Real Time Formal Specification using VDM^{++}

Eugène Dürr,[1]
Stephen Goldsack[2]
Jan van Katwijk[3]

[1] Cap Gemini, The Netherlands
[2] Imperial College, Great Britain
[3] Delft University of Technology, The Netherlands

Abstract

VDM^{++} is a formal Object Oriented Specification language, derived from VDM. It extends VDM by providing object-orientation, real-time as well as parallel features. The use of the language is supported by design guidelines and a tool set. The latter offers graphical representations, syntactic and semantic checking, pretty printing and code generation. In this paper we address the handling of particular real-time issues as supported by the language. An outline specification case study is included.

KEY WORDS: Software Engineering, Real-time Object-Oriented processing, Transportation

1 Introduction

Formal specifications are of growing importance. In this paper we report on our work on the extensions of VDM^{++} in the area of real-time specifications. VDM^{++} is meant to be a notational vehicle for system development[1].

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 standardised reification steps is being developed. The language has been used for specification in a number of commercial projects. One of the major successes of the language was its use in the CombiCom project [13], which realised a distributed Intermodal Tracking and Tracing system for freight transport.

In this paper we discuss the features of the language for real-time specifications.

In section 3 we briefly discuss the main characteristics of the language VDM^{++} , and in section 4 the design method (see also [11, 14]). In 5 we discuss the way action/event modelling is included in the language, and in section 6 we illustrate the typical real-time features of the language by an example. Finally, in section 7 we discuss some conclusions and future work.

2 Related Work

During the last years a number of formal specification notations and tools have been developed. Some of these notations are Petri Net based (e.g. [16]). Others are based on temporal logic (e.g. TLA [17]). Of course, state chart based notations are still in development (e.g. [18])

[1]The VDM++ work was partially funded by the European Information Technology Research Programme, Esprit III no. 6500.

Astral, [19], is yet another formal notation aiming at supporting modelling of real-time systems. As the other notations, ASTRAL has an underlying semantics mechanism that allows - at least partially - verification of consistency and (safety) properties. Finally, we mention CCR [20] as representation of the process-algebraic approach.
Some of these notations have been used quite successfully in various applications. There is, however, intrinsic tension between the formality of the notation needed for formal proofs of system properties, specified in the notation and engineering capabilities, required for use in larger applications.
In VDM^{++} we deliberately have chosen for the support of *engineers*, working on software development. In this approach, the aim is to provide constructs that appeal to the engineers using it, where provability of properties in the resulting specifications is considered important.
Therefore, the concepts of VDM^{++} have been selected to model physical concepts. It is generally accepted that object-orientation is a concept with which real-life situations and physical concepts can be modelled easily.
Our real-time extensions follow the same route of reasoning. Time, as seen in the controlled world is continuous, faithful modelling of real-time systems should reflect that. Events model predicates becoming True, i.e. an event is only an event is someone is willing to interpret it as an event. Based on this observation, it becomes clear that the notation provides constructs to bind the occurrences of events to constructs that specify the reaction of the system on the occurrence of such events.

3 VDM++ brief overview

3.1 Language Characteristics

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], POOL [6] and Forest [7]. VDM^{++} provides a wide range of constructs such that a user can formally specify concurrent, real-time 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). The Real Time features were developed in collaboration with I. Hayes [8, 9, 10]. During the Afrodite project, the language was established and a tool set known as the *Venus tool set* was designed and implemented to provide commonly used object-oriented graphical representations (OMT) as representation of VDM^{++} specifications.
Furthermore, code generators exists with which VDM^{++} specifications can be transformed into C++, Ada95 or Smalltalk code(prototype). (see also: www.ifad.dk/products/venus)
An overall VDM^{++} specification consists of a collection of classes. A class specification in VDM^{++} has the following components :

Class header Apart from the class name declaration an 'is subclass of' clause achieves multiple inheritance from other classes. Indexed inheritance ($Classname[n, \ldots, m]$), with several copies of the same ancestor class, is also available.

Instance Variables The *state* of an object is comprised of the set of instance variables, i.e. variables of simple types or VDM-SL type constructors such as sets, sequences and maps. Object references -*providing client-ship relation* and denoted by @Classname- can also be declared here. Values of instance variables may be constrained by invariant and initial expressions in a way similar to VDM-SL

Methods The functionality provided for
clients is offered by methods which an object is willing to execute on behalf of a client. As

with VDM-SL operations, methods can be specified implicitly as well as explicitly. Yet another form of specification is using the *not yet specified* declaration, indicating that the actual body will be given in a later stage of the development. With the declaration *is subclass responsibility* the actual specification can be delegated. The invocation of a method of a client object has the form *variableName!methodName(parlist)*, where *variableName* is the name of (a reference to) an object of some class and *methodName(parlist)* is a call on a method of that class.

Threads In VDM^{++} objects are considered to model/simulate active world entities. An object can be made active by the specification of a thread through the keyword thread followed by a *threadspecification*. A thread specification has either a procedural or a declarative specification. The declarative form specifies an action to be executed repeatedly with a given period. For example periodic(ΔT)(*methodname*). The procedural thread with statements is not used in this paper. It is assumed that a *virtual* processor is available for each thread that occurs in a specification.

