Formal Semantics of a VDM Extension for Distributed Embedded Systems*

Jozef Hooman[1,2] and Marcel Verhoef[3]

[1] Embedded Systems Institute, Eindhoven, The Netherlands
[2] Radboud University Nijmegen, The Netherlands
hooman@cs.ru.nl
[3] Chess, P.O. Box 5021, 2000 CA Haarlem, The Netherlands.
Marcel.Verhoef@chess.nl

To appear in de Roever Festschrift, LNCS 5930, pp. 142-161, Springer-Verlag, 2010.

Abstract. To support model-based development and analysis of embedded systems, the specification language VDM++ has been extended with asynchronous communication and improved timing primitives. In addition, we have defined an interface for the co-simulation of a VDM++ model with a continuous-time model of its environment. This enables multi-disciplinary design space exploration and continuous validation of design decisions throughout the development process. We present an operational semantics which formalizes the precise meaning of the VDM extensions and the co-simulation concept.

1 Introduction

We present a formal semantics of an extension of the VDM language for the development of software in embedded systems. Examples of embedded systems can be found, for instance, in airplanes, cars, industrial robots, process automation, and consumer electronics devices. In general, the development of such systems is extremely complex. In the early design phases a trade-off has to be made between a large number of design choices. This concerns, for instance, aspects such as mechanical lay-out, accuracy and placement of sensors, types of actuators, processing units, communication infrastructure, software deployment, and costs. This leads to an enormous design space which involves several disciplines, such as mechanical engineering, electrical engineering, and software engineering.

To support multi-disciplinary design space exploration, we propose an approach were each discipline can use its own modeling method, including all discipline-specific simulation and analysis techniques. But by defining a proper notion of co-simulation, which allows simultaneous simulation of discrete-time and continuous-time models, additional insight in the multi-disciplinary interactions and interferences can be obtained. E.g., in [1] we have coupled Rose Real-Time and Matlab/Simulink [2] to allow co-simulation of a UML model of control software with a Matlab/Simulink model of

* 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 Dutch Ministry of Economic Affairs under the Senter TS program.

the continuous behaviour of its environment. To realize this coupling, the platform-dependent notion of time in Rose Real-Time had to be replaced by a mechanism which obtains a notion of simulated time from Simulink. While this is a step forward, it also shows that Rose Real-Time is not very suitable for the co-simulation of control systems because it lacks a suitable notion of simulation time. Moreover, the run-to-completion semantics does not allow interrupts due to relevant events of the physical system under control.

We have also investigated support for design choices concerning the hardware infrastructure and the software-to-hardware mapping. Several methods have been applied to an in-car radio navigation system [3]. Each method has been used to model the relevant aspects of the system such as the characteristics of the processors, their connections, the input from the environment, the processor usage of the embedded software for this input, and the bandwidth needed for communication between processors. These models have been used to simulate and to analyze the performance of a particular deployment of software on hardware. We observed that already with relatively simple models and some sensitivity analysis, a good insight in design choices and performance bottlenecks can be obtained. However, it was also clear that most techniques do not immediately fit into the normal development process. For instance, the abstract models used for software analysis are not suitable for further software development.

On the other hand, development languages that allow the refinement of high-level abstract models towards concrete code, usually do not support typical aspects of embedded systems such as deployment on hardware and performance analysis. For instance, we have investigated the use of the formal language VDM++ for the software development of embedded systems by applying it to the control of the paper path in a printer. The VDM tools offer nice simulation possibilities and allow code generation, but the case study also revealed some limitations, especially concerning the expressiveness of the language to describe distributed real-time systems. In addition, important tool features are missing to analyze models of embedded systems [4].

In this paper, we integrate a number of the lessons learned from the research described above, aiming at a software modeling technique which allows the specification and the analysis of embedded systems at various levels of abstraction. This should include timing and deployment aspects. Moreover, the technique should allow executable specifications with a proper notion of simulation time to enable a coupling with continuous-time models. Given our VDM++ experience and access to supporting tools, we use the VDM language as our starting point. Since VDM extensions can be specified in VDM++ itself, models in the extended language can be executed and we can experiment with them in case studies.

The main contributions of our work are (1) an extension of VDM++ which allows the specification of the mapping of software task onto processing units, (2) an extension with additional timing primitives and a revision of the timing semantics of VDM++, (3) the definition of a notion of co-simulation of a VDM++ model with a continuous-time model, and (4) a formal semantics of the new concepts. Work on (1) and (2) has been described before in [5] and a first description of (3) can be found in [6]. New in this paper is the integration of these concepts and the definition of an integrated formal

operational semantics. The operational semantics has been formalized using the proof tool PVS [7,8].

All our extensions have been specified in VDM++ and, hence, extended models can be executed with the standard VDM tools. Simulation has been used to validate our extensions in a few case studies. To experiment with co-simulation, we have coupled the VDM tool to 20-SIM [9], a tool to model and simulate the behavior of dynamic systems, such as electrical, mechanical or hydraulic systems. The main ideas of our approach, however, are rather independent from VDM. The same holds for the abstract formal semantics which defines the precise meaning of the main concepts.

Related to our formal semantics is work in the context of UML about the precise meaning of active objects, with communication via signals and synchronous operations, and threads of control. In [10] a labeled transition system has been defined using the algebraic specification language CASL, whereas [11] uses the specification language of the theorem prover PVS to formulate the semantics. Note that UML 2.0 adopts the run-to-completion semantics, which means that new signals or operation calls can only be accepted by an object if it cannot do any local action, i.e., it can only proceed by accepting a signal or call. In our VDM++ semantics there will be less restrictions on threads. In addition, none of these works deal with deployments. Related to that aspect is the UML Profile for Schedulability, Performance and Time, and research on performance analysis based on this profile [12].

Concerning our definition of co-simulation, there is an interesting relation with the software architecture for the design of continuous time / discrete event co-simulation tools described in [13]. An operational semantics has been defined in [14]. The main difference is that their approach aims at connecting multiple simulators on a so-called simulation bus, whereas we connect two simulators using a point-to-point connection. They use Simulink and SystemC, whereas we use 20-SIM and VDM++ to demonstrate the concept. The type of information exchanged over the interfaces is identical (the state of continuous variables and events). They have used formal techniques to model properties of the interface, whereas we have integrated the continuous-time interface into the operational semantics of a discrete event system.

This paper is structured as follows. Section 2 provides a brief overview of VDM and the limitations of the current version of timed VDM++. Our proposed extensions for the specification of embedded systems are described in Sect. 3. In Sect. 4 we define an abstract syntax which illustrates the new concepts. This syntax is used to define a formal operational semantics of the proposed changes in Sect. 5. Concluding remarks can be found in Sect. 6.

