Overview of VDM-RT Constructs and Semantic Issues

Kenneth Lausdahl[1], Marcel Verhoef[2], Peter Gorm Larsen[1] and Sune Wolff[3,1]

[1] Aarhus School of Engineering, Dalgas Avenue 2, DK-8000 Aarhus C, Denmark
[2] Chess, PO Box 5021, 2000 CA Haarlem, The Netherlands
[3] Terma A/S, Hovmarken 4, DK-8520 Lystrup, Denmark

Abstract. An inventory of current semantic models of the Vienna Development Method (VDM) notations are presented, for which their purpose, strengths and weaknesses are assessed. The focus will be on VDM Real Time (VDM-RT) with (multi-threading and multi-core) concurrency, communication and real-time. Areas are identified where the semantics is currently unclear, incomplete or even undefined. Challenges in adopting novel language concepts are investigated, for example for modelling uncertainty in real-time distributed systems. Approaches taken by other formalisms are presented and suggestions are offered how these ideas could be applied in the context of VDM-RT. The result of this work aims to become a roadmap for the definition of a full semantics of VDM-RT, on the short term focused on symbolic execution (simulation), but needs to be amenable to formal proof and exhaustive search (model checking) in the future.

1 Introduction

The Vienna Development Method (VDM) was first conceived at the IBM laboratories in Vienna in the early seventies [Luc87,FLV08]. Later on different dialects of VDM evolved at different places in the world and BSI and subsequently ISO standardised a version of VDM called VDM-SL [P. 96,PT92]. As the object-oriented paradigm gained popularity, an object-oriented extension called VDM++ was developed in the European research project called Afrodite [FLM$^+$05]. In order to provide better facilities for modelling real-time embedded and distributed systems a VDM Real Time (VDM-RT) extension was developed [MBD$^+$00,VLH06].

The initial version of VDM-RT was developed in a European research project called VDM In Constraint Environments (VICE) but this was developed for a single CPU [MBD$^+$00]. At the beginning of the PhD work by Verhoef this VICE version was used to attempt to model a distributed real-time embedded system [WTVL04]. This motivated extensions of VDM-RT to incorporate constructs enabling the formulation of distributed systems and as a consequence it was possible to model this case study in a much better fashion.

The semantics made for VDM-SL are very strong, and gives a solid foundation for the language. These semantics are described using the denotational semantics style. Less work has been put into the semantics for VDM++ and VDM-RT, and hence more issues are present. It is some of these issues for VDM-RT that we will describe in this paper, and present possible solutions.

After this introduction, Section 2 provides an introduction to the constructs from VDM-RT that are extensions compared to the VDM++ notation. Afterwards Section 3

provides a list of semantics issues in the current version of VDM-RT that needs to be solved. This is followed by Section 4 which provides a list of the corresponding issues in relation to the co-simulation that is developed as a part of the DESTECS project. At the end of the article, Section 5 provides an overview of the future work in this area and finally Section 6 gives a few concluding remarks about the practical plan for producing the necessary semantic definitions.

2 Constructs in VDM Real-Time

In VDM++ models are structured in classes and inside these one can define constant values, instance variables (the state of instances of a class), functions with expressions as their body and operations which have statements as body and can access and modify instance variables that are visible inside a class. In addition a class can have a thread and synchronisation of threads is controlled using permission predicates that are logical expressions describing the requirements for activation of a particular operation. VDM-RT is an extension to VDM++ and from here it is important to note that static operations can be defined inside VDM++ classes. In VDM-RT it is furthermore possible to make use of asynchronous operations in VDM-RT using the **`async`** keyword. This will spawn a new thread that will be executed concurrently with the calling thread.

In VDM-RT a special **`system`** class administers the static topology of components inside the system. This includes the ability to distribute instances of classes (objects) to a special type of predefined `CPU` class. For each `CPU` the designer can specify the scheduling mechanism as well as the capacity described as number of computations per time unit. In order to enable communication between active threads in the system, `CPU`s can be connected by `BUS`es. For each individual `BUS` the designer can specify the bandwidth as well as which `CPU`s are connected by the `BUS`. All instances created inside a deployed instance will be residing on (deployed to) the same `CPU`. A special virtual `CPU` and a special virtual `BUS` are implicitly created. The virtual `CPU` will be used to all instances that are not explicitly deployed to a `CPU` declared in the system class. This is used for the things that are outside the system (i.e. the environment classes and the debugging desired by a user).