Handling Time Time is considered to be an entity, global to the system. It's value can be read by referring to the implied variable now. Time continuous functions, called *time variables* can be declared. Since, time is considered to be a continuous real variable with infinite accuracy, references to time in a system cannot not use strict equality.

Auxiliary Reasoning Each class specification may have an optional *Auxiliary Reasoning part*. Axioms, properties, and invariants concerning the correctness proofs of this class and constraints for other dependent classes can be specified in such a part. The format is not constrained, the contents of the auxiliary reasoning part are not yet processed by the tools. References to internal states of client objects through read-only access are allowed here.

Example

```
class Train
  instance variables
          speed  : Real;
          power  : Real;
          direction : < Forward > | < Backward >;
          inv objectstate ==P(speed, power)
  methods
          set_power(p : Real)  ≜  [post  power = p] ;
          get_speed() ≜  return(speed)
end Train
```

The example class defines objects of class *Train*. The state of the train consists of the power supplied to drive the train, the transmitted speed and the direction. In the invariant, the (Boolean) function P couples instance variable speed to power.

A system specification is completed by the description of a *workspace*. Usually, a workspace has a special method called the 'initial-method'. The workspace mechanism has the role of the 'main procedure' in other languages. 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*. Furthermore, parallel executing objects are normally started from here.

3.2 Semantics

A complete formal semantics of VDM^{++} has been produced over time. The OO structuring is defined in [23]. The parallel and real time constructs are treated in [24] using RTL. The reasoning techniques are proven in [25]. An overview of all real time semantics can also be found in [26].

4 The Design Approach

The Afrodite design approach is based on the observation that in each development process, the in-depth understanding of domain, system and environment behaviours improves gradually with time. It is of vital importance to provide means for recording these intermediate levels of understanding. In our opinion, this process of improved understanding can be described as the route from initially quite abstract descriptions of the system, into increasingly concrete descriptions, taking implementation aspects in to account.
We sometimes call this "de-abstraction". The successive models can be viewed as shown in figure 1:

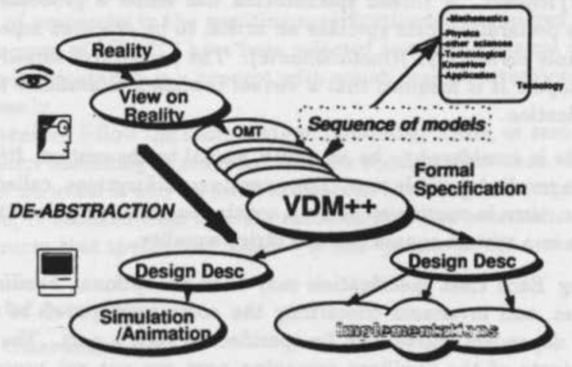

Figure 1: Sequence of Models Paradigm

Each model provides, in principle, a full representation of the ultimate system. For such a system specification, an animation or simulation can be generated. With the addition of a simple user interface one can arrive at a full system representation, of substantial value in discussions with end-users.
Clearly, this is support for the iterative spiral approach. The way in which each model is derived from a previous one is determined by a set of guidelines or rules. This relation between the two subsequent models is illustrated in figure 2:

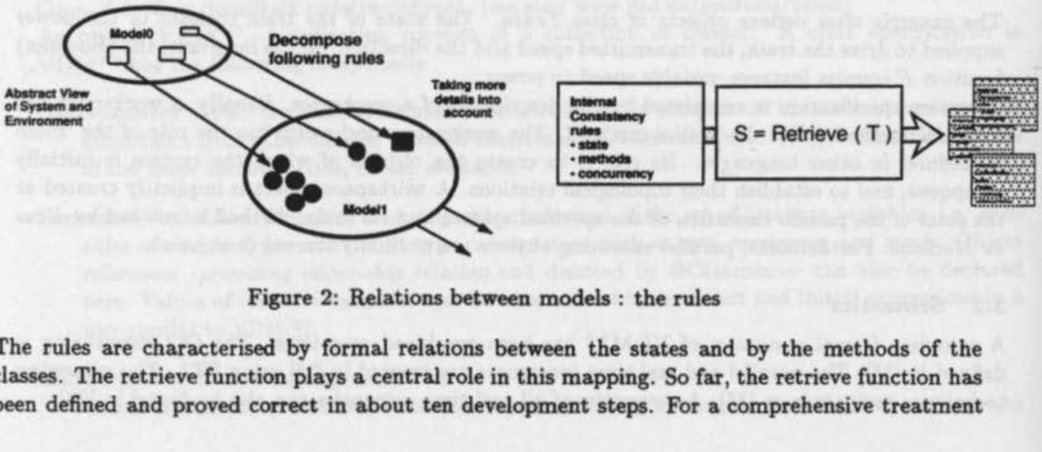
Model0
Decompose following rules
Abstract View of System and Environment
Taking more details into account
Model1
Internal Consistency rules
• state
• methods
• concurrency
S = Retrieve (T)

Figure 2: Relations between models : the rules

The rules are characterised by formal relations between the states and by the methods of the classes. The retrieve function plays a central role in this mapping. So far, the retrieve function has been defined and proved correct in about ten development steps. For a comprehensive treatment see [22]. We have frequently found that writing down the retrieve function for a development step is itself a demonstration of the validity of the transformation involved.

