Radboud Repository

Radboud University Nijmegen

PDF hosted at the Radboud Repository of the Radboud University Nijmegen

The following full text is a preprint version which may differ from the publisher's version.

For additional information about this publication click this link.
http://hdl.handle.net/2066/35658

Please be advised that this information was generated on 2022-08-23 and may be subject to change.

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

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

[1] Chess Information Technology and Radboud University Nijmegen, NL*
[2] Engineering College of Aarhus, Denmark
[3] Embedded Systems Institute and Radboud University Nijmegen, NL
Marcel.Verhoef@chess.nl, pgl@iha.dk, hooman@cs.ru.nl

Abstract. The complexity of real-time embedded systems is increasing, for example due to the use of distributed architectures. An extension to the Vienna Development Method (VDM) is proposed to address the problem of deployment of software on distributed hardware. The limitations of the current notation are discussed and new language elements are introduced to overcome these deficiencies. The impact of these changes is illustrated by a case study. A constructive operational semantics is defined in VDM++ and validated using VDMTools. The associated abstract formal semantics, which is not specific to VDM, is presented in this paper. The proposed language extensions significantly reduce the modeling effort when describing distributed real-time systems in VDM++ and the revised semantics provides a basis for improved tool support.

1 Introduction

The complexity of embedded systems is rapidly increasing; they are becoming distributed almost by default, for example due to the System-on-Chip design philosophy which is often used nowadays. Safety-critical applications have traditionally been federated, meaning that each "function" has its own CPU with minimal interconnections to other functions in the system. This approach is expensive and for some application areas, such as the automobile industry, it is no longer economically viable to do so. The current trend is rather to combine functions together on the same processing unit and then distribute their operation between a number of networked fault-tolerant processors in order to reduce cost. It is not hard to imagine that finding the "right" deployment of functionality over such a distributed architecture, that meets all the imposed system-level requirements, is quite a challenging problem.

It is natural to advocate the use of formal techniques in this application area in order to cope with this complexity and indeed a large body of knowledge exists on their use. Most formal techniques however, are not able to deal with

* This work has been carried out as part of the Boderc project under the responsibility of the Embedded Systems Institute. This project was partially supported by the Netherlands Ministry of Economic Affairs under the Senter TS program.

the combination of complex behavior, timing, concurrency and in particular distribution in a flexible and intuitive way. Tool support often does not scale very well to the size of problems faced by industry. System development lead times remain substantial, even if formal methods can be usefully applied.

The Vienna Development Method (VDM) has been used in several large-scale industrial projects [1,2,3,4]. Their success was very much due to the solid formal basis of the notation and the availability of robust and commercial grade tools. However, not much is known about the application of VDM in the area of distributed real-time embedded systems. In earlier work [5], we reported that it is very hard to describe such systems in VDM. The language is not sufficiently expressive and important tool features are missing to analyze such models.

The aim of this paper is to make VDM++ better suited for describing distributed embedded real-time systems and to enable the design space exploration as mentioned before. In Sect. 2, an overview of the notation and the existing timed extension is presented. The limitations experienced in our earlier work are summarized and we introduce the main proposed adaptations: the addition of deployment and asynchronous communication. A small case study is presented in Sect. 3 that demonstrates the impact of the proposed changes. In Sect. 4, we define an abstract formal semantics of the extended language and discuss how the semantics has been validated. Finally, in Sections 5 and 6 we present related work and we discuss the results achieved.

2 An overview of the VDM notation

VDM++ is an object-oriented and model-based specification language with a formally defined syntax, static and dynamic semantics. It is a superset of the ISO standardized notation VDM-SL [6]. VDM++ was originally designed in the ESPRIT project AFRODITE and it was subsequently improved and tools were implemented by IFAD. Different VDM dialects are supported by industry strength tools, called VDMTOOLS, which are currently owned and further developed by CSK [4]. A timed extension to VDM++ was delivered as part of the VICE project: "VDM++ In a Constrained Environment" [7].

The dynamic semantics of an executable subset of VDM++ is provided as a constructive operational semantics specified in VDM-SL which is roughly 500 pages including informal explanation [8]. The core of this specification is an abstract state machine which is able to execute a set of formally defined primitive instructions. Special functions are supplied to "compile" each abstract syntax element into such a sequence of instructions. The dynamic semantics specification is executable and can be validated using VDMTOOLS. The test suite contains several thousand test cases which are also used to verify the implementation. 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.

