Real Time Formal Specification using VDM^{++}

Jan van Katwijk
Delft University of Technology
Netherlands
Email: J.vanKatwijk@twi.tudelft.nl

Eugène Dürr
CAP Gemini Sogeti
Netherlands

Stephen Goldsack
Imperial College London,
United Kingdom

Abstract

VDM^{++} [1] *is a formal Object Oriented Specification language, derived from VDM. It extends VDM by providing object-orientation and concurrency features. The use of the language is supported by design guidelines and a toolset. The latter offers graphical representations, syntactic and semantic checking, pretty printing and code generation.*

In this paper we address real-time extensions as being developed for the language.

Keywords: *Software Engineering, Real-time Object-Oriented processing,*

1. Introduction

In this paper we report on our work on extending VDM^{++} in the area of real-time specifications. VDM^{++} is meant to be a notational vehicle for modeling and development of systems. Being born from VDM, the language provides a rich set of data structures for building abstract models of the system to be built. To support the development process, data and operation reification techniques, as known in plain VDM, are extended with specific *Object-Oriented* refinement steps (termed "annealing"). The development of a system from a high-level model consists of repeatedly applying these annealing steps. A large collection of more or less standardized reification steps is being developed.

The language has shown to be successful as specification language in a number of commercial project. One of the major successes of the language was its use in the Combicom project [10], [11].

In this paper we discuss extensions to the VDM^{++} language that we are currently working on to support real-time issues in modeling and development.

[1] The VDM++ work was partially funded by the European Information Technology Research Programme, Esprit III under number 6500.

The structure of this paper is as follows. In section 2 we briefly discuss the main characteristics of VDM^{++} , in section 3 we discuss the way action/event modeling is included in the language, and in section 4 we illustrate the typical real-time features of the language by an example. Finally, in section 5 we discuss some conclusions and future work.

2. VDM++, a brief overview

VDM^{++} [3, 1] is a formal specification language based on VDM-SL [5, VDM Standard] and extended in an object-oriented fashion with elements from Smalltalk [4]. VDM^{++} provides a wide range of constructs such that a user can formally specify (concurrent) systems in an object-oriented fashion.

The language was originally created by Eugène Dürr. It has been developed into a full, mature language as as part of the ESPRIT-III project Afrodite (project number 6500). During the Afrodite project, the language was established by developing a complete definition. Furthermore, a toolset was designed and implemented to provide commonly used object-oriented graphical representations (OMT) as representation of VDM^{++} specifications. Apart from a toolset aiming at support in building a specification, a language interpreter was developed (obviously only interpreting the executable elements in the language), while furthermore a C++ prototype code generator was developed with which (parts of) VDM^{++} specifications can be transformed into C++ code. Recent developments of the language are related to extensions for real-time object-oriented specification.

In VDM^{++} a complete formal specification, also called a *Model*, comprises a set of *class* specifications and an optional specification of a *workspace*.

A class specification in VDM^{++} has the following components :

Class header In the class header one specifies the name of the class, and its possible relationship with parent

classes. *is subclass of* clause achieves (multiple) inheritance from other classes.

Instance Variables The state of an object is made up from the values of the instance variables of the object. Instance variables are variables of simple types, VDM-SL typeconstructors as sets, sequences and maps and object references (*clientship relation* denoted by @Classname) can be declared here. The state space of objects of a given class can be constrained by the specification of an *invariant* clause, a relation over the instance variables of the objects of the class. Similarly, the set of allowable initial values of the instance variables of the objects of a given class can be specified using an *initial* specification.

Methods The methods belonging to the objects of a class can be defined several ways:

- *implicitly*, by specifying the pre- and postconditions of the method. As in VDM-SL , the precondition refers to values of state elements before invocation of the method, the post condition may refer to values of state elements as they were before invocation as well as as they are after invocation of the method.

```
class example is
   s: sequence of T;
   method my_try ()
      pre not is_empty (s)
      post is_inverse (~s, s)
...
```

  As in VDM-SL , a reference to a name preceded by a tilde in the post condition indicates a reference to the value prior to invocation of the method.

  Roughly speaking, the VDM-SLsemantics apply, i.e. the operation is only defined when the precondition is satisfied, while loosenessin the post condition is allowed.