In VDM-RT, there is a notion of time but per default no unit of time is assumed so that is up to the user. However, the convention most frequently used is that the time unit is milliseconds. In order to access the current global time, the keyword **`time`** is used. All VDM constructs have been assigned a default duration and thus the semantics include a time penalty. If the user can give estimates of fixed execution times, this can be described using the **`duration`** statement which then overrule the default durations of the body statements. If, instead, the execution time is relative to the speed of the `CPU` on which the thread is deployed, the **`cycles`** statement can be used which again overrule the default durations. In the case of nested durations and cycles these are also overruled by the outermost duration/cycles statement. The virtual `CPU` is special in the way that here all default durations are set to zero in order for the timing influence of the elements outside the system in a simulation having minimal impact (unless the user wishes that to happen by inserting duration statements).

Semantically, a multi-cpu VDM-RT model can either conduct an execution step or progress the global time (for all resources). So conceptually speaking there is a master scheduler that allows all `CPU`s to progress until they need to take a time step. When all the `CPU` schedulers have done this they report back how large a time step they would like to take. For `BUS`es, the analogy is whenever the next message is due to be delivered at a receiving `CPU`. The master scheduler then takes the smallest of these time steps and tell all `CPU` schedulers to advance with this time and execute again if they have no remaining time to wait. The semantics provided in [Ver08] on purpose does not include anything for the scheduling, since in essence, it corresponds to a reduction of the possible interleavings of the concurrent threads executing.

Finally it is possible to specify periodic threads using a 4-tuple (p, j, d, o): p describes the period; j is the jitter, d is the minimum time between invocations of a periodic operation and o is the initial offset (see Fig. 1). Note that this syntax does not allow to specify sporadic behaviors. Sporadic threads are threads which are periodic, whereby only a value for d is specified.

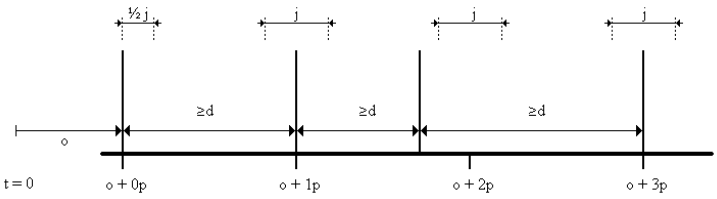

Fig. 1. Period (p), jitter (j), delay (d) and offset (o)

The implementation of the VDM-RT semantics is visible for the executable subset in the interpreters in VDMTools [FLS08] and Overture [LBF+10]. In this paper we will limit our investigation to the VDMJ interpreter from Overture.

3 Open Semantic Issues in VDM-RT

The use of VDM-RT has lead to the identification of missing semantics definitions, or to constructs which are currently defined without distribution having been taken into account. Lastly, the semantic definition of some constructs deviates from the reality implemented in the VDM-RT interpreter. This all makes it hard for a user to specify a model which is faithful to the real world distributed system.

3.1 Public variables

The semantics of the current VDM-RT interpreter allows read and write access to all public variables in a model without the use of a BUS. This means that any public instance variable or value can be accessed from any CPU in a system without taking

distribution into account, where variables located on different CPUs require BUS communication. This will also influence the timing of the CPUs accessing the variable.

We suggest that all access to such public variables are done through a BUS if the CPU reading or writing the public variable is different from the CPU where the variable is located. We suggest that BUS communication is used for any public variables including static variables. Alternative solutions might be to assume that `get` and `set` operations are always implicitly available on class member variables. External references across CPU boundaries, even through public variables, should always be performed through these implicit `get` and `set` operations, much akin to the solution taken in `C#`. Another approach could be to disallow public member variable access across CPU boundaries by static or run-time checking.

3.2 Static variables and operations calls

In the semantics of the current VDM-RT interpreter static variables are globally available to all class instances in a system. Static variables can both be read and written to without taking deployment into account, thus no replication of static variables is done between CPUs in the system. The semantics does not distinguish if a class with a static variable is deployed to more than one CPU, or from which CPU a static variable is accessed. However, by ignoring distribution of static variables, the semantics positions itself far from reality since such distribution has to be carefully performed to ensure that static variables are updated globally at a particular time.