[4] *http://www.csk.com/support_e/vdm/.*

For an in-depth presentation of the language and supporting tools[5] see [3]. We provide an overview in Sect. 2.1 and introduce the timed extensions in Sect. 2.2. The limitations of these extensions are discussed in Sect. 2.3 and we present our proposed language modifications in Sect. 2.4.

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 or 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 re-entrant and their invocation is defined with synchronous (rendez-vous) semantics. It is possible to constrain the execution of an operation by specifying a permission predicate [9]. A permission predicate is a Boolean expression over so-called history counters that acts as a guard for the operation, for example to express mutual exclusion. History counters are maintained per object to count the number of requests, activations and completions per operation.

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. The thread is created whenever the object is created but the thread needs to be started explicitly using the *start* operator. It is possible to specify threads that never terminate.

2.2 The existing timed extension to VDM++

In the VICE project [7], time was added by assigning a user-configurable default duration to each basic language construct. Whenever a statement is evaluated by

[5] Many examples and free tool support can be found at *http://www.vdmbook.com*.

the interpreter, the global notion of time is increased by the specified amount. In this way, it was possible to simulate the timed behavior of a program running on a single processor. In addition, the user can specify the task switch overhead and the scheduling policy used. The duration statement was added to the language, with the concrete syntax *duration(d) IS*, which implies that all statements in *IS* are executed instantaneously and then time is increased by d time units. The duration statement is used to override the default execution time for *IS*. Furthermore, the periodic statement was introduced, with the concrete syntax *periodic(d)(Op)*. This statement can only be used in the thread clause to denote that operation *Op* is called periodically every d time units.

2.3 The limitations of timed VDM++

In previous work [5], 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 either blocked on a permission predicate (guard) or 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 the complexity of the model is increased and analysis of the model becomes harder.
2. Timed VDM++ supports a uni-processor multi-threading model of computation which means that at most one thread can claim the processor and only this active thread can push time in the model forward. 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. The subsystems and the environment are independent and therefore need their own notion of time which requires a multi-processor multi-threading model of computation.
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.

2.4 Proposed changes

Our aim is to minimize the impact on the existing language as much as possible. Ideally, we want to remain backwards compatible in order to reuse existing models and tools. Therefore, we have not considered to merge VDM++ with other techniques. Informally, we propose the following changes:

1. The semantics of timed VDM++ is based on the assumption that at most one thread can push time forward in the model. We propose a richer semantics in which this limitation is removed. 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 timed VDM++.
2. The suggestion is to introduce the *async* keyword in the signature of an operation to denote that an operation is asynchronous. The caller shall no longer be blocked, it can immediately resume its own thread of control after the call is initiated. A new thread is created and started immediately to execute the body of the asynchronous operation.
3. A collection of special predefined classes, *BUS* and *CPU*, are made available to the specifier to construct the distributed architecture in his model. The *system* class is used to contain such an architecture model. User-defined classes 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.
4. The *duration* statement is kept intact to specify time delays that are independent of the system architecture. In addition, we introduce the *cycles* statement, with a similar concrete syntax, to denote a time delay that is relative to the capacity of the resource. The time delay incurred by the message transfer over the *BUS* can be made dependent of the size of the message being transfered, which is a function of the parameter values passed to the operation call.

We will demonstrate the impact of these changes in Sect. 3 using a small case study and in Sect. 4 we present the semantics of the main extensions.

3 Modeling an in-car radio navigation system

In previous work [10,11], we have studied the design of an in-car radio navigation system. Such an infotainment system typically executes several concurrent software applications that share a common, and often distributed, hardware platform. Each application has individual requirements that need to be met and the question is whether all requirements can be satisfied when a particular architecture is chosen. We present a VDM++ model of the distributed in-car radio navigation system using the suggested language improvements. We have focused on modeling the non-functional performance aspects because these will highlight the impact of the language changes most prominently. The case study aims to demonstrate that it is easy to describe distributed architectures and the associated deployment of functionality onto it. The model presented here reflects one of the proposals that was considered during the design, consisting of three processing units connected through an internal communication bus. We use the terms application and task to informally describe the case study. An overview is presented in Fig. 1.