If the rules are applied properly, the design steps together constitute a derivation process scarcely in need of additional proofs. The final system, however, has a proven relation with the first model in its invariants and its method functionality.

For the future, we envisage that this set of annealing steps will help to structure the design process itself. Perhaps this is the first step of an important new direction, comparable to the "goto"-less and structured programming (r)evolution in the 80's. At the time, the programming level was concerned, this time it will be the design and specification level.

In summary the use of design refinement rules:

- limits the number of steps to be made
- structures the design process itself
- introduces a design based on refinement

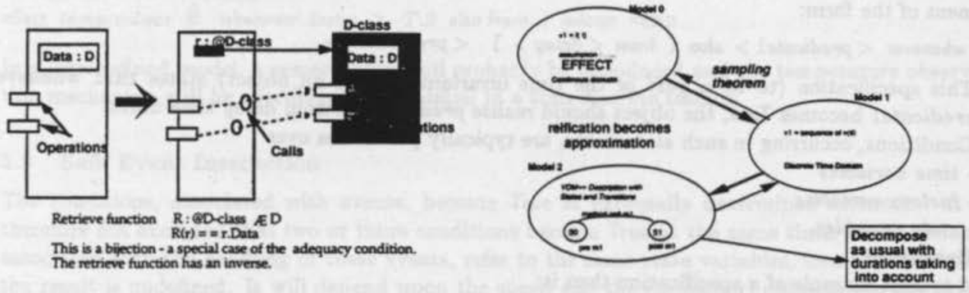
Data : D
Operations
r : @D-class
Calls
D-class
Data : D
tions
Retrieve function R : @D-class Æ D
R(r) = = r.Data
This is a bijection - a special case of the adequacy condition.
The retrieve function has an inverse.
Model 0
EFFECT
Continuous domain
sampling theorem
Model 1
Discrete Time Domain
reification becomes approximation
Model 2
VDM++ Description with States and Transitions
S0
S1
Decompose as usual with durations taking into account

Figure 3: Annealing: The Basic Step and the Real Time Cycle

In the special case of real Time systems, we envisage a preliminary stage in the design using time continuous quantities. Through application of a sampling mechanism the discrete version of this requirements model is derived. Then the decomposition strategy outlined above is followed with additional real time expressions. This design route will be shown in the case study in chapter 6.

5 Real Time Features in Detail

5.1 VDM^{++} Event handling

In reactive system designs often some form of an event driven mechanism is applied. 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.

Generally speaking, an event is the marker in time at which a predicate yields True. We distinguish:

1. an *observable* transition of the state of system. Within an OO specification this is typically a value change of one or more instance variables, possibly also a change in invariant or other expressions.

2. the transition of the systems' state resulting from handing a method request or method invocations. Here the changes are conceptually observable through *history variables*.

VDM^{++} has the following built-in events which are described by the atomic updates of the history variables:

- On each request to execute a method "methodname" an event occurs which is recorded by the system by atomically incrementing the implicit counter *#req(methodname)*
- the actual activation (start) of the method constitutes a second event which is noted by incrementing the implicit counter *#act(methodname)*
- the event defining the completion of the method execution is implicitly recorded by incrementing *#fin(methodname)*

The default available variables #req, #act and #fin record the transitions as counters. Two functions over history variables have been defined to ease specification of common situations:
#active(mn) = #act(mn) - #fin(mn) and #waiting(mn) = #req(mn) - #act(mn) .
Although method invocation in a periodic thread is not by an explicit call, the same events apply here.
VDM^{++} offers the capability of connecting an action to the occurrence of an event, using a statement of the form:

whenever < *predicate*1 > also [from < *delay* >] < *predicate*2 >

This specification (to be a part of the time invariant part of an object) states that whenever *predicate*1 becomes True, the object should realise *predicate*2 within *delay* time units.
Conditions, occurring in such statements, are typically predicates over
- *time variables*
- *history variables*
- *state variables.*