2 Overview of VDM++

VDM++ is an object-oriented and model-based specification language with a formally defined static and dynamic semantics. It is a superset of the ISO standardized notation VDM-SL [15]. Different VDM dialects are supported by industry-strength tools, called VDMTOOLS [16]. The language has been used in several large-scale industrial projects [17,18,19]. However, not much is known about the application of VDM in the area of distributed real-time embedded systems.

The dynamic semantics of an executable subset of VDM++ is defined as a constructive operational semantics in VDM-SL [20]. The core of this specification is an abstract state machine which is able to execute a set of formally defined primitive instructions. Each abstract syntax element is translated into such a sequence of primitive instructions. The industrial success of VDMTOOLS is, for a large part, due to excellent conformance of the tool to the formally defined operational semantics and the round-trip engineering with UML.

A brief overview of the VDM++ language can be found in Sect. 2.1. For an in-depth presentation of the language and supporting tools, we refer to [19]. The main limitations for the specification of embedded systems are described in Sect. 2.2.

2.1 The basic VDM++ notation

In VDM++, a model consists of a collection of class specifications. We distinguish active and passive classes. Active classes represent entities that have their own thread of control and do not need external triggers in order to work. In contrast, passive classes are always manipulated from the thread of control of another active class. We use the term object to denote the instance of a class. More than one instance of a class might exist. An instance is created using the **new** operator, which returns an object reference. A class specification has the following components:

Class header: The header contains the class name declaration and inheritance information. Both single and multiple inheritance are supported.

Instance variables: The state of an object consists of a set of typed variables, which can be of a simple type such as *bool* or *nat*, or complex types such as sets, sequences, maps, tuples, records and object references. The latter are used to specify relations between classes. Instance variables can have invariants and an expression to define the initial state.

Operations: Class methods that may modify the state can be defined implicitly, using pre- and postcondition expressions only, or explicitly, using imperative statements and optional pre- and postcondition expressions.

Functions: Functions are similar to operations except that the body of a function is an expression rather than an imperative statement. Functions are not allowed to refer to instance variables, they are pure and side-effect free.

Synchronization: Operations in VDM++ are synchronous, that is, the caller of an operation waits until the execution of the operation body has been completed.

Thread: A class can be made "active" by specifying a thread. A thread is a sequence of statements which are executed to completion at which point the thread dies. It is possible to specify threads that never terminate.

A timed extension to VDM++ was defined as part of the VICE project [21] by assigning a user-configurable default duration to each basic language construct. In addition, there are two new statements:

Duration The duration statement, with the concrete syntax **duration**(d) *IS*, expresses that first all statements in the instruction sequence *IS* are executed instantaneously and next time is increased by d time units. The duration statement is used to override the default execution time for *IS*.

Period The periodic statement, with the concrete syntax **periodic**$(d)(Op)$, can only be used in the thread clause to denote that operation Op is called periodically every d time units.

2.2 The limitations of (timed) VDM++

In previous work [4], we assessed the suitability of timed VDM++ for distributed real-time embedded systems. We list the most important problems here.

1. Operations in VDM++ are synchronous; calls are executed in the context of the thread of control of the caller. The caller has to wait until the operation is completed before it can resume. This is very cumbersome when embedded systems are modeled. These systems are typically reactive by nature and asynchronous. An event loop can be specified to describe this, but this increases the complexity of the model and its analysis. See also the discussion about the advantages of asynchronous communication in [22].
2. Timed VDM++ is based on a uni-processor multi-threading model of computation. This means that at most one thread can claim the processor and only this active thread can enable progress of time. This is insufficient for describing embedded systems because 1) they are often implemented on a distributed architecture and 2) these systems need to be described in combination with their environment. Hence, we need a multi-processor multi-threading model of computation with parallel progress of time in subsystems and the environment.
3. The duration statement in timed VDM++ denotes a time penalty that is independent of the resource that executes the statement. When deployment is considered, it is essential to also be able to express time penalties that are relative to the capacity of the computation resource. Furthermore, there should be an additional time penalty that reflects the message handling between two computation resources whenever a remote operation call is performed.

In addition, we would like to simulate a VDM model in parallel with a (possibly continuous) model of its environment. This is related to the first two points mentioned above, but also requires a description of the interface between both models and a definition of the semantics of co-simulation.

3 Proposed Extensions for Embedded Systems

We briefly list our proposed changes and illustrate the main concepts by pieces of two examples. The formal semantics will be given in Sect. 5. The five main changes are:

1. The addition of asynchronous communication by introducing the **async** keyword in the signature of an operation to denote that it is asynchronous. For each call of an asynchronous operation a new thread is created which executes the body of the operation. The caller is not blocked and can immediately resume its own thread of control after the call has been initiated.

2. The introduction of a notion of deployment which can be used to specify the allocation of software tasks on processors and the communication topology between processors. Predefined classes, *BUS* and *CPU*, are made available to the specifier to construct the distributed architecture. Instance of these classes can be used to specify particular resources with a specific capacity, policy and overhead. Classes representing software tasks can be instantiated and deployed on a specific *CPU* in the model. The communication topology between the computation resources in the model can be described using the *BUS* class. The *system* class is used to contain such an architecture model.
3. The timing semantics of VDM has been adapted to allow multiple threads running on different processors. Any thread that is running on a computation resource or any message that is in transit on a communication resource can cause time to elapse. Models that contain only one computation resource are compatible to models in the original version of timed VDM++.
4. The introduction of a **cycles** statement to express the duration of statements in terms of cpu cycles. It can be used to denote a time delay that is relative to the capacity of the resource on which the the statement is executed.
5. To enable co-simulation, an XML configuration file can be used to describe the interface between two models, e.g., a VDM model and a model of it environment in another tool such as 20-SIM or Matlab/Simulink. It consists of arrays to define and relate sensors, actuators and events.

3.1 Examples of the extensions

In-car radio navigation system We show how a number of extensions are used in the in-car radio navigation system [3]. The main aim of this case study was to investigate which hardware topology of processors and interconnections could be used for a number of software tasks, such that certain timing deadlines would be satisfied. The three main applications, radio, navigation, and Man-Machine Interaction (MMI), have been specified in our extended version of VDM.

As an example, Fig. 1 shows the *Radio* class which has two asynchronous operations: `AdjustVolume` and `HandleTMC`.

```
class Radio
operations
  async public AdjustVolume: nat ==> ()
  AdjustVolume (pno) ==
    ( duration (150) skip; RadNavSys`mmi.UpdateVolume(pno) );

  async public HandleTMC: nat ==> ()
  HandleTMC (pno) ==
    ( cycles (1E5) skip; RadNavSys`navigation.DecodeTMC(pno) )
end Radio
```

Fig. 1. The *Radio* class

Since we are mainly interested in the overall timing behaviour of the system, we use the **skip** statement to represent internal computations. In the *AdjustVolume* operation

we use, for illustration purposes, the **duration** statement to express that this internal computation (e.g., changing the amplifier volume set point) takes 150 time units. The *HandleTMC* operation, which deals with Traffic Message Channel (TMC) messages, uses the **cycles** statement to denote that a certain amount of time expires relative to the capacity of the computation resource on which it is deployed. If this operation is deployed on a resource that can deliver 1000 cycles per unit of time then the delay (duration) would be 1E5 divided by 1000 is 100 time units. A suitable unit of time can be selected by the modeler.