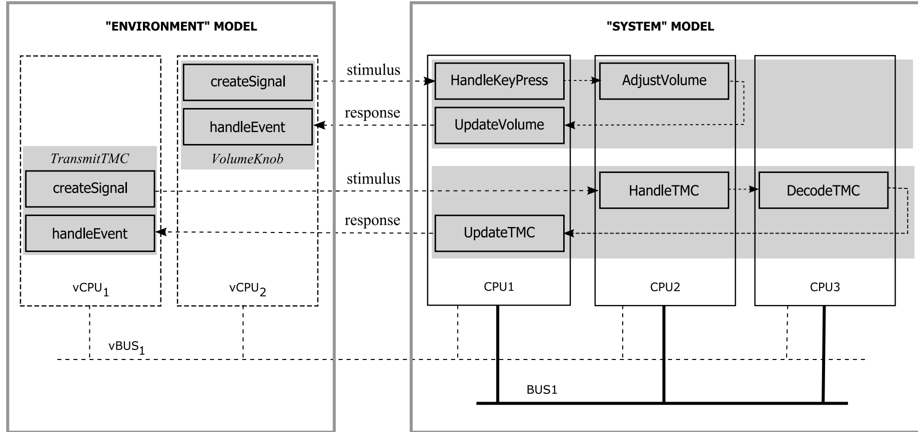

Fig. 1. Informal overview of the case study

Two applications are running on the system: *ChangeVolume* and *ProcessTMC*. Each application consists of three individual tasks. The *ChangeVolume* application, represented by the top right gray box, controls the volume of the radio. The task `HandleKeyPress` takes care of all user interface input handling, `AdjustVolume` modifies the volume accordingly and `UpdateVolume` displays the new volume setting on the screen. The *ProcessTMC* application, indicated by the bottom right gray box in Fig. 1, handles all Traffic Message Channel (TMC) messages. TMC messages arrive at the `HandleTMC` task where they are checked and forwarded to the `DecodeTMC` task to be translated into human readable text which is displayed on the screen by the `UpdateTMC` task.

Two additional applications represent the environment of the system: *VolumeKnob* and *TransmitTMC*. The former is used to simulate the behavior of a user turning the volume knob at a certain rate and the latter is used to simulate the behavior of a radio station that transmits TMC messages. Both applications inject stimuli into the system (`createSignal`) and observe the system response (`handleEvent`).

In the remainder of this section, we will present how applications and tasks from the informal case study description relate to classes, operations and threads in VDM++. Furthermore, we will show how distributed architectures are described and how objects are deployed. We present the environment model in more detail in Sect. 3.1 and the system model in Sect. 3.2.

3.1 The environment model

In our case study, there are two environment applications. Each application is represented by a class, the tasks are represented by operations in that class. An instance of the class is automatically deployed on an implicit computation resource, denoted by the dashed boxes in Fig. 1. Environment applications operate in parallel to the system and independent of each other. Execution of an

environment application does *not* affect the notion of time in other environment or system applications. Environment applications communicate with the system applications over an implicit communication resource that connects all computation resources in the model.

Typical system-level temporal and timing properties can be specified over the interface between the environment and the system model. Informal examples of these requirements are: *"The order of the VolumeKnob stimuli is preserved by the output response sequence of the system."* and *"For each HandleTMC stimulus, the maximum allowed response time shall be less than 1000 time units."*. These requirements can be modeled using standard VDM++ constructs. The `TransmitTMC` class is presented in Fig. 2.

```
class TransmitTMC
instance variables
  static private id : nat := 0;
  protected e2s : map nat to nat := {|->};
  protected s2e : map nat to nat := {|->}

operations
  getNum: () ==> nat
  getNum () == ( dcl res : nat := id; id := id + 1; return res );

  async public handleEvent: nat ==> ()
  handleEvent (pev) == s2e := s2e munion {pev |-> time}
  post forall idx in set dom s2e & s2e(idx) - e2s(idx) <= 1000;

  createSignal: () ==> ()
  createSignal () ==
    ( dcl num : nat := getNum(); e2s := e2s munion {num |-> time};
      RadNavSys`radio.HandleTMC(num) )

thread periodic (1000) (createSignal)
sync mutex(getNum)

end TransmitTMC
```