- *explicitly*, using functional and imperative constructs, although this style is not encouraged in the specification of high-level models.

- as *not yet defined*, A specification as *not yet defined* is useful when the environment of an object demands the existence of a method, but at the same time – at that stage of development – it is not considered necessary to present its exact definition.

Invocation of a method of a client object is considered to be a *statement*. The invocation syntax is

```
expression  ! methodname (parameters)
```

For a well-formed specification, the *expression* yields an object of a class that contains the method *methodname*.

Controlled Inheritance An inheritance reduction clause can restrict visibility of methods obtained from the superclass to the external user of an object.

Auxilary Reasoning A class might have an *Auxilary Reasoning* part. In such a part, axioms, properties, and invariants required to perform correctness proofs and constraints for other depending classes can be specified here. References to internal states of client objects are limited to read-only accesses. The format is not constrained, the contents of the auxiliary reasoning part are not processed by the available tools.

Outline of an example

```
class Train
   instance variables
      speed : N;
      power : N;
      direction: (Forward, Backward);
      inv speed = F (power)
   methods
      set_power () == not yet defined
      get_speed () == not yet defined
end Train
```

The example class defines objects of class *Train*, used in a later section of this paper. The state of the train consists of the power supplied to drive the train, and the direction of the transmitted speed. The instance variable speed is related to the power by the invariant P.

A system specification is completed by the description of a *workspace*. The workspace mechanism has the role of the 'main procedure' in other languages. Usually, a workspace has a special method called the 'initial-method'. Its role is to create the objects of which the system is initially composed, and to establish their topological relations. A workspace object is implicitly created at the start of the pseudo execution of the specified system and its initial method is invoked by *Deus ex Machina*. The (pseudo) execution of parallel executing objects is started from here.

3. Real-time specification and VDM^{++}

3.1. Introduction

The VDM^{++} paradigm states that building a high-level model of the system should result in a model of the software system, together with its environment. The model is closed, i.e. it contains all interactions between the submodel of the software system and the submodel of the environment.

Elements in the environment are usually represented by objects, as is the model of the software system itself. Objects

therefore are considered to model/simulate active world entities, as a result, objects need some form of behaviour.

In VDM^{++} we have chosen to make an object active by allowing a class to contain a specification of a *thread*. The thread definition has the form

```
thread
   thread specification
```

A thread specification is either a procedural thread, or a declarative one. The declarative form specifies a periodic action, the execution of a method, to be executed with a given period.

$periodic(\Delta T)(methodname)$

It is assumed that for each thread a processor is available for the execution of methods.

The procedural form a the thread is not used throughout this paper and will not be discussed here.

Apart from objects being active by letting them have threads, we needed the notion of time. In VDM^{++} , a global clock is assumed, that can be read by anyone. It's value can be read by refering to the implied clock variable now.

Since, time is considered to be a continuous variable with infinite accuracy, expressions using references to time in a system cannot not use strict equality.

Introducing time this way, allows the functionality of methods to depend on time (at least to a certain extent).

Consider as an example

```
class example is
  .......
  method exec_time () returns d: real;
  post (d = time - ~time)
```

Since $time$ is considered to be a state element, its value before and after invocation of a method can be queried.

In the example an implicit method is specified. The post condition of the method states that the returned value (d), will be equal to the difference between the value of $time$ as it holds after the invocation of the method is finished and the value of $time$ as it holds before invocation. I.e., the result of invocation of the method will be the duration of its execution.

3.2. VDM^{++} and events

The paradigm in modeling real-time systems states that the state of the system acts as the interface between the controlling and the controlled system. Each object in the system inspects the state and reacts upon changes in this state. Each change in the state can be considered to be an event. More formally speaking, an event is the marker in time at which a predicate yields **true**.

Being in an object-oriented world however, where hiding is one of the fundamental issues, only a part of the whole system state is directly visible. This is fundamentelly different than e.g. in VDM-SL where the whole state is visible from each operation.

Some attention to what can be part of the predicate defining an event is therefore inevitable. Elements to be taken into account when defining events are:

1. an *observable* transition of the state of system. Within an OO specification visibility of the state is typically limited to the values of the instance variables of the current object, the values of instance variables of parent objects and the values of instance variables in the worksapce. Visibility of change is therefore typically limited to a change in the value of these variables.