Static operations are allowed to be called from any class instance in a system and are executed on the CPU of the caller. Deployment is not considered thus no BUS communication. Ignoring distribution of static operations causes a number of problems, (1) by ignoring distribution and using static operations to access static variables the problem of static variables applies as described above; (2) if a static operation is called from a CPU which does not have any knowledge of the static operation(since no instances of its class is deployed to it) then it should not be possible to execute it locally, however, this is currently possible. If a static operation should be executed it must exist in the system and be deployed to a CPU; (3) If a static operation is accessed and an instance of a class where it is defined is deployed in a system, access to the definition should happen through a BUS, if accessed from another CPU than the one hosting the static definition. This is not the case since all static calls bypass all BUS communication.

We suggest to change the way static variables and operations are interpreted in VDM-RT.

Static variables should only exist within the CPU they have instances that are deployed on restricting direct access from other CPUs. If more than one instance of a class declaring a static variable are deployed to the same CPU the static variable should naturally be shared. Thus, in this way static variables will become shared per CPU.

Static operations should be available throughout the system, however, the execution of the operation may vary based on the deployment, which can be split up into three alternatives:

Static definition available on calling CPU The caller of a static operation is located on the same CPU as at least one instance of the class declaring the static operation.

In this case the operation can be accessed directly and all execution will take place at the same CPU.

Static definition only available on virtual CPU No instances of the class declaring the static operation exists on any system CPU, thus the caller does not have direct access to the static operation. In this case the static operation will be hosted by the virtual CPU, and the caller has to access the static operation through a BUS connection to the virtual CPU. The static operation is executed on the virtual CPU and the call on the callers CPU.

More than one CPU hosts the static definition If more than one CPU holds instances of the class declaring the static operation and no instance exists on the CPU of the caller, a CPU is selected non-deterministically to host the static operation call. Thus in this case the calling CPU request a call of the operation, this is send over the BUS to the host CPU which activates and executes the static operation and returns the result over the BUS to the calling CPU.

The suggestion provided for static operations does not give semantics to permission predicates specifying the scope of history counters for static operations. This is a problem if a static operation is evaluated on a non-deterministically chosen CPU where the history counters are local to the chosen CPU.

3.3 Time advance and periodic threads

The VDM-RT semantics in [Ver08] and of the VDM-RT interpreter specifies that time must progress when expressions or statements are executed on a system CPU. However one exception is explicit use of a duration statement which require the virtual CPU to advance time as well. The semantics defines periodic threads as statements being activated for execution each P time units on a CPU, it does not give any guarantee about the execution, only that it can execute after its activation time. However no semantics are defined for the case where a model is blocked but contains periodic threads which causes a model to wrongly deadlock. A deadlock occurs in the current semantics if a model consists of at least two threads, one being main and another being a periodic thread, where main blocks after starting the periodic thread. The reason for this deadlock is that time does not progress if nothing is executing, which occurs when main blocks and the current periodic thread finishes at time t where $t < P$ and where P is the period of the periodic thread running. If $t < P$ then the activation time of the next periodic thread is not reached and the model will deadlock. This is not aligned with the idea of periodic threads, where all periodic threads can be activated independent of any model state. The current semantics does not give any special semantics to multiple periodic threads located on the same CPU. The semantics only allow one thread to execute on a CPU at any point in time. However this is a problem if periodic threads are located on the virtual CPU and have the same period which in effect should let them execute synchronously because time is ignored on the virtual CPU.

We suggest that the semantics allow forward time jump in the execution of a model if all threads are blocked and one or more periodic threads exist. If a model is blocked at time t it can suggest the master scheduler to jump to P_e if $t < P_e$ when P_e is the time of the next period of the periodic thread which has the earliest activation time.

4 Open Semantic Issues with Co-Simulation

Extending VDM real-time to support co-simulation as used in DESTECS [BLV+10] requires a number of new semantic definitions and clarifications of existing informal definitions. From [Ver08] it shows that semantics must be given to *interrupts* and time must be given an international standard such as [s] seconds, so other simulators e.g. from the continuous time domain, can synchronise time during simulation.

