the abstract level we could model the abstract syntax of threads as a sequence of statements:

types

$$Thread = TimedStmt^*$$

The semantics for threads in the existing version of VDM++ then is a set of traces with timing information interleaving the execution of the different threads in a VDM++ description. Traces are sequences of trace elements where trace elements represents an uninterrupted execution on the processor with information about when the execution began and ended. In [10] it was demonstrated how this set of traces can be specified.

Thus in order to illustrate the consequences of the suggested modifications to VDM++ it is probably worthwhile to briefly describe how the existing interpreter is able to evaluate the VDM++ treads and producing one of the possible traces depending upon the chosen scheduling algorithm. In general the interpreter is constructed as a stack-machine where the VDM++ descriptions first are compiled into instructions that can subsequently be interpreted by the stack-machine. The entire interpreter is itself specified in VDM-SL taking up more than 400 pages including explanation [11].

We will not consider the actual compilation stage here for space limitations. Figure 2 (a) presents a short overview of the structure of the main functionality dealing with the stack-machine interpretation of the instructions relevant for threads. The different operations mentioned in Figure 2 (a) have the following responsibilities:

EvalRun: This is the top-level operation for the stack-machine used when the user wish to execute a VDM++ construct. Prior to this the different VDM++ descriptions have been compiled into instruction code.

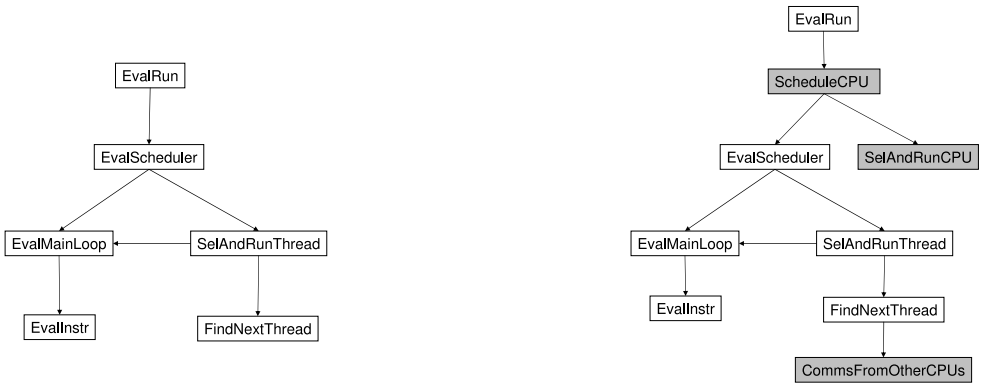
EvalScheduler: This operation deals with slicing the evaluation of the different instructions belonging to different threads up according to the scheduling policy selected by the user. It makes use of *EvalMainLoop* for evaluating the chosen thread and once that returns it uses *SelAndRunThread* to select and execute the next thread if any more needs to be executed.

EvalMainLoop: This is the main stack machine loop responsible for executing a sequence of instructions. Its execution may be blocked in different ways (e.g. by a permission guard being false) and will signal back about this when it is finished.

EvalInstr: This is the evaluation of a single instruction in the instruction set. It is essentially a large case onto the different kinds of instructions to their own evaluation operation.

SelAndRunThread: This operation selects and run the next thread according to the scheduling policy selected. In case no threads are ready to be executed at this point of time this operation is also responsible for advancing the time to the next periodic thread becoming ready to run.

FindNextThread: This operation is responsible for finding the next thread to be executed. Thus, it must inspect the guards of the different blocked threads and take this into account with respect to the selected scheduling policy.



(a) Existing VDM++ Interpreter.

(b) New VDM++ Interpreter.

Fig. 2. Operation Call Structure in the VDM++ Interpreter - **FIXME: sizes**

4.2 Updates to the Interpreter for the suggested approach

With the suggested approach with multiple processors the interleaved semantics for threads will be modified. Now potential true parallel execution will be carried out at the different processors. Conceptually speaking this can be considered as having a set of traces similar to the semantics for the current semantics for VDM++ per processor. Subsequently traces from each processor than needs to be merged according to the respective time stamps with the traces from the other processors.