2. the transition of the systems' state resulting from handing a method request or method invocations. Here the changes are made observable through the introduction of *synchronisation variables*. VDM^{++} has the following built-in events:

   - the request to execute a method. $req(methodname)$ will become **true**, raising an event, as soon as a request to execute the method $methodname$ is issued;
   - the actual activation (start) of a method, $act(methodname)$ will be **true**, raising an event, as soon as the invocation of the method is a fact;
   - the finalisation of a method, $fin(methodname)$ will be **true**, raising an event, as soon as the invication of the method $methodname$ terminates.

   The default available variables #req, #act and #fin record these transitions as counters. Notice that the events will be raised independent of the method of invocation of a method. I.e., although a method invocation in a periodic thread is not visibly by an explicit call, the same events apply.

Requirements for event handling in a specification notation are (i) the ability to specify events, (ii) the ability to specify the system's reaction upon an event, and (iii) the ability to specify temporal constraints on the occurrences of events and reactions. In our VDM^{++} extension, we offer the capability specifying within the thread part of a class specification a connection between an action to the occurrence of an event, using a statement of the form:

```
whenever condition also
       [from delay ] ==>  predicate
```

Such an action specification tells that whenever *condition* becomes **true**, the system should realize *predicate* within *delay* units of time.

Obviously, having the notion of time available, one might raise an alarm at a specified time

```
whenever  now > ThisAfternoon
also from delta ==> fin (methodname)
```

This specification states that an event is raised as soon as a certain calendar time is reached. On raising this event, the execution of the method *methodname* should be finished within delta units of time.

Generalizations are straightforward:

```
whenever P (now, period) also
from delta ==> fin (methodname)
```

Suppose that P is a predicate that yields **true** when

$$now\ MOD\ period < Small\ Value$$

I.e., in this case, we emulate a periodic obligation.

An obvious well formedness condition is that

$$Duration(methodname) < period$$

The ability to refer to 'state' of method invocations allows us to specify e.g. the upperbound of the execution of a particular invocation. Consider as an example

In this specification, an upperbound for the execution of *methodname* is given. As soon as *methodname* is invoked, the specification requires the implementation to ensure that the finalization of the invocation of *methodname* is not later than delta1 units of time.

```
whenever  req (methodname)
also from delta1 ==> fin (methodname)
```

In this specification, it is indicated that the execution of a method *methodname* should be terminated within $delta1$ units of time after a request to its invocation is made.

```
whenever #act (methodname) >
         #fin (methodname)
also from delta ==> .....
```

This example shows a statement in which an unspecified predicate (indicated by '...', is to become **true** within delta units of time from #act > #fin becoming **true**.

It is assumed that the performance "consumption" for the evaluation process of the various conditions is taken into account when writing the specification. The existence of the "from delta" construct acknowledges that in the real world computations and reactions takes time and cannot be executed in zero time. Omitting such a restriction specifies a requirement that is hard to fullfil: the predicate acting as consequent in the implication is to be made **true** in zero units of time.

If sensor based systems are involved, the interface between the controlled system and the controlling system may contain variables that can be assumed to be changed by the environment.

E.g.

$$whenever\ \ temp > t$$

can be used. In a later refined model, a sensor object will probably be introduced and the temperature observation mechanism will be specified in more detail.

3.3. Discussion

The semantics of having more than a single *whenever* clause within a single thread deserves some attention. Current semantics assume that multiple *whenever* clauses within such a thread exclude each other. In the event that two or more events become **true** at the same moment, only one of the clauses will be executed further. It is the responsibility of the specifier to ensure that all specified *whenever* clauses are indeed schedulable within their deadlines. Consider as an example

```
class example
   .....
   methods
   ....
thread
   whenever Pa also within delta1 ==> Qa
   whenever Pb also within delta2 ==> Gb
   ...
end example
```

Assume furthermore that, in order to satisfy Qa a method need to be invoked that takes Ex_1 units of time. Assume furthermore that, in order to satisfy Qb a method need to be invoked that takes Ex_2 units of time.