Fig. 2. The *TransmitTMC* class

Two instance variables are maintained to log the stimuli (*e2s*) and the responses (*s2e*). These variables are mappings from a unique natural number provided by the operation `getNum`, to identify each stimulus, to another natural number that represents the time at which the event was recorded. The *time* keyword provides access to the "wall clock" of the interpreter whenever the model is executed. A TMC event is inserted into the system by the periodic thread `createSignal` every 1000 time units by calling the asynchronous operation `HandleTMC` of the *Radio* class shown in Fig. 3. The operation `handleEvent` is called by the system at the end of the `UpdateTMC` operation (not shown here),

indicating that the event was completely processed by the *ProcessTMC* application. The worst-case response time requirement is encoded as a postcondition to the `handleEvent` operation. The postconditions are checked when the model is simulated. Whenever the postcondition is false, the interpreter will stop and the state of the system can be inspected to determine the cause of the problem. Other timeliness requirements can be specified in a similar way.

3.2 The system model

In the system model of our example, there are two independent applications that consist of three tasks each. Tasks can either be triggered by external stimuli or by receiving messages from other tasks. A task can also actively acquire or provide information by periodically checking for available data on an input source or delivering new data to an output source. All three notions of task activation are supported by our approach. Note that task activation by external stimuli can be used to model *interrupt handling*. The `HandleKeyPress` and `HandleTMC` tasks belong to this category. The other tasks in our system model are message triggered. We already used periodic task activation in the environment model (`createSignal`).

```
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. 3. The *Radio* class

Application tasks are modeled by asynchronous operations in VDM++. Fig. 3 presents the definition of `AdjustVolume` and `HandleTMC`, which are grouped together in the *Radio* class. We use the *skip* statement for illustration purposes here, it can be replaced with an arbitrary complex statement to describe the actual system function that is performed, for example changing the amplifier volume set point. Note that `AdjustVolume` uses the *duration* statement to denote that a certain amount of time expires independent of the resource on which it is deployed. The duration statement now states that changing the set point always takes 150 time units. For illustration purposes, `HandleTMC` 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 an 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.

A special class called *CPU* is provided to create computation resources in the system model. 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 with zero overhead is used, although our approach is not restricted to any policy in particular.

```
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. 4. The top-level system model for the case study

A special class *BUS* is provided to create communication resources in the system model. A communication resource is characterized by its throughput, specified by the number of messages that can be handled per unit of time, the scheduling policy that is used to determine the order of the messages being exchanged and a factor to denote the protocol overhead. 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 with zero overhead, but again the approach is not restricted to any policy in particular. An overview of the VDM++ system model is presented in Fig. 4.

4 Abstract Operational Semantics

In this section we formalize the semantics of the proposed changes to VDM++, as described in Sect. 2.4. To highlight the main changes and modifications, an abstract basic language which includes the new constructs is defined in Sect. 4.1. We describe the intended meaning and discuss the most important issues that had to be addressed when formalizing this. In Sect. 4.2, a formal operational semantics is defined. Validation of this semantics is discussed in Sect. 4.3.

4.1 Syntax

We abstract from many aspects and constructs in VDM++ and assume given definitions of classes, including explicit definitions of synchronous and asynchronous operations. It is assumed that these definitions are compiled into a sequence of instructions. We abstract from most local, atomic instructions (such as assignments) and consider only the *skip* instruction. Let d denote a nonnegative time value, and let $duration\ (d)$ be an abbreviation of $duration(d)\ skip$. Assume that, for an instruction sequence IS, the statement $duration(d)\ IS$ is translated into $IS \mathbin{\hat{}} duration(d)$, where internal durations inside IS have been removed and the "ˆ" 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. Let *ObjectId* be the set of object identities, with typical element *oid*. *Operation* denotes the set of operations, with typical element *op*. The predicate $syn?(op)$ is true iff the operation is synchronous. The syntax of the instructions is defined by:

$$\begin{array}{lll} Instr. & I ::= skip \mid call(oid, op) \mid duration(d) \mid periodic(d)\ IS \\ Instr.\ Seq. & IS ::= \langle\rangle \mid I \mathbin{\hat{}} IS \end{array}$$

These basic instructions have the following informal meaning:

- *skip* represents a local statement which does not consume any time.
- $call(oid, op)$ 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 by permission predicates as discussed in Sect. 2.1. 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. As shown in Sect. 3.2, $cycles(d)$ can be expressed as a duration statement.
- $periodic(d)\ IS$ leads to the execution of instruction sequence IS each period of d time units.