Similarly, classes *Navigation* and *MMI* have been specified. The complete system model is presented in Fig. 2 where instances of the three classes are allocated on different processors. In this case, each instance is deployed on a separate *CPU*. Each computation resource is characterized by its processing capacity, specified by the number of available cycles per unit of time, the scheduling policy that is used to determine the task execution order and a factor to denote the overhead incurred per task switch. For this case study, fixed priority preemptive scheduling (denoted by <FP>) with zero overhead is used.

```
system RadNavSys
 instance
 variables
  -- create the application tasks
  static public mmi := new MMI();
  static public radio := new Radio();
  static public navigation := new Navigation();

  -- create CPU (policy, capacity, task switch overhead)
  CPU1 : CPU := new CPU(<FP>, 22E6, 0);
  CPU2 : CPU := new CPU(<FP>, 11E6, 0);
  CPU3 : CPU := new CPU(<FP>, 113E6, 0);

  -- create BUS (policy, capacity, message overhead, topology)
  BUS1 : BUS := new BUS(<FCFS>, 72E3, 0, {CPU1, CPU2, CPU3})

 operations
  -- the constructor of the system model
  public RadNavSys: () ==> RadNavSys
  RadNavSys () ==
     ( CPU1.deploy(mmi);          -- deploy MMI on CPU1
       CPU2.deploy(radio);        -- deploy Radio on CPU2
       CPU3.deploy(navigation) ) -- deploy Navigation on CPU3
end RadNavSys
```

Fig. 2. The top-level system model for the in-car radio navigation system

Class *BUS* is used to specify the communication resources in the system. A communication resource is characterized by (1) the scheduling policy that is used to determine the order of the messages being exchanged, (2) its throughput, specified by the number of messages that can be handled per unit of time, (3) a time penalty to denote the

protocol overhead, and (4) the computation resources it connects. The granularity of a message can be determined by the user. For example, it can represent a single byte or a complete Ethernet frame, whatever is most appropriate for the problem under study. For this case study, we use First Come First Served scheduling (denoted by <FCFS>) with zero overhead.

Water level control The water level control example [6] illustrates the interaction between a software model in VDM and a model of its continuous environment in 20-SIM. The 20-SIM tool is used to model the relevant dynamic behavior of a water tank (using so-called bond graphs), describing how the water level depends on the input flow, the drain, and whether a valve is open or closed. Such models can be simulated using so-called solvers, which are implementations of numerical integration techniques that approximate the solution of a set of differential equations. Depending on the type of model, these solvers may use a fixed time step or a variable step size. In the last case, the step size may change in time, depending on the dynamics of the system, the required accuracy, and the required detection of certain events such as a variable crossing a certain border.

The time-triggered control of such a continuous-time model can be easily expressed in our extension of VDM, as shown in Fig 3, where `level` is a shared continuous variable that represents the height of the water level in the tank. Shared variable `valve` is used to change the state of the valve. The **periodic** clause states that the operation *loop* is called periodically, namely once per second.

```
class TimeBasedController

instance variables
   static public level : real;
   static public valve : bool := false  -- default is closed

operations
   loop: () ==> ()
   loop () ==
   if level >= 3 then valve := true
   else if level <= 2 then valve := false;

threads
periodic(1.0)(loop)

end TimeBasedController
```

Fig. 3. Time-based controller description in VDM++

An alternative is event-based control, where the continuous plant generates relevant events, e.g., when the water level has reached certain limits. As an example. let `High(level,3.0)` be the event generated by the solver of the plant when the water level is increasing and reaches value `3.0`. Similarly, event `Low(level,2.0)` is generated when the level is decreasing and reaches level `2.0`.

The event handlers for these events are specified in VDM as asynchronous operations, *open* and *close*, as shown in Fig 4. The **duration** statement in the definition of *open* states that opening the valve in this case takes 50 msec. The **cycles** statement in the definition of *close* denotes that closing the valve takes 1000 cycles. Assuming this class is deployed on a processor with a capacity of 100000 cycles per second, then executing `valve :=` **false** will take 10 msec. Note that the result of the assignment is only available after this time has passed. Finally, the **mutex** clauses state that the operations are declared mutually exclusive, that is, only one operation call can be active at any time. Hence, the execution of the body of an operation cannot be interrupted by the execution of any other operation body.

```
class EventBasedController

instance variables
   static public level : real;
   static public valve : bool := false  -- default is closed

operations
   static public async open: () ==> ()
   open () == duration(0.05) valve := true;

   static public async close: () ==> ()
   close () == cycles(1000) valve := false;

sync
   mutex(open, close);
   mutex(open);
   mutex(close)

end EventBasedController
```

Fig. 4. The event-based controller description in VDM++

The relation between the events generated by the plant simulation and the corresponding handlers in VDM is defined in a separate XML configuration file. For brevity, we use an informal description as presented in Fig. 5. Sensor output of the plant model is bound to the `level` variable in the VDM model. Similarly, VDM variable `valve` provides actuator input to the plant. Furthermore, the `open` and `close` operations are defined as the handlers for the `High(level,3.0)` and `Low(level,2.0)` events. In other words, these asynchronous operations will be called automatically whenever the corresponding event fires. This will cause the creation of a new thread which will die as soon as the operation is completed.

```
sensor[1] = cpu1.EventBasedController`level
actuator[1] = cpu1.EventBasedController`valve
event[1] = High(level,3.0) -> cpu1.EventBasedController`open
event[2] = Low(level,2.0) -> cpu1.EventBasedController`close
```

Fig. 5. The interface configuration file

4 Syntax and Informal Semantics

To be able to highlight the formal semantics of the extensions proposed in the previous section, we define a syntax which abstracts from many aspects and constructs in VDM++. Our syntax does not contain class definitions and explicit definitions of synchronous and asynchronous operations. Assume given a set *Operation* of operations, with typical element *op*, and predicate *syn?*(op) which is true iff the operation is synchronous. We also assume that the body of operations is compiled into a given sequence of instructions. Let *ObjectId* be the set of object identities, with typical element *oid*.

Assume given a set of variables *Var* $=$ *InVar* $\cup$ *OutVar* $\cup$ *LVar* where *InVar* is the set of input/sensor variables, *OutVar* is the set of output/actuator variables, and *LVar* a set of local variables. The input and output variables (also called IO-variables) are global and shared between all threads and the continuous model. Hence, they can also be accessed by the solver of a continuous model, which may read the actuator variables and write the sensor variables. Let *IOVar* $=$ *InVar* $\cup$ *OutVar*. Let *Value* be a domain of values, such as the integers.