If, at any moment Pa and Pb become **true** at the same time, a necessary condition is that

$$Ex_1 + Ex_2 < min(delta1, delta2)$$

In general, however, two or more conditions in different objects may become **true** at the same time, raising different exceptions within different threads. If the actions, associated with the handling of these events, refer to the same state variables, the result is, formally speaking, undefined. It will depend upon the speed and the (arbitrary) order in the execution of the components, race conditions might occur.

Safe specifications can therefore not use instance variables in an updating mode, unless extensive proof is delivered about non interference to these variables at all times. Read only access is safe. We are considering a language extension which allows users to add their own sync variables

(variables which can be updated atomically) similar to the currently built-in #req, #act and #fin.

It is interesting to compare the event/action approach, as followed here, to the enabling approach. In the latter, at any state *an* operation will be selected that has a fullfilled precondition. No queuing of 'enabled' operations will take place. In our approach, queuing will occurr indeed. Whever two or more events are raised the associated operations will be queued for execution.

An interesting question was raised in [13]. In this paper, the notion of real-time object (RTO.k) was introduced. Such real-time objects would form the basic building blocks of real-time systems. The structure of an RTO, obviously, resembles the structure of the VDM^{++} class. However, in RTO.k objects, the methods are partitioned into two classes. Elements of one class are exported and made visible to cleints of the objects, elements of the other class are kept invisible to the external world. It is with elements of this latter class that structures, comparable to our *whenever* constructs are built.

Whether or not additional constraints should be put on the methods that can be used within thread specification is subject to further research.

Notice that an action/event model has some intrinsic problems, since the conditions leading to raising an event may have been changed whenever the action, scheduled by the event, is executed. Actions that are invoked as a response upon the raising of an event should establish whether or not the conditions, causing the event to be raised, still exist.

4. A case study: the railroad controller

Effectively evaluating formal specification notations is not always easy. In our research, we use a single example for the comparison of different specification notations. In [12], a brief comparison is given of the use of formal specification notations for a simple train controller. In this section, we briefly discuss the use of VDM^{++} as vehicle for the specification of the same controller.

The specification is sufficiently realistic to get a good insight in the possibilities and limitations of specification notations. It involves both data and control issues. The railroad itself is to be represented as some (rather complex) data structure. Physical devices (i.e. switches and trains) have to be manipulated as functions of the time and system state as well. Furthermore, the controller is subject to temporal constraints. Each train has deadlines w.r.t. its arrival time on specified positions, while furthermore, reactions of the system on occurrences of events should be within specified times.

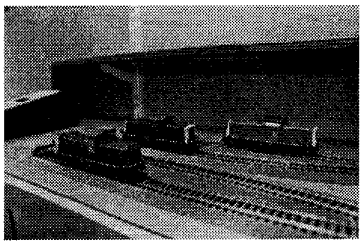

Figure 1. Fragment of railroad

4.1. The problem

A (toy) railroad system is built up from a variety of (connected) rail elements (straight elements, bowed elements, crossings and switches. Rail elements have a fixed length of app 23 cm, they are grouped into physical blocks, *a block is connected to hardwired devices. Through a computer (serial) interface, commands can be given:*

- *status enquiry commands (i.e. blocks being occuppied or not).*
- *switch setting/resetting commands.*
- *train commands, setting speed (between 0 and 30 cm/sec), setting lights, a horn and a direction (in our model, we ignore the horn and the lights).*

The controlled behaviour of the train should conform to requirements *set for the train movements by a specified mission, one for each train. Missions, and therefore train routes, are expressed in terms of* tracks, *where a track is a path over a block.*

The problem is to specify (and subsequently design and implement) a software controller with which the behaviour of the system, i.e. train movements and switch status, can be controlled. Following obervations made by many researchers in the field, we require the model to be closed, *i.e. encompassing both a model for the controller and for the controlled entities.*

A train drives according to a sequence of *commands*. A command indicates what a train is required to do on a given track, i.e. a *PASS* command indicates that the train should drive over the specified track and should have reached the end of it no later than the specified time. (A track represents a block, i.e. a group of connected rail elements, together with an indication of an entry and exit. Blocks, entries and exits are encoded as naturals. A track is encoded as a triple (*block, entry, exit*).)

