On the Use of VDM++ for Specifying Real-time Systems [*]

Marcel Verhoef [**] (Marcel.Verhoef@chess.nl)

Chess Information Technology BV, P.O. Box 5021, 2000 CA Haarlem, NL

Abstract. Language extensions have been suggested in the past to make VDM++ better suited for specification of real-time applications and tool support was developed to analyze these extended VDM++ models. Practical experiences with the language extensions and the supporting tools are discussed in this paper. Improvements to the language extensions and tool support are suggested.

1 Is there really a need for language extensions?

One of the important qualities of formal techniques is that abstraction is used to focus on the core properties of the system design. In particular the topics concurrency and real-time have a long standing tradition in academic research providing notations, tools and techniques to do just that. But it seems that the practical application of these solutions has not propagated sufficiently towards industry, particularly not to the mainstream of computing. The formal notations proposed by academia are in general very powerful and expressive but they are often not compatible with (or rather disruptive to) traditional system design, and in the past they did not scale up to the size of a typical industrial design problem which posed a high hurdle for their practical application.

One could, most likely successfully, argue that the traditional industrial design approach in many cases is fundamentally flawed and that the current way of working should therefore not hinder introduction of obviously superior techniques, even despite the scalability issue. It is the experience of the author however, that making small incremental improvement steps from the existing situation, bringing more and more formality into the design process, has a much better chance of acceptance in industry than the revolutionary approach.

The pragmatic introduction of formal techniques in combination with existing informal techniques and development processes is often referred to as "lightweight" or "invisible" formal methods [1,2]. The Danish company IFAD has been notably successful in the 1990's, marketing and selling their product

[*] 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.

[**] and Radboud University Nijmegen, Institute for Computer and Information Sciences, P.O. Box 9010, 6500 GL Nijmegen, NL. *http://www.cs.ru.nl/ita/*

VDMTools, in support of the VDM notation, leaving a track record of several very successful and large-scale industrial applications of VDM. In 2005, the Japanese company CSK bought the intellectual property rights to these tools and furthermore a new book appeared describing both the VDM++ notation and the tool, see [3].

These industrial VDM++ projects, which are partly discussed in the book, have shown that formal languages, tools and techniques are actually easy to adopt in practice, especially if they can be used in combination with informal techniques. They merely require additional training of the technical experts, assuming basic knowledge of mathematics and logic. These domain experts are normally rather open to new ideas, in particular if it directly supports their task at hand. The challenge in industry however is to become productive fast and bridging the gap between a concrete design problem and abstract specification always seems to be the bottleneck, often due to lack of experience.

The Model Driven Architecture, as proposed by the Object Management Group (see *http://www.omg.org/mda*), addresses this problem by creating so-called platform specific models (PSM), platform independent models (PIM) and a set of mapping rules between those models. PSMs are used to capture and communicate domain specific information, PIMs are abstractions that focus on some essential system properties. The mapping rules should facilitate automatic transformations between these models. Although this approach seems very appealing, it has not yet been proven to work in practice, in particular if the "semantic distance" between the PSM and PIM is big.

Another approach is to integrate different languages, such as the Circus language [4] which combines the formal notations Z, CSP and refinement calculus into a single paradigm. Alternatively, one could consider extending an existing notation such that it is easier to express domain specific problems directly in that language. Sometimes it suffices to introduce some "syntactic sugar" or modeling patterns to ease specification of some problem, but leaving the original language semantics intact. If this doesn't suffice, both the syntax and the semantics of the language need to be extended. However, the advantage over the MDA approach is still that large parts of the syntax and the semantics of the original language can often be reused.

Already in 2000, as part of the ESPRIT project VICE (VDM++ In a Constrained Environment), language extensions where proposed to facilitate specification of real-time systems in VDM++. These language extensions were accompanied by a special version of VDMTools which enabled the analysis of these enhanced models. Both the language extensions and the tool have been around for a while now but only very little experiences are reported on the use of either.