Our time domain is the nonnegative real numbers; *Time* $= \{t \in \mathbb{R} \mid t \geq 0\}$. We use d to denote a time value and **duration** (d) as an abbreviation of **duration**(d) **skip**. Assume that, for an instruction sequence *IS*, the statement **duration**(d) *IS* is translated into *IS* $\hat{}$ **duration**(d), where internal durations inside *IS* have been removed and the "$\hat{}$" operator concatenates the duration instruction to the end of a sequence. The concatenation operation is also used to concatenate sequences and to add an instruction to the front of the sequence. Functions *head* and *tail* yield the first element and the rest of the sequence, resp., and $\langle\rangle$ denotes the empty sequence. The **cycles** statement has been omitted here since it is equivalent to a duration statement, given a certain deployment. The **periodic** statement has been generalized to allow the periodic execution of an instruction sequence instead of an operation call only.

The distributed architecture of an embedded control program can be represented by so-called nodes. Let *Node* be the set of node identities. Nodes are used to represent computation resources such as processors. On each node a number of concurrent threads are executed in an interleaved way. In addition, execution may be interleaved with steps of the solver. Function *node* : *Thread* $\rightarrow$ *Node* denotes on which node each thread is executing. Each thread executes a sequential program, that is, a statement expressed in the language of Table 1. Furthermore, assume given a set of links, defined as a relation between nodes, i.e., *Link* $=$ *Node* $\times$ *Node*, to express that messages can be transmitted from one node to another via a link. In the semantics described here, we assume for simplicity that a direct link exists between each pair of communicating nodes. Note that *CPU* and *BUS*, as used in the radio navigation case study, are concrete examples of a node and a link.

The solver may send events to the control program. Let *Event* be a set of events. Assume that an event handler has been defined for each event, i.e., an instruction sequence and a node on which this statement has to be executed (as a new thread), denoted by the function *evhdlr* : *Event* $\rightarrow$ *Instr. Seq.* $\times$ *Node*. The syntax of our sequential programming language is given in Table 1, with $c \in$ *Value*, $x \in$ *Var*, and $d \in$ *Time*.

These basic instructions have the following informal meaning:

– **skip** represents a local statement which does not consume any time.

Table 1. Syntax of Instructions

Value Expr.	$e ::= c \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2$
Bool Expr.	$b ::= e_1 = e_2 \mid e_1 < e_2 \mid \neg b \mid b_1 \vee b_2$
Instr.	$I ::=$ **skip** $\mid x := e \mid$ **call**$(oid, op) \mid$ **duration**$(d) \mid$ **periodic**(d) *IS* $\mid$ **if** b **then** IS_1 **else** IS_2 **fi** $\mid$ **while** b **do** *IS* **od**
Instr. Seq.	$IS ::= \langle\rangle \mid I \hat{} IS$

- $x := e$ assigns the value of expression e to x.
- **call**(oid, op) denotes a call to an operation *op* of object *oid*. Depending on the *syn?* predicate, the operation can be synchronous (i.e., the caller has to wait until the execution of the operation body has terminated) or asynchronous (the caller may continue with the next instruction and the operation body is executed independently). There are no restrictions on re-entrance here, but in general this can be restricted in VDM by so-called permission predicates. These are not considered here, also parameters are ignored.
- **duration**(d) represents a time progress of d time units. When d time units have elapsed the next statement can be executed.
- **periodic**(d) *IS* leads to the execution of instruction sequence *IS* each period of d time units.
- **if** b **then** IS_1 **else** IS_2 **fi** executes instruction sequence IS_1 if b evaluates to true and IS_2 otherwise.
- **while** b **do** *IS* **od** repeatedly executes instruction sequence *IS* as long as b evaluates to true.

The formalization of the precise meaning of the language described above raises a number of questions that have to answered and on which a decision has to be taken. We list the main points:

- How to deal with the combination of synchronous and asynchronous operations, e.g., does one has priority over the other, how are incoming call request recorded, is there a queue at the level of the node or for each object separately? We decided for an equal treatment of both concepts; each object has a single FIFO queue which contains both types of incoming call requests.
- How to deal with synchronous operation calls; are the call and its acceptance combined into a single step and does it make a difference if caller and callee are on different nodes? In our semantics, we distinguish between a call within a single node and a call to an operation of an object on another node.
  For a call between different nodes, a call message is transferred via a link to the queue of the callee; when this call request is dequeued at the callee, the operation body is executed in a separate thread and, upon completion, a return message is transmitted via the link to the node of the caller.
  For a call within a single node, we have made the choice to avoid a context switch and execute the operation body directly in the thread of the caller. Instead, we could have placed the call request in the queue of the callee.

– Similar questions hold for asynchronous operations. On a single node, the call request is put in the queue of the callee, whereas for different nodes the call is transferred via a link. However, no return message is needed and the caller may continue immediately after issuing the call.
– How are messages between nodes transferred by the links? In principle, many different communication mechanisms could be modeled. As a simple example, we model a link by a set of messages which include a lower and an upper bound on message delivery. For a link l, let $\delta_{min}(l)$ and $\delta_{max}(l)$ be the minimum and maximum transmission time. It is easy to extend this and, for instance, make the transmission time dependent on message size and link traffic.
– How to deal with time, how is the progress of time modeled? In our semantics, there is only one global step which models progress of time on all nodes. All other steps do not change time; all assumptions on the duration of statements, context switches and communications have to be modeled explicitly by means of duration statements.
– What is the effect of the interleaved execution of assignments to shared variables in different threads? As mentioned in the previous point, the execution of basic statements such as skip and assignment takes zero time. Hence, in our semantics any sequence of statements between two successive duration statements is executed atomically (in zero time). For instance, if we execute the instruction sequence $\textbf{duration}(1) \,\hat{}\, x := 1 \,\hat{}\, x := x + 1 \,\hat{}\, \textbf{duration}(1)$ in parallel with the sequence $\textbf{duration}(1) \,\hat{}\, x := 5 \,\hat{}\, y := x \,\hat{}\, \textbf{duration}(1)$ then there are two possible results; we might get $x = 5 \wedge y = 5$ or $x = 2 \wedge y = 5$. This in contrast with $\textbf{duration}(1) \,\hat{}\, x := 1 \,\hat{}\, \textbf{duration}(1) \,\hat{}\, x := x + 1 \,\hat{}\, \textbf{duration}(1)$ in parallel with $\textbf{duration}(1) \,\hat{}\, x := 5 \,\hat{}\, \textbf{duration}(1) \,\hat{}\, y := x \,\hat{}\, \textbf{duration}(1)$, where additionally $x = 2 \wedge y = 1$, $x = 2 \wedge y = 2$, $x = 6 \wedge y = 5$, and $x = 6 \wedge y = 6$ are possible.
– What is the precise meaning of $\textbf{periodic}(d)$ *IS* if the execution of *IS* takes more than d time units? We decided that after each d time units a new thread is started to ensure that every d time units the *IS* sequence can be executed. Of course, this might potentially lead to resource problems for particular applications, but this will become explicit during analysis.