Examples

A typical example of a specification then is:
whenever *#act(methodname)*>0 also from δ_1 *#act(methodname)* $= 0$.
Here, an upper-bound for the execution of *methodname* is given. As soon as *methodname* is dispatched, the specification requires the implementation to ensure that the finalisation of the invocation of *methodname* is not more than δ_1 units of time later. The language provides a shorthand notation for this: duration(*methodname*) $\leq \delta_1$.
Obviously, having the notion of time, one can raise an alarm at a specified time
whenever *now* > *ThisTime* also from δ *#act(alarmMethod)* > *#fin(alarmMethod)*
This states that after a certain calendar time is reached, the execution of the alarm-method should be started within δ time units. Alternatively written: ... *#active(alarmMethod* > 0.
The next example shows a statement in which an unspecified predicate (indicated by '...' e.g. a resource is in use), is to become True within δ units of time from #act > #fin becoming True (the start of the method).
whenever *#act(methodname)* $\geq$ *#fin(methodname)* also from δ ...
To specify that execution of a method is urgent and must not be blocked for long one might write:
whenever #waiting(methodname) > 0 also from delta ...
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 impossible to fulfil: the predicate acting as consequent in the implication is to be made True in zero units of time.
It will be clear that periodic obligations can be specified using an event style:

```
 periodic($\Delta$ $T$)  (amethod) means :
forall $k$ in $Nat$ whenever   now $\approx$  $k * \Delta T$
     also from $\delta$   #active(amethod) > 0
```

with $duration(amethod) + \delta \leq \Delta T$ as a precondition, this specification activates the method at least once in each interval $< k * \Delta T, (k+1) * \Delta T >$

5.2 Time Variables

If sensor based systems are involved, the interface between the controlled system and the controlling system may contain variables that can be assumed to change spontaneously with time. Such variables are considered to be *Time continuous functions*. Their use is confined to high level models of the system and its environment which function as context for the high level requirements. On these variables assumptions and effects can be defined. The effect clauses is the equivalent of an invariant for time variables: it is a timed property which should always hold.

```
time variables
    temp : real;
    colour : RED—GREEN
effect temp, colour ≜ whenever temp ≥ T_0 also from δ colour =RED
```

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

5.3 Safe Event Interaction

The conditions, associated with events, become True at externally determined moments. It is therefore not excluded that two or more conditions become True at the same time. If the actions, associated with the handling of these events, refer to the same state variables, formally speaking, the result is 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. Passive classes may be protected against interference by concurrent method calls by the use of permission guards, defining the object states at which each method may be activated. These guards are mainly predicates over history variables.
The semantics of having more than a single *whenever* clause within a single thread deserves some attention. Current semantics allow multiple *whenever* clauses within such an object. In the event that two or more events become True at the same moment, unsafe situations show up when both events try to access/update the same instance variable or require (conflicting) method executions.

6 The Design Case

6.1 The problem description

We give here the full design route of a typical real time problem from the literature. The early highly abstract models are kept (too) simple, because we want to highlight the derivation methodology which is typical for VDM^{++} . In the Booch version (See also Mary Shaw's article [27] in which several methods and their solutions to the "Cruise Control system" by G. Booch are presented.) the Cruise Control System *(CCS)* has the following properties:
It exists to maintain the speed of a car even over varying terrain, when turned on by a driver. When the brake is applied the system must relinquish speed control until told to resume. The system must increase and decrease speed as directed by the driver. As a general condition we have *system*

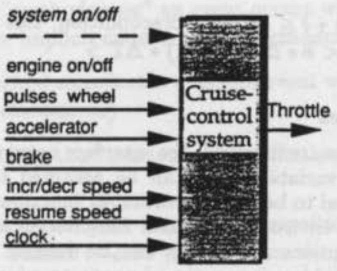

Figure 4: Block Diagram of the system

ON/OFF. ON denotes that the system should maintain the car speed. If OFF the system (functionality) does not exist: the power is off. The inputs are:

- *Engine* ON/OFF.
- *Brake* ON. When the brake is pressed, the CCS temporarily reverts to manual control.
- *Pulses from Wheel*. Pulses are sent each revolution of the wheel.
- *Increase/decrease speed*. The setting of the desired speed.

The **Output** :*Throttle* The Digital value for the engine throttle setting.

Defining the Speed

Mary Shaw rightfully mentioned, that a car's speed is not determined by the rotation speed (+ size) of only one wheel. In curves differential gears cause variations in rotational speed between the inner and outer wheels of the car. So we take here $\omega(t)$ is the angular velocity of the drive shaft. The quantity $\omega(t)$ is a *time continuous function* and because we can safely assume that the tire radius is constant over the lifetime of our CCS, it is the representation of the actual speed of the car. The desired revolution speed is called ω_d.

6.2 Goal description

The Cruise Control Requirement when the system = ON can now be defined as if *engine* = ON $\wedge$ *brake*= OFF then:

$$\begin{aligned} &\textbf{whenever } \omega(t) > \omega_d + \Delta_1 \textbf{ alsofrom } \delta_1\ \omega(t) \leq \omega_d + \Delta_1 \\ \wedge\ &\textbf{whenever} \omega(t) < \omega_d - \Delta_2 \textbf{ alsofrom } \delta_2\ \omega(t) \geq \omega_d - \Delta_2 \end{aligned} \tag{1}$$

In a graphical representation is given in figure 5.
There can be good reasons to take different values for the constants in the figure, but for simplicity we use: $\delta_1 = \delta_2 = \delta$ and $\Delta_1 = \Delta_2 = \Delta$. This simplification allows use to combine the two deviations in one line using the absolute value.