To formalize deployment, assume given a set of nodes *Node* and a function *node* which gives for each object identity *oid* its processor, denoted $node(oid)$. Furthermore, assume given a set of links, defined as a relation between nodes $Link = Node \times Node$, 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 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 make the transmission time dependent of, e.g., 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 precise meaning of $periodic(d)$ *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.

4.2 Formal Operational Semantics

The aim of the operational semantics is to define the execution of the language defined in Sect. 4.1. 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. 8).

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. [12]. Let *Thread* be a set of thread identities; each thread i is related to one object, denoted by o_i. This also leads to the deployment of threads: $node(i) = node(o_i)$. Finally, we extend the set of instructions *Instruction* 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.

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

- $instr : Thread \rightarrow seq[Instruction]$ 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.
- $status : Thread \rightarrow \{dormant, alive, waiting\}$ to denote the status of threads.
- $q : ObjectId \rightarrow queue[Thread \times Operation]$ 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.
- $now : Time$ to denote the current time.

For a configuration C, we use the notation $C(f)$ to obtain its field f, such as $C(instr)$. 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. We use $exec(C, i)$ as an abbreviation for $C(curthr)(node(i)) = i$ to express that thread i is executing on its node. Let $fresh(C, oid)$ 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 $instr(i)$.

Steps have been grouped into several definitions, leading to the following overall definition of a step.

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

Definition 4 (Execute Instruction). A step $C \longrightarrow C'$ corresponds to the execution of an instruction iff there exists a thread i such that $exec(C, i)$ and $head(C(instr)(i))$ is one of the following 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,
  $C' = C[\ instr(i) \mapsto tail(C(instr)(i))\ ]$
- $call(oid, op)$: Let IS be the explicit definition of operation op of object oid. If caller and callee are on the same node, i.e. $node(i) = node(oid)$ and $syn?(op)$ then IS is executed directly in the thread of the caller, i.e.,
  $C' = C[\ instr(i) \mapsto IS\,\hat{}\ tail(C(instr)(i))\ ]$
  Otherwise, if not $syn?(op)$, we add the pair (i, op) to the queue of oid, i.e.,
  $C' = C[\ instr(i) \mapsto tail(C(instr)(i)),\ q(oid) \mapsto insert((i, op), C(q)(oid))\ ]$
  If $node(i) \neq node(oid)$ and link l connects the nodes, then the call is transmitted via l, so added to the linkset. If $syn?(op)$, thread i becomes *waiting*:
  $C' = C[\ instr(i) \mapsto tail(C(instr)(i)),\ status(i) \mapsto waiting,$
  $\qquad linkset(l) \mapsto insert(m, C(linkset)(l))\ ]$
  where $m = (call(i, oid, op), C(now) + \delta_{min}(l), C(now) + \delta_{max}(l))$. Similarly for asynchronous operations, except that then the status of i is not changed.
- $duration(d)$: To allow progress of time, we require that all threads that are *alive* and have a non-empty instruction sequence can only perform a duration. Then time may progress with t time units if $C(now) + t$ is smaller or equal than all upper bounds of messages in link sets and t is smaller or equal than all durations that are at the head of an instruction sequence of some thread. To ensure progress of time (and to avoid Zeno behavior) we choose the largest t satisfying these conditions. Durations in instruction sequences are modified by the following definition which yields a new function from threads to instruction sequences:
  $NewDuration(C, t) = \lambda i :$ if $head(C(instr)(i)) = duration(d_i)$
  $\qquad$ then if $d_i - t = 0$ then $tail(C(instr)(i))$
  $\qquad\qquad$ else $duration(d_i - t)\,\hat{}\ tail(C(instr)(i))$
  $\qquad$ else $C(instr)(i)$.
  $C' = C[\ instr \mapsto NewDuration(C, t),\ now \mapsto C(now) + t\ ]$
- $periodic(d)\ IS$: In this case, IS is added to the instruction sequence of thread i and a new thread $j = fresh(C, o_i)$ is started which repeats the periodic instruction after a duration of d time units, i.e.
  $C' = C[\ instr(i) \mapsto IS,\ instr(j) \mapsto duration(d)\,\hat{}\ periodic(d)\ IS,$
  $\qquad status(j) \mapsto alive\ ]$
- $return(j)$: Then we have $node(i) \neq node(j)$ and the return is transmitted via the link l which connects the nodes, i.e.,