5 Formal Operational Semantics

The operational semantics presented in this section defines the execution of the language given in Sect. 4 formally. To focus on the essential aspects, we assume that the set of objects is fixed and need not be recorded in the configuration. However, object creation can be added easily, see e.g. [11]. Threads can be created dynamically, e.g., to deal with asynchronous operations and events received from the solver. Let *Thread* be the set of thread identities, including dormant threads that can be made alive when a new thread is created. Each thread i is related to one object, denoted by o_i. This is used to define a new *node* function which defines the deployment of threads by means of the *node* function on objects: $\mathit{node}(i) = \mathit{node}(o_i)$.

Recall that any sequence of statements between two successive duration statements is executed atomically in zero time. However, the execution of such a sequence might

be interleaved with statements of other threads or a step of the solver. Concerning the shared IO-variables in *IOVar* this means that we have to ensure atomicity explicitly. To this end, we introduce a kind of *transaction* mechanism to guarantee consistency in the presence of arbitrary interleaving of steps. Thread i is only allowed to modify IO-variable x if there is no transaction in progress by any other thread. The transaction is committed as soon as the thread performs a time step.

Finally, we extend the set of instructions with an auxiliary statement **return**(i). This statement will be added during the executing at the end of the instruction sequence of a synchronous operation which has been called by thread i.

To capture the state of affairs at a certain point during the execution, we introduce a *configuration* (Def. 1). Next we define the possible steps from one configuration to another, denoted by $C \longrightarrow C'$ where C and C' are configurations (Def. 3). This finally leads to a set of runs of the form $C_0 \longrightarrow C_1 \longrightarrow C_2 \longrightarrow \ldots$ (Def. 9).

Definition 1 (Configuration). A *configuration* C contains the following fields:

- *instr* : *Thread* $\rightarrow$ *Instr. Seq.*
  which is a function which assigns a sequence of instructions to each thread.
- *curthr* : *Node* $\rightarrow$ *Thread*
  yields for each node the currently executing thread.
- *status* : *Thread* $\rightarrow \{dormant, alive, waiting\}$
  denotes the status of threads.
- *lval* : *LVar* $\times$ *Thread* $\rightarrow$ *Value*
  denotes the value of each local variable for each thread.
- *ioval* : *IOVar* $\rightarrow$ *Value*
  denotes the committed value of each sensor and actuator variable.
- *modif* : *IOVar* $\times$ *Thread* $\rightarrow$ *Value* $\cup \{\perp\}$
  denotes the values of sensor and actuator variables that have been modified by a thread and for which the transaction has not yet been committed (by executing a duration statement). The symbol $\perp$ denotes that the value is undefined, i.e., the thread did not modify the variable in a non-committed transaction.
- q : *ObjectId* $\rightarrow$ *queue*[*Thread* $\times$ *Operation*]
  records for each object a FIFO queue of incoming calls, together with the calling thread (needed for synchronous operations only).
- *linkset* : *Link* $\rightarrow$ *set*[*Message* $\times$ *Time* $\times$ *Time*]
  records the set of the incoming messages for each link, together with lower and upper bound on delivery. A message may denote a call of an operation (including calling thread and called object) or a return to a thread.
- *now* : *Time*
  denotes the current time.

For a FIFO queue, functions *head* and *tail* yield the head of the queue and the rest, respectively; *insert* is used to insert an element and $\langle\rangle$ denotes the empty queue. For sets we use *add* and *remove* to insert and remove elements. For a configuration C we use:

- $C(f)$ to obtain its field f. For example, $C(\mathit{instr})(i)$ yields the instruction sequence of thread i in configuration C.

– *exec*(C, i) as an abbreviation for $C(\mathit{curthr})(\mathit{node}(i)) = i$, which expresses that thread i is executing on its node.
– *fresh*(C, oid) to yield a fresh, not yet used, thread identity (so with status *dormant*) corresponding to object *oid*.

To express modifications of a configuration, we define the notion of a variant.

Definition 2 (Variant). The *variant* of a configuration C with respect to a field f and value v, denoted by $C[\ f \mapsto v\ ]$, is defined as

$$(C[\ f \mapsto v\ ])(f') = \begin{cases} v & \text{if } f' = f \\ C(f') & \text{if } f' \neq f \end{cases}$$

Similarly for parts of the fields, such as *instr*(i).

We define the value of an expression e in a configuration C which is evaluated in the context of a thread i, denoted by $[\![\ e\ ]\!](C, i)$. The main point is the evaluation of a variable, where for an IO-variable we use the *modif* field if there is an uncommitted change:

$$[\![\ x\ ]\!](C, i) = \begin{cases} C(\mathit{modif})(x, i) & \text{if } x \in \mathit{IOVar}, C(\mathit{modif})(x, i) \neq \bot \\ C(\mathit{ioval})(x) & \text{if } x \in \mathit{IOVar}, C(\mathit{modif})(x, i) = \bot \\ C(\mathit{lval})(x, i) & \text{if } x \in \mathit{LVar} \end{cases}$$

The other cases are trivial, e.g., $[\![\ e_1 \times e_2\ ]\!](C, i) = [\![\ e_1\ ]\!](C, i) \times [\![\ e_2\ ]\!](C, i)$ and $[\![\ c\ ]\!](C, i) = c$. It is also straightforward to define when a Boolean expression b holds in the context of thread i in configuration C, denoted by $[\![\ b\ ]\!](C, i)$. For instance, $[\![\ e_1 < e_2\ ]\!](C, i)$ iff $[\![\ e_1\ ]\!](C, i) < [\![\ e_2\ ]\!](C, i)$, and $[\![\ \neg b\ ]\!](C, i)$ iff not $[\![\ b\ ]\!](C, i)$.

Definition 3 (Step). $C \longrightarrow C'$ is a *step* if and only if it corresponds to the execution of an instruction (Def. 4), a time step (Def. 5), a context switch (Def. 6), the delivery of a message by a link (Def. 7), or the processing of a message from a queue (Def. 8).

Definition 4 (Execute Instruction). A step $C \longrightarrow C'$ corresponds to the execution of an instruction if and only if there exists a thread i such that *exec*(C, i) and $\mathit{head}(C(\mathit{instr})(i))$ is one of the following (underlined) instructions:

skip:
Then the new configuration equals the old one, except that the skip instruction is removed from the instruction sequence of i, that is,
$C' = C[\ \mathit{instr}(i) \mapsto \mathit{tail}(C(\mathit{instr})(i))\ ]$

$x := e$:
We distinguish two cases, depending on the type of variable x.