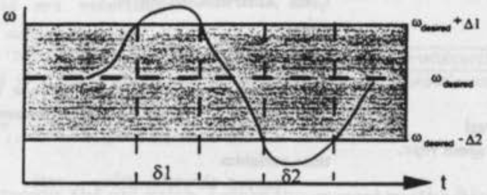

Figure 5: Velocity as time continuous function

6.3 The Abstract Controller: A Requirements Model

We can give the following specification for the closed system :

```
Class ControlledCar
values
    Δ, δ          : real
instance variables
    brake_state      : ON|OFF;
    engine_state     : ON|OFF;
    active           : boolean
  inv objectstate ≜    engine_state = ON   ∧   brake_state   = OFF
    ω_d              : Real;  - - the desired speed
time variables
  ω   :   real   -- the actual speed(t)
  effect object − state ≜    active ⇒   cite equation(1)
                   ∧ not active ⇒    ...   - -manual control
end ControlledCar
```

Remark

The fact that the *ControlledCar* should operate even over varying terrain makes it in principle an open system. To avoid very complicated formulas we will ignore this aspect in the next section when we define the car+engine behaviour. We use here the transfer function $\mathcal{H}(t)$ between speed and throttle value in the time domain.(We might have used differential equations instead.) We assume such a relation exists. Such a relation may be complicated due to air-resistance, external winds, surface resistance and up-hill/down-hill roads.

End of Remark

We will now make a distinction between the *Controller* and the *Controlled System*.

$\boxed{\text{Controller}} \rightarrow \textit{throttlevalue} \rightarrow$

$\leftarrow \textit{speed} \leftarrow \boxed{\text{CarEngine}}$

A CarEngine object hides the relation between the throttle-value and the speed. The controller observes continuously the deviation of the actual speed from the desired speed. It sets the throttle with a symbolic function $\mathcal{F}$. The splitting creates two classes.
In the workspace the following coupling is necessary between the time continuous variables in the created objects from both classes :

```
topology[ post aCarEngine · carspeed   = aController · speed
  ∧ aController · throttlesetting     = aCarEngine · throttlevalue]
```

Through substitution we can show that these two objects in the given topology have equivalent behaviour as the previous ControlledCar object. This splitting is thus a real refinement.

```
Class CarEngine
time variables
  input throttlevalue : real
        ω : real  --the speed repr.
   effect objectstate ≜
      carspeed = 𝓗(t) * throttlevalue
endCarEngine
```

```
Class AbstractController
values
            δ, Δ, c : real
instance variables
            active      :  ... -- see ControlledCar ;
            ω_d         : real;  -- desired speed
              init desired == ω_d := ...; -- init at creation time
time variables
          output throttlesetting : real
          input ω : real
  effect objectstate ≜ active ⇒
       whenever | ω − ω_d | > Δ   also fromδ
                    throttlesetting= 𝓕(ω, ω_d)
endAbstractController
```

6.4 The Design Route

We now focus on the design of the controller. It will be clear however that all reasoning (= requirement tracing and satisfaction proof) will need the CarEngine behaviour to reach the level of our initial requirements on the car speed. The above sketched *AbstractController* will be the start for the design process. In one overview the derivation process can be given as follows:

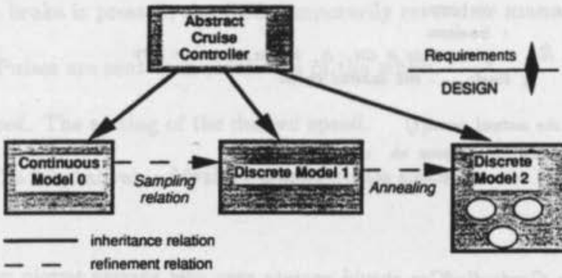

Figure 6: Design Process overview

• First a slightly more detailed ContinuousController is designed.
• Next the discrete version of this controller is specified.
• Finally, through an annealing step, the throttle and the speed sensor are put into separate objects which are connected to the controller.
All these versions of the Controller are subclasses of the abstract controller.

6.5 Continuous-Controller

The first Model of our Cruise Controller, operates conceptually in the time continuous domain, where the effect clause formalises the time dependent (dynamic) parts of the system requirements. We introduce for the symbolic function a (too) simple linear algorithm. The engine and brake dependency and the *setDesiredSpeed*() method are added here. This method will have to be implemented by each derived subclass in the implementation.

```
class ContinuousController
       is subclass of AbstractController
time variables
```

```
  -- $\omega$ and throttlesetting are inherited
 effect $\omega$, throttlesetting $\triangleq$ active $\Rightarrow$
   ( whenever $|\omega - \omega\_d| > \Delta$   also from $\delta$   throttlesetting = $\overline{throttlesetting} - C * (\omega - \omega\_d)$
 $\wedge$ whenever $|\omega(t) - \omega\_d| < \Delta$   also from $\delta$   throttlesetting = $\overline{throttlesetting}$
methods
  setDesiredSpeed(s :real) $\triangleq$ $\omega\_d$ := s
end ContinuousController
```

In VDM^{++} the $\overline{variable}$ means the old value, before the current method invocation. In this version, the effect clause is an refinement (or implementation) of the one in the AbstractController and must be consistent with it.
The system itself now consists of One Cruise Controller Object, which is created and activated. All values from the external world are still incorporated into one *monolithic object*. The main role of this $Model_0$ is to reveal the dependencies between the variables and to investigate all invariants. Invariants come in two versions: the well known static ones and the dynamic effect clauses constraining the time variables. A continuous class may have both.