$C' = C[\ instr(i) \mapsto tail(C(instr)(i)),\ linkset(l) \mapsto insert(m, C(linkset)(l))\ ]$
where $m = (return(j), C(now) + \delta_{min}(l), C(now) + \delta_{max}(l))$

Definition 5 (Context Switch). A step $C \longrightarrow C'$ corresponds to a context switch iff there exists a thread i which is not running, i.e. $\neg exec(C, i)$, and also not dormant or waiting, i.e. $C(status)(i) = alive$. Then i becomes the current thread and a duration of δ_{cs} time units is added to represent the context switching time: $C' = C[\ instr(i) \mapsto duration(\delta_{cs})\,\hat{}\,C(instr)(i),\ curthr(node(i)) \mapsto i\ ]$

Definition 6 (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 $C(linkset)(l)$ with $lb \leq C(now) \leq ub$. There are two possibilities for message m:

- $call(i, oid, op)$: Insert the call in the queue of object oid:
  $C' = C[\ q(oid) \mapsto insert((i, op), C(q)(oid)),$
  $linkset(l) \mapsto remove((m, lb, ub), C(linkset)(l))\ ]$
- $return(i)$: Wake-up the caller, i.e.
  $C' = C[\ status(i) \mapsto alive,\ linkset(l) \mapsto remove((m, lb, ub), C(linkset)(l))\ ]$

Definition 7 (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 $head(C(q)(oid)) = (j, op)$. Let $j = fresh(C, oid)$ be a fresh thread and IS be the explicit definition of op. Then if the operation is synchronous, i.e. $syn?(op)$, then we start a new thread with IS followed by a return to the caller:
$C' = C[\ instr(j) \mapsto IS\,\hat{}\,return(j),\ status(j) \mapsto alive,\ q(oid) \mapsto tail(C(q)(oid))\ ]$
Similarly for an asynchronous call, where no return instruction is added:
$C' = C[\ instr(j) \mapsto IS,\ status(j) \mapsto alive,\ q(oid) \mapsto tail(C(q)(oid))\ ]$

Definition 8 (Operational Semantics). The operational semantics of a specification in the language of Sect. 4.1 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, such as: initially no thread has status *waiting*, all current threads are *alive*, the auxiliary instruction *return* does not occur in any instruction sequence, and all queues and link sets are empty.

Observe that in the current semantics the threads that may execute are chosen non-deterministically. By introducing fairness constraints, or a particular scheduling strategy such as round robin or priority-based pre-emptive scheduling, the set of execution sequences can be reduced. Another reduction can be obtained by the use of permission predicates.

4.3 Validation

The formal operational semantics has been validated by formulating it in the typed higher-order logic of the verification system PVS [6] and verifying properties about it using the interactive theorem prover of PVS.

[6] PVS is freely available, see *http://pvs.csl.sri.com/*. The PVS files and all VDM++ models are available on-line at *http://www.cs.ru.nl/~marcelv/vdm/*.

In fact, the formal operational semantics presented in this paper is based on a much larger constructive (and therefore executable) operational semantics of the extended language, which has been specified in VDM++ itself. This "bootstrapping" approach allows us to interpret models written in the modified language by symbolic execution of its abstract syntax in the constructive operational semantics model using the existing and unmodified VDMTools.

A large collection of test cases has been created to observe the behavior of each new language construct and we are fairly confident that the proposed language changes are consistent. The constructive operational semantics is currently approximately 100 pages including the test suite. It can be used as a specification to implement the proposed language changes in VDMTools.