4.1 Interrupts

Interrupts is a new way to connect model and environment where interrupts spawns new threads called *interrupt handlers* and influences the scheduling of a model. Interrupts can occur at any point in time during execution. In the semantics of VDM-RT a model is connected to the environment through a class instance deployed to the virtual CPU which feeds environment events into the model and allows the model to read environment state. We propose to extend the VDM-RT semantics with definitions for interrupt handling. An interrupt is an event occurring in the environment which is pushed into the model and handled by an *interrupt handler*. An interrupt handler must execute with a variable priority higher than all normal threads and it should be possible to determine if an interrupt handler has completed within a certain time frame. To enable interrupts in VDM-RT a more advanced scheduler, capable of prioritising interrupts, is required along with a clear definition of how interrupts influences the execution of both normal and asynchronous threads. Lastly, interrupt handlers should be asynchronous operations. Listing 1.1 is a proposal of a definition of an interrupt handler.

```
class InterruptHandler
operations
public async notify : () ==> ()
notify == is subclass responsibility;
end InterruptHandler
```

Listing 1.1. VDM `InterruptHandler` signature

However the definition in Listing 1.1 should not be part of the model but built into the tool. In Listing 1.2 an example is given of an interrupt handler for a button press event in the environment.

```
class ButtonPressInterruptHandler is subclass of InterruptHandler
public async notify : () ==> ()
notify == ...
end ButtonPressInterruptHandler
```

Listing 1.2. VDM `ButtonPressInterruptHandler` example for Button Press

We propose that interrupt handlers are to be registered on a CPU and not in a system. All interrupt handlers are then to be executed on the CPU they are bound to. The CPU can then be extended with an operation to register the interrupt handler such as:
`public regIntHandler : InterruptHandler * int ==> ()`
An interrupt handler should then be able to be registered by instantiation of the handler

class and then deployed to the CPU as shown in Listing 1.3 where an instance of the button press interrupt handler is registered on `cpu1`.

```
system S
...
cpu1 : CPU;
handler : ButtonPressInterruptHandler;
... -- inside the constructor
cpu1.regIntHandler(handler,MAX_PRIORITY);
...
end S
```

Listing 1.3. An example showing how an interrupt hander can be registered on a CPU.

Note that an alternative approach, involving a syntax change to the VDM RT notation, was proposed in [Ver05].

4.2 Measurement of time in VDM-RT

In VDM-RT all expressions and statements have a predefined default execution time specified which is set to 2 time units for all expressions and statements. The semantics defines two constructs to explicitly override the default durations at runtime: durations and cycles. Duration statements define the number of time units the enclosing statement takes to override any inner (default) durations. The cycle statement defines the number of cycles the enclosing statement takes, the time units can be derived by dividing the number of cycles by the CPU speed+1.

However, defining time as time units is not adequate to enable a co-simulation with a continuous time simulator, since time must be synchronised between the two simulation engines for the outcome of the simulation to be valid. The architecture of the VDM-RT controller will not correspond to reality if synchronisation of time is wrong, since it will affect the calculation speed of the architecture of the VDM-RT controller.

We suggest to give time steps a unit. This will enable synchronisation with a continuous time simulator. If such a unit is chosen to be *[s] seconds* then we would also suggest that the capacity of CPUs is changed into calculations per seconds specified in *[Hz]* and BUS rates to be specified in *[bit/s]*. Duration statements should also use seconds to specify how long the enclosing statement takes. Alternative, if no time unit is chosen but a clear connection between CPUs, BUSes, duration and cycle statements is defined, then a mapping between the VDM-RT model time and the connected continuous time co-simulator notion of time can be defined as a parameter to the VDM-RT interpreter (mapping logical time units in VDM-RT into real-time and vice versa).

Lastly, we would also suggest to change default durations for all expressions and statements from durations into cycles, this way a model's timing will change if the CPU capacity changes. Furthermore, each expression and statement should have individual cycles specified. The cycles of an assignment expression given by `x := y + z` can be calculated by breaking it down into low level commands as shown in Table 1 where the `mov` move command is used to move `y` and `z` into two registries and then `cmp` is used to combine them and move them to `x`. This is 4 cycles in total for the assignment.

If this assignment was to be executed on a CPU running at 2 KHz, performing 2000 cycles / second then the assignment would take 0.002 second to complete.

Cycles of VDM assignment :=	
Instruction	Description
`mov`	Move y to registry
`mov`	Move z to registry
`cmp`	Combine the two registries
`mov`	Move result to x
4	Total cycles

Table 1. Calculation of VDM equals by use of assembly instructions.

4.3 Limitation of Co-Simulation interface

The co-simulation interface is limited to support a small set of types understood by both discrete time and continuous time simulators. This set will consist of types such as booleans, integers and doubles/reals no complex structures can be transferred. However, a common way of handling decimal numbers is required so simulation engines will be able to compare decimal numbers across the interface. A question remaining about restricted types e.g. an integer with an invariant attached. Should this be enabled and when should they then be checked?