Figure 2 (b) illustrates how the overall structure for the stack machine interpreter would need to be changed with the suggested approach. The operations in Figure 2 (b) with a gray background corresponds to the new operations required. The main responsibilities for these would be:

ScheduleCPU: The main *EvalRun* operation would now call the *ScheduleCPU* that would be responsible for the scheduling of the different processors being chosen for execution. Because of the possibility of asynchronous communication between the different CPUs the slices must be sufficiently small in granularity.

SelAndRunCPU: This operation is responsible for selecting the next CPU to execute once a CPU have been thrown out of the *ScheduleCPU* operation.

CommsFromOtherCPUs: This operation needs to take communication from other CPU's into account. Independently of the scheduling policies chosen for the different CPUs each of the CPU's will have a special thread dedicated to deal with such incoming messages with higher priority than the threads specified by the user.

1. Show extracts of some of the semantics operations from the two figures.
2. Show the semantics of the implicit routing of messages.

5 Concluding remarks and future work

To Be Written.

1. no clock drift, assume time synchronous for the whole model

Acknowledgments

The authors which to thank Evert van de Waal and Shin Sahara for their valuable comments and support when writing this paper.

References

1. Lee, E.A.: Absolutely positively on time: What would it take? IEEE Computer (2005) 85–87
2. Huijsman, R., van Katwijk, J., Plat, N., Toetenel, H.: Action-Event Modeling Using an Extended VDM-SL. Technical Report 93-117, Faculty of Technical Mathematics and Informatics, Delft University of Technology, P.O. Box 356, 2600 AJ Delft, The Netherlands (1993)
3. Garlan, D., Delisle, N.: Formal Specifications as Reusable Frameworks. In Dines Bjørner, C.H., Langmaack, H., eds.: VDM '90 VDM and Z– Formal Methods in Software Development, VDM Europe, Springer-Verlag (1990) 150–163
4. Andrews, D., Bruun, H., Hansen, B., Larsen, P., Plat, N., et al.: Information Technology — Programming Languages, their environments and system software interfaces — Vienna Development Method-Specification Language Part 1: Base language. ISO. (1995) Draft International Standard: 13817–1.
5. CSK: VDM homepage. *http://www.csk.com/support_e/vdm/index.html* (2005)
6. van den Berg, M., Verhoef, M., Wigmans, M.: Formal Specification of an Auctioning System Using VDM++ and UML – an Industrial Usage Report. In Fitzgerald, J., Larsen, P.G., eds.: VDM in Practice. (1999) 85–93
7. Hörl, J., Aichernig, B.K.: Validating voice communication requirements using lightweight formal methods. IEEE Software **13–3** (2000) 21–27
8. Verhoef, M.: On the use of VDM++ for specifying real-time systems. Proc. First Overture workshop (2005)
9. Wandeler, E., Thiele, L., Verhoef, M., Lieverse, P.: System Architecture Evaluation Using Modular Performance Analysis – A Case Study. In Margaria, T., Steffen, B., Philippou, A., Reitenspiess, M., eds.: Proc. 1st International Symposium On Leveraging Applications of Formal Methods (ISOLA 2004), Paphos, Cyprus, University of Paphos, Department of Computer Science (2004) 209–220 Also available at *http://www.cs.ru.nl/research/reports/info/ICIS-R05005.html*.
10. Mukherjee, P., Bousquet, F., Delabre, J., Paynter, S., Larsen, P.G.: Exploring Timing Properties Using VDM++ on an Industrial Application. In Bicarregui, J., Fitzgerald, J., eds.: The Second VDM Workshop. (2000)
11. CSK: The Dynamic Semantics of CSK VDM++ VICE. Technical report, CSK Corporation, Japan (2005) Company Confidential.