5 Related work

In the context of UML, there is related work [13,12] about the precise meaning of active objects, with communication via signals and synchronous operations, and threads of control. In [13] a labeled transition system has been defined using the algebraic specification language CASL, whereas [12] 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. Hence, the number of threads is more restricted than in the VDM++ semantics described here. In addition none of these works deal with deployments. The related work that comes closest here is the UML Profile for Schedulability, Performance and Time, and research on performance analysis based on this profile [14].

6 Concluding remarks

We propose an extension of VDM++ to enable the modeling of distributed real-time embedded systems. These language extensions allows us to experiment with different deployment strategies at a very early stage in the design. On the syntactic level, the changes seem minor but they make a big difference. The model of the in-car navigation system presented in this paper is significantly smaller than the model that was created earlier with timed VDM++. Moreover, the new model covers a much larger part of the problem domain. We believe that important system properties can be validated in a very cost-effective way if these features are implemented in VDMTools.

A constructive operational semantics was defined for a language subset to prototype and validate the required improvements in the semantics. The changes are substantial but they still fit the general framework of the full VDM++ dynamic semantics. Furthermore, a generalized abstract operational semantics, that is not specific to VDM, is presented in this paper as a result.

One might argue that VDM and therefore this work, is not very relevant for distributed real-time embedded systems. We believe that this is not true. The

Japanese company CSK, which has recently bought the intellectual property rights to VDMTools, is targeting this market in particular and they have already expressed their interest in our results. For example, we were granted access to the company confidential dynamic semantics specification. Furthermore, we hope that the availability of free VDM tools and the recently published book [3] will revitalize the VDM community.

Acknowledgments The authors wish to thank the anonymous reviewers, Søren Christensen, John Fitzgerald, Finn Overgaard Hansen, Shin Sahara and Evert van de Waal for their valuable comments and support when writing this paper.

References

1. Van den Berg, M., Verhoef, M., Wigmans, M.: Formal Specification of an Auctioning System Using VDM++ and UML – an Industrial Usage Report. In Fitzgerald, J., Larsen, P.G., eds.: VDM in Practice. (1999) 85–93
2. Hörl, J., Aichernig, B.K.: Validating Voice Communication Requirements using Lightweight Formal Methods. IEEE Software **13–3** (2000) 21–27
3. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object–oriented Systems. Springer (2005)
4. Kurita, T., Oota, T., Nakatsugawa, Y.: Formal Specification of an IC for Cellular Phones. In: Proceedings of Software Symposium 2005, Software Engineering Association of Japan (2005) 73–80 (in Japanese)
5. Verhoef, M.: On the Use of VDM++ for Specifying Real-time Systems. Proc. First Overture Workshop (2005)
6. Andrews, D. J., Larsen, P. G., Hansen, B. S., Brunn, H., Plat, N., Toetenel, H., Dawes, J., Parkin, G. and others: Vienna Development Method – Specification Language – Part 1: Base Language. ISO/IEC 13817-1 (1996)
7. 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)
8. Larsen, P.G., Lassen, P.B.: An Executable Subset of Meta-IV with Loose Specification. In: VDM '91: Formal Software Development Methods, Springer LNCS 551 (1991) 604–618
9. Lano, K.: Logic Specification of Reactive and Real-time Systems. Journal of Logic and Computation **8-5** (1998) 679–711
10. Wandeler, E., Thiele, L., Verhoef, M., Lieverse, P.: System Architecture Evaluation Using Modular Performance Analysis – A Case Study. Software Tools for Technology Transfer, Springer (2006) (to appear)
11. Hendriks, M., Verhoef, M.: Timed Automata Based Analysis of Embedded System Architectures. In: Proc. WPDRTS'06. IEEE (2006)
12. Hooman, J., van der Zwaag, M.: A Semantics of Communicating Reactive Objects with Timing. Software Tools for Technology Transfer **8-2** Springer (2006) 97–112
13. Reggio, G., Astesiano, E., Choppy, C., Hussmann, H.: Analysing UML Active Classes and Associated Statecharts - a Lightweight Formal Approach. In: Fundamental Approaches to Software Engineering, Springer LNCS 1783 (2000) 127–146
14. 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, Springer LNCS 3273 (2004) 143–157