6.6 Towards a Discrete Solution

Because the above stated requirements still operate in a time continuous domain, a transformation into an equivalent discrete object is necessary before we can apply OO and discrete software actions like method invocations, periodic executions etc. For this simple set of relations the descretising is not too difficult. The first activity which is needed in the analysis is the definition of the time resolution required for this problem.

6.7 Timing analysis

Suppose we use a sensor which generate a fixed number of pulses (e.g. 10) for each revolution of the wheel. The frequency of the pulses is then a linear function of the angular velocity and thus of the car speed. We propose here to use an implementation which determines the frequency by counting the pulses in a fixed interval.
Fault tolerance for errors such as wheels not rotating, or missed pulses would need other or additional sensor equipment.
With a tire size of say 33 cm (11 in.) we get a perimeter $\approx 2m$. A cruising speed of 72 km/h (= 20 m/s) $\rightarrow \omega \approx 60\, rad/s \approx 10\, revolutions/s$ gives a pulse frequency of 100 Hz.
A human driver would have a response time of about 0.1 sec between the observation of a deviation and the action. The car could accelerate to a new speed in about 1 sec. If we consider this as targets for the responses of the automatic system, we need to detect a change of speed in about 0.1 sec. If we decide to take a pulse count ten times a second we shall have 100 pulses per interval at 72 km/h. So the pulse resolution will yield about 1% measurement accuracy. This should be satisfactory. The accuracy decreases as the speed decreases, but at 36 km/h it still would be 2% which could be set as the lowest speed at which the automatic system is to work. For the revolution speed the minimum tolerance is thus $\Delta \geq 0.1\, rev/s (= 0.6\, rad/s)$
The highest working speed is determined by the highest frequency at which the pulse counter can work, which will be approximately the system clock pulse rate. A system clock of 1000 Hz will then fail above 720 km/h. This is probably sufficient for any car to be used in traffic.

Control Loop Frequency

If we accept a minimum speed of 36 km/h with the 2% error the control loop period is 0.1 s or 100 ms, or a frequency of 10 Hz. The periodic obligation is thus set on 100 ms.

The discrete Model can now be derived because we showed that the 100 ms repetition rate with a 1 ms clock accuracy is acceptable under the currently stated requirements. An analysis such as the one described above, is certainly one of the many safety analyses needed in any real time system.

6.8 Discretising

Conceptually the next step is to sample the time continuous variables introduced in the first model and generate a sequence of observations/samples approximating the time continuously variable. Our SW/HW system has to operate based on these series of samples. The following intermediate step shows in a tractable way that the design route is correct. We describe the transformation process here with a *sampler* object. This object creates from the time continuous variable w a time series of discrete observations stored in the sequence *wdiscrete*. Its behaviour is an abstract version of an A/D converter. In this problem the pulse generator on the wheel takes up part of this functionality.

```
class Sampler
instance variables
  wdiscrete : seq of real  --discrete time series
time variables
input  w          : real  -- continuous time variable
methods
takesample()
 ≜ [ post let mysample:real in
      ∃ t ∈  [n̅o̅w̅,now] ·  mysample = ω    ∧  wdiscrete = w̅d̅i̅s̅c̅r̅e̅t̅e̅ ⁀ [mysample] ]
timed post : duration(mysample) = δ_p ≪ 100
thread
  periodic(100)(takesample)
end Sampler
```

The $\overline{\text{now}}$ denotes the staring moment of the execution and now the end. The *time resolution* of the specification is defined as 1 *milliSecond*: this means that the periodic obligation fires here each 100 ms.

6.9 Derivation of the Discrete Model

This discrete model uses a series of samples on the rotational speed ω over time. We have now to refine/translate our effect clauses from the *AbstractCruiseController* into observations on this sequences of samples. Here again one is confronted with the trade-offs so common in engineering based on acceptable inaccuracies. One could state that formally that the $Model_1$ to be developed now, as again a monolithic but discrete representation of our problem is an *Approximation* of the previous model.
Given a sequence of samples : $\omega_1, \omega_2, \ldots, \omega_t$ in which ω_t is the most recent sample the requirement

```
whenever |ω(t) − ω_d | > Δ also from δ ...
```

has to be transformed into into a set of actions which should be taken each time when a new sample arrives. We assume here that the invariant is assured at the moment of ω_{t-1}. This is done by the system initialisation.
With the appearance of new sample ω_t three possibilities show up :