– If $x \in \mathit{IOVar}$ we require that there is no transaction in progress by any other thread, that is, for all i' with $i' \neq i$ we have $C(\mathit{modif})(x, i') = \bot$. Then the value of e is recorded in the modified field of i:
$C' = C[\mathit{instr}(i) \mapsto \mathit{tail}(C(\mathit{instr})(i)), \mathit{modif}(x, i) \mapsto [\![\ e\ ]\!](C, i)]$
As we will see later, all values belonging to thread i in C(*modif*) are removed and bound to the variables in $C(\mathit{ioval})$ as soon as thread i completes a time step (Def. 5). This corresponds to the intuition that the result of a computation is available only at the end of the time step that reflects the execution of a piece of code.

– If $x \in \mathit{LVar}$ then we change the value of x in the current thread:
$C' = C[\mathit{instr}(i) \mapsto \mathit{tail}(C(\mathit{instr})(i)), \mathit{lval}(x,i) \mapsto [\![\, e \,]\!](C,i)]$

call(oid, op):
Let *IS* be the explicit definition of operation *op* of object *oid*. We consider four cases:

– Caller and callee are on the same node, i.e. $node(i) = node(\mathit{oid})$.
- If *syn?*(op) then *IS* is executed directly in the thread of the caller:
$C' = C[\ \mathit{instr}(i) \mapsto \mathit{IS}\hat{}\,\mathit{tail}(C(\mathit{instr})(i))\ ]$
- If not *syn?*(op), we add the pair (i, op) to the queue of *oid*:
$C' = C[\ \mathit{instr}(i) \mapsto \mathit{tail}(C(\mathit{instr})(i)),$
$\quad\quad q(\mathit{oid}) \mapsto \mathit{insert}((i, op), C(q)(\mathit{oid}))\ ]$

– Caller and callee are on different nodes, i.e. $node(i) \neq node(\mathit{oid})$. Suppose link l connects these nodes. Then the call is transmitted via link l, which is represented by adding message $m = (\mathit{call}(i, \mathit{oid}, op), C(\mathit{now}) + \delta_{min}(l), C(\mathit{now}) + \delta_{max}(l))$ to the linkset of l.
- If *syn?*(op), thread i becomes *waiting*:
$C' = C[\ \mathit{instr}(i) \mapsto \mathit{tail}(C(\mathit{instr})(i)),\ \mathit{status}(i) \mapsto \mathit{waiting},$
$\quad\quad \mathit{linkset}(l) \mapsto \mathit{insert}(m, C(\mathit{linkset})(l))\ ]$
- Similarly for asynchronous operations, when not *syn?*(op), except that then the status of i is not changed:
$C' = C[\ \mathit{instr}(i) \mapsto \mathit{tail}(C(\mathit{instr})(i)),$
$\quad\quad \mathit{linkset}(l) \mapsto \mathit{insert}(m, C(\mathit{linkset})(l))\ ]$

duration(d):
A duration statement leads to global progress of time, including a time step in the solver of the continuous model of the environment. This time step will be defined in Def. 5.

periodic(d) *IS*:
In this case, *IS* is added to the instruction sequence of thread i and a new thread $j = \mathit{fresh}(C, o_i)$ is started which repeats the periodic instruction after a duration of d time units, i.e.
$C' = C[\ \mathit{instr}(i) \mapsto \mathit{IS},\ \mathit{instr}(j) \mapsto \textbf{duration}(d)\hat{}\,\textbf{periodic}(d)\ \mathit{IS},\ \mathit{status}(j) \mapsto \mathit{alive}\ ]$

if b **then** IS_1 **else** IS_2 **fi**

– If $[\![\, b \,]\!](C, i)$ then $C' = C[\mathit{instr}(i) \mapsto \mathit{IS}_1\hat{}\,\mathit{tail}(C(\mathit{instr})(i))]$
– Otherwise, $C' = C[\mathit{instr}(i) \mapsto \mathit{IS}_2\hat{}\,\mathit{tail}(C(\mathit{instr})(i))]$

while b **do** *IS* **od**:

– If $[\![\, b \,]\!](C, i)$ then
$C' = C[\mathit{instr}(i) \mapsto \mathit{IS}\hat{}\,\textbf{while}\ b\ \textbf{do}\ \mathit{IS}\ \textbf{od}\hat{}\,\mathit{tail}(C(\mathit{instr})(i))]$
– Otherwise, $C' = C[\mathit{instr}(i) \mapsto \mathit{tail}(C(\mathit{instr})(i))]$

return(j):
In this case we have $node(i) \neq node(j)$. Let l be the link which connects these nodes. Then $m = (\mathit{return}(j), C(\mathit{now}) + \delta_{min}(l), C(\mathit{now}) + \delta_{max}(l))$ is transmitted via l:
$C' = C[\ \mathit{instr}(i) \mapsto \mathit{tail}(C(\mathit{instr})(i)),\ \mathit{linkset}(l) \mapsto \mathit{insert}(m, C(\mathit{linkset})(l))\ ]$

Definition 5 (Time Step). A step $C \longrightarrow C'$ is called a *time step* only if all current threads are ready to execute a duration instruction or have terminated. More formally, for all i with *exec*(C, i), $C(\mathit{instr})(i)$ is $\langle\rangle$ or of the form **duration**$(d)\,\hat{}\,\mathit{IS}$. Then the definition of a time step consists of three parts: (1) the definition of the maximal duration of the time step as allowed by the VDM model, (2) the execution of a time step by the solver, leading to intermediate configuration C_s (3) updating all durations of all current threads, committing all variables of the current threads, and dealing with events generated by the solver.

1. Time may progress with t time units if
   - t is smaller or equal than all durations that are at the head of an instruction sequence of an executing thread, and
   - $C(\mathit{now}) + t$ is smaller or equal than all upper bounds of messages in link sets.

   Define the maximal length of the time step t_m as the largest t satisfying these conditions.

2. If $t_m > 0$ the solver tries to execute a time step of length t_m in configuration C. Concerning the variables, the solver will only use the *ioval* field, ignoring the *lval* and *modif* fields. It will only read the actuator variables in *OutVar* and it may write the sensor variables in *InVar* in field *ioval*. As soon as the solver generates one or more events, its execution is stopped. This leads to a new configuration C_s and a set of generated events *EventSet*. Since the solver takes a positive time step, we have $C(\mathit{now}) < C_s(\mathit{now}) \leq C(\mathit{now}) + t_m$. If $C_s(\mathit{now}) < C(\mathit{now}) + t_m$ then *EventSet* $\neq \emptyset$. Moreover, $C_s(f) = C(f)$ for field $f \in \{\mathit{instr}, \mathit{curthr}, \mathit{status}, \mathit{lval}, \mathit{modif}\}$.

   If $t_m = 0$ then the solver is not executed and $C_s = C$, *EventSet* $= \emptyset$. This case is possible because we allow **duration**(0) to commit variable changes, as shown in the next point.