Consider for example the following specification of a small mission

```
INIT   (1, 1, 2)
START  (1, 2, 1), 1000, 1010
PASS   (2, 1, 2), 1020
PASS   (3, 1, 2), 1030
STOP   (4, 1, 2), 1035
```

```
START   (4, 2, 1), 1035, 1040
PASS    (3, 2, 1), 1050
PASS    (2, 2, 1), 1060
STOP    (1, 2, 1), 1070
```

After being initialized (*INIT*) on track (1, 1, 2), the train starts at time 1000 on track (1, 2, 1) (in our encoding, track (1, 2, 1) is the reverse from (1, 1, 2)). The train is forced to move such that it leaves that track no later than 1010. The subsequent track on the route, (2, 1, 2), should be passed such that the train reaches the end of the track no later than 1020, etc. The track (2, 1, 2) being the logical successor of track (1, 2, 1) is obviously a well-formedness constraint on the railroad.

4.2. A VDM^{++} solution

The structure of the specification In [9] the architecture of the solution is given in great detail. Due to space constraints, we limit the presentation of the solution here to a discussion of some details.

The specification contains the following classes:

- The class *Train*. Objects of this class model the physical trains, as driving on the physical railroad;
- The class *RailRoad*. A single object of this class models the railroad on which the trains are driving;
- The class *TrainPosition*. Objects of this class model the relation between the trains and the railroad, i.e. they model the positions of the trains.
- The class *TrainController*. Objects of this class (one for each train) model the controller function to be executed in order to have the associated train show the required behaviour. The objects of this class contain the sequences of commands, to be executed for the associated trains.
- The class *AllocationTable*. In order to enforce that no two trains ever appear on the same track, a track is assigned to eat most one train. A train only drives on allocated tracks. Management of the tracks and their association to trains is maintained in a single object of class *AllocationTable*.

Specification of train positions Train positions relate trains to the railroad. A position can be derived, given a previous position, the structure of the railroad and the driving characteristics of the train.

```
class TrainPosition
   instance variables
      railroad: @RailRoad
      train   : @Train
      position: Position
   methods
      set_position (p: Position) == not yet defined
      get_position (p: Position) == not yet defined
      new_position () == not yet defined
   thread
      period (100 ms) (new_position)
end TrainPosition
```

The behaviour of each actual train in the system is modeled by an instance of the class *TrainPosition*. The class relates the trains (models of which are maintained in objects of class *Train*) to the actual railroad (modeled in an object of class *RailRoad*).

The thread ensures that the position information will be updated once every 100 milli seconds. Taken the speed of the train into account (max 30 cm/sec) the maximum difference between two position updates is less than or equal to 3 cm.

The railroad object, referred to from the class *TrainPosition* provides information on the structure of the railroad. Given a position on the railroad, and given a distance, a new position (taking into account the state of the switches) is computed.

Train objects contain state information on the trains they represent.

Outline of the controller To each individual train a controller object is associated that manages that train. The controller object contains an encoding of the mission of the train and essentially executes a finite state machine for that train. The FSM is given in figure 2.

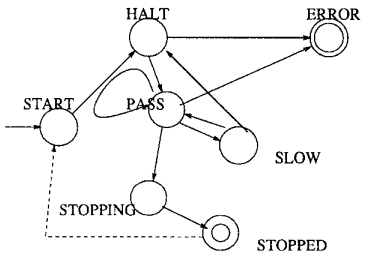

Figure 2. The state diagram of the controller

The states and the transitions in the FSM are determined by (i) the command for the train to be executed, (ii) the position of the train and (iii) the current time.

1. A train starts in state START as soon as the starting time is reached.
2. In this state, the next track on the route of this train is computed, and the train is set in state HALT.

3. A train in state PASS is a running train. Such a train remains running at least until it enters the next track on its route. If the command to be performed on this next track is a *PASS* command, an attempt will be made to allocate the next track on the route. If this attempt fails, the train will slow down (and enter state SLOW), if the attempt succeeds the state PASS will be reentered. If the next command is *STOP* command, state STOPPING will be entered.

4. In state STOPPING, the train will move forward until it reaches the middle of the track and take state STOPPED and halt. In this state, the only valid next command is a *START* command.