5 Future Work

In addition to the issues raised above there are a number of additional desirable extensions to VDM-RT that we think are worth consideration in the future. When semantics work for VDM-RT is undertaken we propose that such possible language extensions are also considered for their semantics consequences for including each of them can be based on a proper semantic foundation when a decision is to be taken by the VDM-10 language board. The desirable extensions include:

- As indicated in Section 3.3 it is necessary to have multiple virtual CPUs in order to specify fully time independent input stimuli in the environment. Essentially a new virtual CPU is assigned to each new instance which is created outside the the scope of the `system` class. This would have the advantage that the threads executing on objects deployed at the virtual CPU would be fully independent of each other from a timing perspective.
- Enabling the use of the **`time`** keyword in permission predicates and pre and post conditions for operations. This would give more expressive power to the VDM modeller but it is so far unclear what the semantic consequences would be and how it would impact the performance of the interpreter.

– Identically, history counters (as used in permission predicates) should be extended with a notion of time. For example, `#age` which would return the amount of time passed since the last `#req` was passed. This could be used to specify response time requirements as a precondition, or maximum operation elapse times as a postcondition. For more examples, see [Ver05].
– As indicated before, add the ability to specify so-called sporadic periodic threads, for example using the concrete syntax

```
threads sporadic (d) drinkWine
```

where d only specifies the minimum interarrival time between two periodic calls to `drinkWine`.
– Include probabilistic capabilities in the semantics for both periodic (and sporadic) threads and duration statements. For example, we could write

```
duration(10, 100) drinkBeer();
```

to specify that the execution of the operation `drinkBeer` would take an arbitrary amount of time chosen from the $[10, ..., 100]$ interval. Arbitrary could mean that the strategy used is specified as a global parameter at simulation level (i.e. a random value taken from a normal or exponential distribution). As an extension, one could even explicitly specify how the value from the interval is taken by adding an extra parameter, which refers to an algorithm, taking the two bounds as input parameters, which on return computes the value. This would for example allow the ability to specify caching behaviour (first call is expensive, next calls are cheap). The same strategy could be followed for the looseness provided in periodic threads with jitter or sporadic threads. However evaluation of the algorithm must be done without affecting the timing of the model in which it is declared.
– The current system class forces a static topology of the system components. It is worthwhile analyzsng what the semantic consequences would be for enabling a dynamic deployment architecture [Nie10].
– What about including multiple BUSes between two CPUs and if that is specified how shall the routing of such messages be decided upon?
– Shall it be possible to let the user set a limit for buffer size for the messages received at a CPU? Currently, when a remote call is attempted, the caller is not blocked when the BUS is busy. It assumes that there exists an infinite buffer between the BUS and each CPU. Detecting design bottlenecks is therefore potentially hampered because these limits do exist in reality. Perhaps the buffer size should be a global simulator setting, or a configurable item per CPU or per CPU/BUS interconnection.
– Would it be desired to introduce a new construct to express *broadcast* messages, allowing a model to contain public static broadcast operations which can be picked up from any class on any CPU? If introduced should it then be a new keyword **`broadcast`** or a new use of **`static`** and **`async`**.
– What would the semantics be if we had multiple system classes in order to describe system of systems?
– What are the semantics with VDM exceptions (ie. errors) thrown in inter-CPU calls? In particular, what happens to exceptions raised by async calls? Do they propagate over the physical CPU border? What happens with synchronous calls

across a CPU boundary? What if the message gets lost on the bus (relevant in the context of the DESTECS project)? Does this lead to an exception on the caller CPU?
- We do not have explicit definitions of the scheduling policies. In fact some of the names of these are misleading. How shall it be possible to define more of these in the future both for CPUs and BUSes? Can they be specified in a late bound fashion, as part of the model, rather than as a built-in part of the language?

6 Concluding Remarks

This paper have pointed out a number of semantics issues that needs to be solved for the VDM-RT language. The plan for the future is to get an agreement in the Overture Language Board about what the semantics shall be like and then define the semantics of the VDM-RT extensions in relation to the ISO VDM-SL standard. This will most likely be in the same operational style as used in [Ver08] but inspiration will also be taken from other languages with similar constructs such as Circus with time and resources [SCJS10]. Similarly, for the probabilistic extensions, inspiration can be gained from the MODEST language and supporting tools [HH09]. With respect to the definition of the VDM RT semantics, one could for example break up the work into tasks such as:

- Define core abstract syntax (CAS) for VDM
- Define mappings from VDM dialects to CAS
- Define CAS semantics

Here the semantics could be split up into areas of interest (e.g. concurrency scheduling, object orientation, time and distribution/deployment).