3. Starting from configuration C_s and *EventSet*, next (a) the durations are decreased with the actual time step performed, leading to configuration C_d (b) transactions are committed for threads with zero durations, leading to configuration C_m, and (c) new threads are created for the event handlers, leading to final configuration C'.
   Let $t_s = C_s(\mathit{now}) - C(\mathit{now})$ be the time step realized by the solver.
   (a) Durations in instruction sequences are modified by the following definition which yields a new function from threads to instruction sequences, for any thread i,
   $$\mathit{NewDuration}(C, t_s)(i) = \begin{cases} \textbf{duration}(d_i - t_s)\,\hat{}\,\mathit{tail}(C(\mathit{instr})(i)) & \text{if } \mathit{head}(C(\mathit{instr})(i)) = \textbf{duration}(d_i) \\ C(\mathit{instr})(i) & \text{otherwise} \end{cases}$$
   Let $C_d = C_s[\ \mathit{instr} \mapsto \mathit{NewDuration}(C, t_s)]$

   (b) Let $\mathit{ThrDurZero}(C) = \{i \mid \mathit{exec}(C, i) \text{ and } \mathit{head}(C(\mathit{instr})(i)) = \textbf{duration}(0)\}$ be the set of threads with a zero duration. For these threads the transactions are committed and the values of the modified variables are finalized. This is

defined by two auxiliary functions:
NewIoval$(C)(x) =$

$$\begin{cases} v & \text{if } \exists\, i \in \textit{ThrDurZero}(C) \text{ and } C(\textit{modif})(x,i) = v \neq \bot \\ C(\textit{ioval})(x) & \text{otherwise} \end{cases}$$

Note that at any point in time at most one thread may modify the same global variable in a transaction. Hence, there exists at most one thread satisfying the first condition of the definition above, for a given variable x.
The next function resets the modified field, for any x and i,

$$\textit{NewModif}(C)(x,i) = \begin{cases} \bot & \text{if } i \in \textit{ThrDurZero}(C) \\ C(\textit{modif})(x,i) & \text{otherwise} \end{cases}$$

Then $C_m = C_d[\textit{ioval} \mapsto \textit{NewIoval}(C), \textit{modif} \mapsto \textit{NewModif}(C)]$

(c) For each event $e \in EventSet$ with *evhdlr*$(e) = (\textit{IS}_e, n_e)$, let i_e be a fresh - not yet used - thread identity with status *dormant* and *node*$(i_e) = n_e$. Then we define an auxiliary function *EventInstr*(C) : *Thread* $\rightarrow$ *Instr. Seq.* which installs event handlers. For any thread i,

$$\textit{EventInstr}(C)(i) = \begin{cases} \textit{IS}_e & \text{if } i = i_e \text{ for some } e \in \textit{EventSet} \\ C(\textit{instr})(i) & \text{otherwise} \end{cases}$$

In addition, we awake the threads of the event handlers by changing their status. Define, for any i,

$$\textit{NewStatus}(C)(i) = \begin{cases} \textit{alive} & \text{if } i = i_e \text{ for some } e \in \textit{EventSet} \\ C(\textit{status})(i) & \text{otherwise} \end{cases}$$

Then $C' = C_m[\textit{instr} \mapsto \textit{EventInstr}(C_m), \textit{status} \mapsto \textit{NewStatus}(C_m)]$

Observe that $C'(now) = C_s(now) = C(now) + t_s$ with $t_s \leq t_m$.

Definition 6 (Context Switch). A step $C \longrightarrow C'$ corresponds to a context switch iff there exists a thread i which is alive, not running, and has a non-empty program which does not start with a duration, i.e., $\neg \textit{exec}(C,i)$, $C(\textit{status})(i) = \textit{alive}$, $C(\textit{instr})(i) \neq \emptyset$, and $head(C(\textit{instr})(i)) \neq \textbf{duration}(d)$ for any d. Then i becomes the current thread and a duration of δ_{cs} time units is added to represent the context switching time:
$C' = C[\ \textit{instr}(i) \mapsto \textbf{duration}(\delta_{cs})\hat{}\, C(\textit{instr})(i), \textit{curthr}(\textit{node}(i)) \mapsto i\ ]$

Note that more than one thread may be eligible as the current thread on a node at a certain point in time. In that case, a thread is chosen nondeterministically in our operational semantics. Fairness constraints or a scheduling strategy may be added to reduce the set of possible execution sequences and to enforce a particular type of node behavior, such as round robin or priority-based pre-emptive scheduling.

Definition 7 (Deliver Link Message). A step $C \longrightarrow C'$ corresponds to the message delivery by a link iff there exists a link l and a triple (m, lb, ub) in $C(\textit{linkset})(l)$ with $lb \leq C(\textit{now}) \leq ub$. There are two possibilities for message m:

– *call*$(i, \textit{oid}, op)$: Insert the call in the queue of object *oid*:
$C' = C[\ q(\textit{oid}) \mapsto \textit{insert}((i, op), C(q)(\textit{oid})),$
$\textit{linkset}(l) \mapsto remove((m, lb, ub), C(\textit{linkset})(l))\ ]$

– *return*(i): Wake-up the caller, i.e.
$C' = C[\ status(i) \mapsto \textit{alive},\ \textit{linkset}(l) \mapsto remove((m, lb, ub), C(\textit{linkset})(l))\ ]$

Definition 8 (Process Queue Message). A step $C \longrightarrow C'$ corresponds to the processing of a message from a queue iff there exists an object *oid* with $\textit{head}(C(q)(\textit{oid})) = (i, \textit{op})$. Let $j = \textit{fresh}(C, \textit{oid})$ be a fresh thread and *IS* be the explicit definition of *op*. If the operation is synchronous, i.e. *syn?*(*op*), then we start a new thread with *IS* followed by a return to the caller:
$C' = C[\ instr(j) \mapsto \textit{IS}\hat{}\ \textbf{return}(i),\ status(j) \mapsto \textit{alive},\ q(\textit{oid}) \mapsto \textit{tail}(C(q)(\textit{oid}))\ ]$
Similarly for an asynchronous call, where no return instruction is added:
$C' = C[\ instr(j) \mapsto \textit{IS},\ status(j) \mapsto \textit{alive},\ q(\textit{oid}) \mapsto \textit{tail}(C(q)(\textit{oid}))\ ]$

Definition 9 (Operational Semantics). The operational semantics of a specification in the language of Sect. 4 is a set of execution sequences of the form $C_0 \longrightarrow C_1 \longrightarrow C_2 \longrightarrow \ldots$, where each pair $C_i \longrightarrow C_{i+1}$ is a step (Def. 3) and the initial configuration C_0 satisfies a number of constraints:

– no thread has status *waiting*,
– on each node, the currently executing thread is *alive*,
– a thread is dormant iff it has an empty execution sequence,
– the *modif* field is $\perp$ everywhere,
– all queues and link sets are empty, and
– the auxiliary instruction **return** does not occur in any instruction sequence.