5. In state SLOW, the train will continue to attempt to allocate the required resource. If the train reaches the middle of the track *before* being able to allocate this resource it enters state HALT and the train halts. If the train does obtain the required resource, state PASS will be (re)entered, and the train will move with the speed required to meet its temporal constraint.

6. In state HALT, the train will continue its attempts to allocate the resource during a period of time. If however, this period is finished, the train will enter state ERROR and stops functioning. Otherwise, it will enter state PASS and it continues.

Modeling the control for the trains is then modeling the automaton in a single class.

Objects of the class contain a thread that will continuously react upon events and issue commands when reacting upon them.

```
Class TrainController is
   instance variables
      mission: Instructions*
      train: @TrainPosition
      TrackTable  : @AllocTable
      CurrentMode : (HALT, STOPPING,
                     STOPPED, PASS,
                     SLOW, START)
      CurrentTarget : Position
         -- The target of
         -- the current command
      SlowTime    : Time
      inv .....
      init ......
   methods
      ....
   Thread
      ....
      whenever      -- Transition Pass to Pass
         CurrentMode = PASS and
         TrackOf (CurrentTarget) #
           TrackOf (PositionOf (train)) and
         WantsToContinue (hd Instructions) and
         CanAllocateNextTrack (hd Instructions,
                                    TrackTable)
      also from 100 ==>
        CurrentMode = PASS and
        CurrentTarget =
          TargetFrom (hd Instructions) and
          act (train.
               set_power (ComputePower (
               TargetFrom (hd Instructions))))
      and mission = tl ~mission

   whenever      -- Transition Slow to Pass
      CurrentMode = SLOW and
      now < SlowTime + deadline and
      CanAllocateNextTrack (hd Instructions,
                                 Tracktable)
      also from 500 ==>
            CurrentMode = PASS and
            CurrentTarget =
                TargetFrom (hd Instructions) and
            act (train. set_power (
                   ComputePower (
             TargetFrom (hd Instructions)))) and
            mission = tl ~mission
   ....
end TrainController
```

For each transition in the FSM the pre- and postconditions can be written down easily in a functional style. Notice however that the postcondition contains the *act* predicate. The conforming implementation is to ensure that the method set_power is activated with the correct parameter, within 100 resp. 500 milli seconds after the predicate became **true**, raising the event.

Specification and implementation The complete specification of the railroad controller in VDM^{++} took less than 400 lines. This is roughly comparable to the sizes of specifications of the same controller in other notations. As in other notations, it excludes the specification of the railroad data structures, which were 'borrowed' from the plain VDM specification of the same railroad (see [9]).

The specification is, due to its structuredness, somewhat larger than a similar specification in plain VDM-SL.

The next step in the development has been the translation of this highly structured specification into Ada-83 code. The translation was performed by simply transforming each (active) class into a task type, with an instantiation for each object. Although the translation is in general a fairly simple one, two problems occur:

- the 'model' used in the VDM^{++} specification is an action/event model. I.e. 'some' event occurs, as a result an action is scheduled. The Ada model is, however, a synchronous one. The translation is therefore somewhat obscured since essentially an action/event model is 'implemented' in Ada.

  It is interesting that a translation into Ada-95 where asynchronicity can be obtained by using protected ob-

jects, which is underway, turns out to be straightforward.

- systematically dealing with temporal constraints in the implementation is not always trivial. The semantic gap between the *also form* part of the VDM^{++} whenever clause and the delay alternative in the Ada selective wait construct is fairly large. Ada (and Ada-95 as well in that respect) do not provide much support for a direct translation. Neither Ada nor Ada-95 provide support for mapping temporal constraints on scheduling directives.

5. Results and Conclusions

VDM^{++} has shown to be a valuable modeling tool in the area of industrial systems. The first phase of the project, establishing the language and a supporting toolset has been completed. Current research aims at

- establishing the real-time elements of the language,
- investigating and formalizing the annealing steps in the systems development.