At the practical level the plan is that Kenneth Lausdahl and Peter Gorm Larsen will spend a week at York University with Jim Woodcock at the beginning of November 2010 to work on this semantic definition. We hope that more volunteers are interested in taking part in this semantics effort.

Acknowledgements

The work reported in this paper have partly been carried out in the DESTECS project which have partially been funded by the European Commission. In addition, we would like to thank Nick Battle for proof reading a draft of this paper.

References

[BLV+10] J. F. Broenink, P. G. Larsen, M. Verhoef, C. Kleijn, D. Jovanovic, K. Pierce, and Wouters F. Design support and tooling for dependable embedded control software. In *Proceedings of Serene 2010 International Workshop on Software Engineering for Resilient Systems*. ACM, April 2010.

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

[FLS08] John Fitzgerald, Peter Gorm Larsen, and Shin Sahara. VDMTools: Advances in Support for Formal Modeling in VDM. *ACM Sigplan Notices*, 43(2):3–11, February 2008.

[FLV08] J. S. Fitzgerald, P. G. Larsen, and M. Verhoef. Vienna Development Method. *Wiley Encyclopedia of Computer Science and Engineering*, 2008. edited by Benjamin Wah, John Wiley & Sons, Inc.

[HH09] Arnd Hartmanns and Holger Hermanns. A modest approach to checking probabilistic timed automata. In *QEST*. IEEE Computer Society, September 2009.

[LBF+10] Peter Gorm Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl, and Marcel Verhoef. The Overture Initiative – Integrating Tools for VDM. *ACM Software Engineering Notes*, 35(1), January 2010.

[Luc87] Peter Lucas. VDM: Origins, Hopes, and Achievements. In Airchinnigh Bjørner, Jones and Neuhold, editors, *VDM '87 VDM – A Formal Method at Work*, pages 1–18. VDM-Europe, Springer-Verlag LNCS 252, 1987.

[MBD+00] Paul Mukherjee, Fabien Bousquet, Jérôme Delabre, Stephen Paynter, and Peter Gorm Larsen. Exploring Timing Properties Using VDM++ on an Industrial Application. In J.C. Bicarregui and J.S. Fitzgerald, editors, *Proceedings of the Second VDM Workshop*, September 2000. Available at www.vdmportal.org.

[Nie10] Claus Ballegaard Nielsen. Towards dynamic reconfiguration of distributed systems in vdm-rt. In *Semantic Issues in VDM: a BCS-FACS and Overture Workshop*, September 2010.

[P. 96] 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.

[PT92] Nico Plat and Hans Toetenel. A formal transformation from the BSI/VDM-SL concrete syntax to the core abstract syntax. Technical Report 92-07, Delft University, March 1992.

[SCJS10] Adnan Sherif, Ana Cavalcanti, He Jifeng, and Augusto Sampaio. A process algebraic framework for specification and validation of real-time systems. *Formal Aspects of Computing*, 22:153–191, 2010.

[Ver05] Marcel Verhoef. On the use of VDM++ for specifying real-time systems. *Proc. First Overture workshop*, November 2005.

[Ver08] Marcel Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. PhD thesis, Radboud University Nijmegen, 2008. ISBN 978-90-9023705-3.

[VLH06] Marcel Verhoef, Peter Gorm Larsen, and Jozef Hooman. Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods*, pages 147–162. Lecture Notes in Computer Science 4085, 2006.

[WTVL04] Ernesto Wandeler, Lothar Thiele, Marcel Verhoef, and Paul Lieverse. System Architecture Evaluation Using Modular Performance Analysis – A Case Study. In Tiziana Margaria, Bernard Steffen, Anna Philippou, and Manfred Reitenspiess, editors, *Proc. 1st International Symposium On Leveraging Applications of Formal Methods (ISOLA 2004)*, pages 209–220, Paphos, Cyprus, November 2004. University of Paphos, Department of Computer Science. Also available at *http://www.cs.ru.nl/research/reports/info/ICIS-R05005.html*.