To avoid Zeno behaviour, we require that for any point of time t there exists a configuration C_i in the sequence with $C_i(now) > t$.

6 Concluding Remarks

We have defined a formal operational semantics for an extension of VDM++ which supports the development of embedded systems. The semantics has been validated by formulating it in the typed higher-order logic of the verification system PVS [4] and verifying properties about it using the interactive theorem prover of PVS. In fact, the formal operational semantics presented in this chapter is based on a much larger constructive (and therefore executable) operational semantics of the extended language, which has been specified in VDM++ itself. This approach allows symbolic execution of models written in our extended language using the existing and unmodified tool set VDMTOOLS.

Besides the examples mentioned in Sect. 3.1, our extended VDM++ language has been applied to an experimental set-up that represents part of the paper path in a printer. This has been done in three phases:

1. In the first phase, the emphasis is on global system analysis by modeling and simulation using formal techniques. A model of the dynamic behavior of the physical system (with pinches, motors, and paper movement) was built using bond graphs, supported by the 20-SIM tool. The controller software has been modeled using

[4] The PVS files are available at http://www.cs.ru.nl/∼hooman/VDM4ES.html.

VDM++ and our extensions, supported by VDMTOOLS. The co-simulation of both models, as defined in this paper, has been used for multi-disciplinary design space exploration. This revealed a number of problems that were due to misunderstandings between designers of different disciplines and that would in normal industrial practice only have been detected in the integration phase.
2. In the second phase, the emphasis is on elaborating the software model of the control application. The level of detail in the VDM++ model has been increased incrementally, until source code could be generated from it automatically. The generated code has been compiled using a standard C++ compiler running on the simulator host, in our case a normal personal computer running on the Windows platform. The resulting dynamic link library (DLL) has been used for so-called software-in-the-loop simulations against the unmodified model of the plant in 20-SIM.
3. In the third phase, the unmodified C++ code generated from the VDM++ models developed in the second phase is compiled for the target platform. The resulting application has been uploaded to the embedded controllers of the experimental set-up for testing. It showed that the continuous validation during the process leads to high-quality code that meets the control objectives with high accuracy.

More details about this application can be found in [23].

Acknowledgments The first author would like to thank Willem-Paul de Roever for numerous reasons: for the invitation to start as a researcher in European project Descartes, for the opportunity to participate in subsequent international projects, for the collaboration on joint papers, and for many years of supervision, guidance and stimulation.

References

1. Hooman, J., Mulyar, N., Posta, L.: Coupling Simulink and UML models. In Schnieder, B., Tarnai, G., eds.: FORMS/FORMATS 2004. (2004) 304–311
2. The Mathworks: Matlab/Simulink. (2008) http://www.mathworks.com/.
3. Wandeler, E., Thiele, L., Verhoef, M., Lieverse, P.: System architecture evaluation using modular performance analysis: a case study. International Journal of Software Tools for Technology Transfer (STTT) **8**(6) (2006) 649–667
4. Verhoef, M.: On the use of VDM++ for specifying real-time systems. In Fitzgerald, J., Larsen, P.G., Plat, N., eds.: Towards Next Generation Tools for VDM: Contributions to the First International Overture Workshop. CS-TR 969, School of Computing Science, Newcastle University (June 2006) 26–43
5. Verhoef, M., Larsen, P.G., Hooman, J.: Modeling and validating distributed embedded real-time systems with VDM++. In: FM 2006: Formal Methods. Volume 4085 of Lecture Notes in Computer Science (LNCS)., Springer (2006) 147–162
6. Verhoef, M., Visser, P., Hooman, J., Broenink, J.: Co-simulation of distributed embedded real-time control systems. In: Integrated Formal Methods - IFM. Volume 4591 of Lecture Notes in Computer Science (LNCS)., Springer (2007) 639–658
7. Owre, S., Rushby, J., Shankar, N.: PVS: A prototype verification system. In: Conference on Automated Deduction, Lecture Notes in Artificial Intelligence 607, Springer (1992) 748–752
8. SRI International: PVS. (2008) http://pvs.csl.sri.com/.
9. Controllab Products: 20-sim. (2008) http://www.20sim.com/.

10. Reggio, G., Astesiano, E., Choppy, C., Hussmann, H.: Analysing UML active classes and associated statecharts - a lightweight formal approach. In: FASE 2000 - Fundamental Approaches to Software Engineering. Volume 1783 of Lecture Notes in Computer Science (LNCS)., Springer (2000) 127–146
11. Hooman, J., van der Zwaag, M.: A semantics of communicating reactive objects with timing. International Journal of Software Tools for Technology Transfer (STTT) **8**(4) (2006) 97–112
12. Bennet, A., Field, A.J., Woodside, M.C.: Experimental Evaluation of the UML Profile for Schedulability, Performance and Time. In: UML2004 - The Unified Modeling Language. Volume 3273 of Lecture Notes in Computer Science (LNCS)., Springer (2004) 143–157
13. Nicolescu, G., Boucheneb, H., Gheorghe, L., Bouchhima, F.: Methodology for efficient design of continuous/discrete-events co-simulation tools. In Anderson, J., Huntsinger, R., eds.: High Level Simulation Languages and Applications - HLSLA. SCS (2007) 172–179
14. Gheorghe, L., Bouchhima, F., Nicolescu, G., Boucheneb, H.: Formal definitions of simulation interfaces in a continuous/discrete co-simulation tool. In: Proc. IEEE Workshop on Rapid System Prototyping, IEEE Computer Society (2006) 186–192
15. Andrews, D., Larsen, P., Hansen, B., Brunn, H., Plat, N., Toetenel, H., Dawes, J., Parkin, G., et al.: Vienna Development Method Specification Language Part 1: Base Language. (1996) ISO/IEC 13817-1.
16. CSK Systems Corporation: VDMTOOLS. (2008) Free tool support can be obtained from http://www.vdmtools.jp/en/.
17. 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 – proceedings of the VDM workshop at FM'99. (1999) 85–93
18. Hörl, J., Aichernig, B.K.: Validating voice communication requirements using lightweight formal methods. IEEE Software **13–3** (2000) 21–27
19. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object-oriented Systems. Springer, New York Examples are available at http://www.vdmbook.com.
20. Larsen, P.G., Lassen, P.B.: An Executable Subset of Meta-IV with Loose Specification. In: VDM '91: Formal Software Development Methods, VDM Europe, Springer (1991) 604–618
21. 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)
22. Clarke, D., Johnsen, E.B., Owe, O.: Concurrent objects à la carte. In: Correctness, Concurrency, and Compositionality, LNCS, this volume, Springer (2008)
23. Verhoef, M.: Modeling and Validating Distributed Embedded Real-Time Control Systems. PhD thesis, Radboud University Nijmegen, The Netherlands (2008)