$\omega_d - \Delta < \omega_t < \omega_d + \Delta$	⇒no change needed
$\omega_t > \omega_d + \Delta$	⇒speed reduction needed
$\omega_t < \omega_d - \Delta$	⇒acceleration needed

Speed reduction and acceleration are achieved here by invoking a throttle control method with either a positive or a negative parameter. More intelligence can be built in by using higher order integration methods for the calculation of a new throttle value. The sequence length and the number of parameters has to be adopted accordingly of course. Here we limit ourselves to the simple case of two parameters: ω *and* ω_{rate}. Whether one wants to transfer the desired speed also to the throttle controller or not (and let it keep the old value) is a matter of taste and perhaps maybe of performance.
The *operationalisation of the effect clause* now transforms for the action part into:

$$\text{whenever} \quad \text{not } \omega\_d - \Delta < \omega\_t < \omega\_d + \Delta \quad \text{also from } \delta \quad \#\text{active}(throttleControl) = 1$$

Here again it is clear that we can only send a control signal to the throttle. The first part of the whenever clause is made operational by periodically checking the condition. Here the used sample value sequence (*omega*) contains always the last two samples of the previously defined unbounded sequence produced by the sampling *wdiscrete*.
This yields the following discrete specification, which is based on the ideal set of samples *wdiscrete*, without specifying how they are obtained. *omega*[2] is here always the last sample. Formally the relation between these sequences is: $omega(1) = wdiscrete(len(wdiscrete) - 1) \wedge omega(2) = wdiscrete(len(wdiscrete))$

```
class DiscreteMonolythicCruiseController
            is sub class ofAbstractCruiseController
values
 deviation = 0.6 ;
functions
in_range  R • R • R →  Bool
in_range(wt,wd,delta) == in_range = | wt - wd | > delta
instance variables
            omega : seq of real;
            wd    : real;
               inv omega ≜ len omega = 2
             rate : real
         inv rate,omega == rate = omega(2) -omega(1)
methods
 throttle_control ( w_1,w_2 :real) is not yet specified
 [timed post == duration(throttle_control) <  δ_3 ]

 handleSample()
  ≜   if active then
    if in_range(omega(2), w_d, deviation) then skip
     else throttle_control(omega(2) - w_d, rate)
 [ timed post : duration(handleSample) = δ_4]
thread
  periodic(100)( handleSample())
Aux Reasoning
 duration(throttel_control)  =  δ_3  <  δ_4
 duration(handleSample)      =  δ_4  <  100
end DiscreteCruiseController
```

6.10 The Discrete Second Model *Model*$_2$

We want to replace the sampling mechanism with an abstract representation of the real world sampler: *the Sensor Object* through "Annealing". The sensor will conceptually count continuously (e.g. using integration) the pulses and is interrogated by the controller each 100 ms. It produces a new value for the speed through the method *getsample*. The second step taken is to anneal the throttle controller via a separate object too. One could do the same thing with the other input variables like the engine and the brakes. For sake of simplicity we keep these values here internally inside our controller and assume that state changes on these variables are relayed through externally initiated methods calls of our controller:

```
class DiscreteCruiseController2
functions
in_range  R • R • R → Bool ...
instance variables
    aSensor        : @Sensor
  aThrottle        : @ThrottleController

  engine_state,brake_state : ON|OFF;
  active           : boolean
    inv objectstate ≜ active =
      engine_state = ON ∧ brake_state  = OFF;
  omega            : seq1 of real;
    inv omega ≜ len omega = 2
  w_d,rate                 : real;
      inv rate,omega ≜ rate = omega(2) -omega(1)

methods
  setEngine() is not yet specified;
  setBrake() is not yet specified;
  setDesiredSpeed(w_n :real)≜ [postw_d = w_n];

  doSample()
    == def asample = aSensor ! getsample() in
    (if active then
    (omega := omega ^ [asample] ;
    if in_range(asample, w_d, deviation) then skip
    else aThrottle ! setControl(asample-w_d,rate)))
  timed post : duration(doSample)=δ_5
thread
  periodic(100)(doSample())
Aux Reasoning
  duration(doSample) = δ_5  < 100
end DiscreteCruiseController
```

The duration requirements for the *setControl* method can be calculated if upperlimits for each of the statements in the *doSample* method are choosen.
The workspace of the final system creates one Controller, one sensor and one throttle controller, establishes the topology between these objects and starts their periodic threads.

6.11 Issues Under-specified

Overshoot **As in any discrete control system, special attention should be payed to overshoot and undershoot situations. Here only the maximum interval length appears in the specification. Such a requirement depends on unknown parameters and properties. Suppose the car is driving up-hill and does not have power enough to maintain the desired speed. It may take an undefined long time, until e.g. the top of the hill is reached. All this time the speed will be too low.**

Engine behaviour A second observation is the relation between throttle setting and speed. A more complete model (transfer functions from control engineering) is needed to ensure that the required behaviour of the car with cruise controller is satisfied. The requirement that the terrain may vary makes this extra complicated.

Undefined behaviour The current solution will expose unexpected (=undefined) transient behaviour, if the desired speed level is suddenly increased or decreased with large quantities. This is however outside the requirements scope.