Our experiences in the design and the useage of VDM^{++} have shown that there is a significant amount of tension between *useability* of the notation and the degree of *formalism* in the notation. Our choices have been infuenced by the desire to develop a notation that can be used as an engineering's tool. I.e. we have been aiming at a suitable notation, sufficiently defined, that allows engineers from a variety of disciplines to specify high level models of the system to be build. Less emphasis has been given to formal verifyability of properties of particular models.

Although we still have a long way to go w.r.t. the formal verifyability of properties of given models, we have paid attention to a rigorous approach to refinement (annealing).

The real-time extensions we are working with, are presented in this paper. The structure is similar to extensions earlier proposed for VDM-SL , however, the object-orientedness of the notation has influenced the design.

Current work involves handling the delay between the event being raised and the associated action to be invoked and the problem of verification.

6. References

References

[1] Dürr, E (1994), The use of Object-Oriented Specifications in Physics. PhD Thesis, Utrecht University, Utrecht, The Netherlands, 1994, ISBN 90-393-0684-2.

[2] Dürr, R and N. Plat Editors (1995). VDM^{++} Language Reference Manual, Afrodite (ESPRIT-III project number 6500) document AFRO/CG/ED.LRM/V10, Cap Volmac.

[3] Dürr, E and J. van Katwijk, (1992), VDM^{++} - A formal Specification Language for Object-Oriented Designs. In: Georg Heeg, Boris Magnusson, Editors *technology of Object-Oriented Languages and Systems*, pp 63 - 78. Prentice hall International, Proceedings of Tools Europe '92.

[4] Goldberg, A., Editor, (1984) Smalltalk-80, The Interactive Programming Environment. Addison Wesley Publishing Company.

[5] Jones, C. (1990). Systematic Software Development using VDM (2nd edition). Prentice Hall International.

[6] Mahoney, B. and Hayes, I. Using Continuous Real Functions to Model Time Histories. In Bailes, P. Editor, *Proceedings of the 6th Australian Software Engineering Conference (ASWEC91)*, pp 257-270, Australian Computer Society.

[7] Eugène Dürr, Stephen Goldsack and Nico Plat. *Rigorous Development of Concurrent Object Oriented Systems*. Tutorial(MM5) at the Tools Europe '94 Conference, March 7-10,Versailles France. In : Technology of Object Oriented Languages and Systems, Editors. B Magnusson,B.Meyer, J.Nerson, J.F. Perrot TOOLS 13, ISBN 0-13-350539-1, Prentice Hall, UK, (page 515)

[8] W. Cellary, E. Gelenbe, T. Morzy, *Concurrency Control in Distributed Database Systems* , Studies in Computer Science and AI nr. 3, North-Holland, Elseviers Science Publishers, Amsterdam, 1988.

[9] Ton Biegstraaten, Klaas Brink, Jan van katwijk, Hans Toetenel. A simple railroad controller: A case study in real-time specification. Technical Report 94-86. Reports of the Faculty of Technical Mathematics and Informatics. Delft University of Technology, Delft 1994.

[VDM Standard] ——. *VDM Specification Language: Proto-Standard(Draft)*. Document N-246(I-9), BSI IST/5/-/19 and ISO/IEC JTC1/SC22/WG19, December 1992.

[10] *Cap Gemini CombiCom Team*, CombiCom Internal Deliverable S.1.X,
High Level Formal specification of the CombiCom architecture, CombiCom Consortium.

[11] *Cap Gemini CombiCom Team*, CombiCom Internal Deliverable S.3.X,
Final Formal specification of the CombiCom architecture, CombiCom Consortium.

[12] J. van Katwijk, W.J. Toetenel. Comparing formal specifications by measuring. Proceedings of the second International Workshop on Real-Time Computing Systems and Applications. IEEE 1995, ISBN 0-8186-7106-8, pp 184 - 190.

[13] K.H.Kim, C. Subbaraman, Y. Kim. Imitation of RTO.k Objects using PCD Components in C++. In *these proceedings*.

7. About the authors

- Dr Ir E.H.Dürr is Technical Manager at Cap Gemini Sogeti, and Assistant Professor at Utrecht University, both in the Netherlands.
- Prof. J.van Katwijk, is professor at the Faculty of Mathematics and Informatics of the Technical University of Delft, The Netherlands
- Prof S. Goldsack is emeritus Professor at the Department of Computing, at Imperial College in London U.K.