7 Conclusions

We have shown, in the frame work of a well known (canonical) real time example, the novel approach [8] of using time continuous functions within a formal specification language as VDM^{++}. The derivation of the discrete solution through intermediate development steps clearly highlights the trade-offs involved in each design step. The ideal time continuous specification plays the role of the exact but un-implementable model. The derivation of the discrete models through common control and signal theory transformations, yields a correct approximation of the ideal model.
The high level time constraints have been carried along through out the entire design path. In the final model they end up as maximum processing time requirements of simple elementary methods. For each implementation in a given environment (OS,RT kernel, HW, etc.) these upper limits can be measured at unit level of the code (procedures, methods etc.). If these upper limits can be satisfied (including the necessary system overhead because their end time is specified !) the entire system will meet the upper level non-functional requirements (performance criteria).

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] E. Dürr, and N. Plat Editors (1995). VDM^{++} *Language Reference Manual*, Afrodite (ESPRIT-III project number 6500) document AFRO/CG/ED.LRM/V10, Cap Volmac, see: www.ifad.dk/projects/afrodite

[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] America, P. (1986) *Definition of the programming language POOL-T*, Esprit 415 document 0091, Philips Research Laboratories, Eindhoven

[7] Goldsack, S. (1991) *Distributed Object Orientation,* Journal of Object Oriented Programming, Volume 4, no.1 , March/April 1991.

[8] 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.

[9] B.Mahoney and I.J.Hayes, *A case-study in Timed Refinement: A mine Pump*, IEEE transactions on Software Engineering, Sept. 1992, pp 817-826

[10] C.Millerchip, B. Mahoney, I.J. Hayes, *A Whole System Specification of the Boiler System*, in D.Bel Belluz and H.C. Ratz(eds.), "Software Safety: Everybody's business- proc 1993 Int. Workshop on Design and Review of Software Controlled Safety Related Systems", 1994

[11] 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)

[12] 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.

[13] Eugène Dürr, N.Plat and Michiel de Boer ,*Tracking and Tracing Rail Traffic using* VDM++ : In M.G.Hinchey and J.P. Bowen Editors : Applications of Formal Methods, Chapter 9, Prentice Hall, September 1995 ISBN 0-13-366949-1.

[14] *Concurrent and Real Specifications in* VDM++ in : S.Goldsack and S. Kent Editors: Formal Methods in Object Technology Prentice Hall, March 96, ISBN 3-540-19977-2.

[15] 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.

[16] G. Bruno et al. *A New Petri Net based Formalism for specification, Design and Analysis of Real-Time Systems.* In: Proceedings IEEE Real-Time Systems Symposium, IEEE 1993 pp 294-301.

[17] L. Lamport. *The Temporal Logic Of Actions.* ACM Toplas, 16 (3) 872-923 1995

[18] Y. Kesten, A. Pnueli.*Time and hybrid State charts and their textual representation.* In: Proceedings 2-nd International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems. Vol 571 LNCS, pp 591-620 Springer Verlag 1992.

[19] Ghezzi G. , R. Kemmerer. *Astral: an Assertion language for Specifying Real-Time Systems.* Proceedings of the 3-d European Software Engineering Conference 1991, pp 122-146 1991.

[20] R. Gerber, Insup Lee. *A Layered Approach to Automating the Verification of Real-Time Systems.* IEEE Transactions on Software Engineering. Vol 18 no 9, september 1992, pp 768 - 784.

[21] Goldsack, K.Lano, E.H.Dürr, *Annealing and Datadecomposition* VDM++ ACM Sigplan Notices, Vol 31, No 7, July 1996

[22] S.Goldsack, K.Lano, E.H.Dürr, *Refinement of Object Structures in* VDM++ , from ftp://theory.doc.ic.ac.uk, get file at papers/Goldsack/annealing.ps

[23] S. Kent and R. Moore, *An Axiomatic Semantics for* VDM++ *: OO aspects*, Afrodite Document ,July ,1993 , AFRO/IC/SK/SEM-OO/V2

[24] K.Lano *Expressing the Semantics of* VDM++ *in RTL*, November 1994, AFRO/IC/KL/SEM2/V2

[25] K.Lano *Reasoning techniques in* VDM++ , November 1994, AFRO/IC/KL/RT/V2

[26] K.Lano, S.Goldsack, J. Bicarregui, S. Kent *Real-time Action Logic and Applications*, ECOOP 96, Linz Austria, The Proofs workshop, June 6 1996 to appear

[27] Mary Shaw, *Comparing Architectural Design Styles* in IEEE Software Nov. 1995

All AFRODITE documents can be retrieved electronically via www.ifad.dk/projects/afrodite.

About the authors

- Dr Ir E.H.Dürr is Technical Manager at Cap Gemini, Assistant Professor at Utrecht University -both in the Netherlands- and visiting Professor at the Ecole Nationale Superieure des Techniques Industrielles et des Mines de Nantes, France.(http://www.fys.ruu.nl/ durr)

- 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.