Purpose of this paper. A small case study was performed using the extended VDM++ notation and the supporting tools in the context of the BODERC project (Beyond the Ordinary: Design of Embedded Real-time Control) at the Embedded System Institute (see *http://www.esi.nl/boderc*). Without claiming to be complete, some valuable lessons learnt can already be drawn from this modeling exercise. These observations are intended to revitalize the debate on extending the VDM++ language for specifying real-time systems.

This paper is organized as follows. First, we explore how real-time systems can be modeled using the proposed extensions to VDM++. Some observations are made based on a case study and finally we suggest some improvements to the language and the supporting tools.

2 What is required for specifying real-time systems?

The unique selling point of formal modeling is that it allows early analysis of system properties. Problems identified during the requirements analysis and design phase are easier to fix than problems identified during test and integration, which saves costs and improves the time-to-market and product quality. However, in the area of real-time systems this is notoriously difficult to achieve. The reason for that is two-fold.

First of all, the general tendency is to focus on the functional requirements of the product first and foremost. The non-functional requirements (such as timeliness and throughput) are only considered in practice *after* the functional design is completed. Ask any software engineer how much time it takes to execute the code they wrote and in most cases they are not able to provide the answer. But more importantly, they often do not consider it to be their problem at all. The general belief is: “Surely, the hardware will be fast enough”. The well-known Moore’s law [5] has not helped to change this attitude; many projects were saved by the fact that available hardware at the time of product release was indeed faster than the state of the art at the time of design, or hardware has become so cheap that is was economically viable to increase the computing or communication capacity. This trend seems to be no longer feasible, in particular in the embedded systems area, see for example the on-line article [6]. Companies are forced to produce at increasingly lower cost to remain competitive in poor market conditions.

Secondly, the analysis techniques currently used in industry are not very effective. For example, worst-case execution time (WCET) analysis is indeed possible, but only if you have the source code, the compiler and know the target platform. These are available rather *late* in the design, in particular when the hardware is developed in parallel to the software, which is often the case in embedded systems. Modern CPU and System-on-Chip architectures have features (such as multi-stage instruction pipe-lining and advanced caching algorithms) where WCET analysis can only provide coarse results. Often benchmarking, running and measuring pieces of application code on the real target hardware, is the only practical solution to circumvent this problem. Alternately, one has to accept a large margin of error which will result in a conservative and over dimensioned design. Similarly, rate monotonic analysis (RMA, [7]) can be used to analyze schedulability, but only if all the tasks are periodic. Earliest deadline first (EDF, [8]) scheduling is known to be optimal for non-periodic tasks but no efficient implementation exists. The time-triggered architecture (TTA, [9]) does offer a solution to increase predictability by apriori distributing the capacity of the resources over all available tasks, but also at the cost of over dimensioning.

With respect to the first problem, the root cause is the notations used to design software. Performance is often considered a "second-class" citizen because there is no easy way to specify this type of requirements. For example, Real-time Object-Oriented Modeling (ROOM, [10]) only supports the notion of timers for which the semantics (e.g. resolution, accuracy) is depending on the target platform used. Selic et al claim that this is sufficient for soft real-time systems but they also admit that it is probably not suited (and was never intended) for hard real-time systems. Results obtained from simulation might differ significantly from the code running on the target environment. However, the situation is improving with the advent of UML 2 (which has borrowed a lot of concepts from ROOM) and the Profile for Schedulability, Performance and Time (available at *http://www.uml.org*). At least it is now possible to annotate software models with performance data using an industry-wide accepted notation. The mentality problem however, can only be solved by education and training.

With respect to the second problem, a lot of progress has been made the last few years. Model checkers such as FDR2 (*http://www.fsel.com*) and μCRL (*http://homepages.cwi.nl/~mcrl/*) can deal with very large state spaces, making analysis of concurrency practically usable. Similarly, timed-automata model checkers, in particular the tools UPPAAL and TIMES (*http://www.uppaal.com* and *http://www.timestool.com* respectively) are improving significantly as well, allowing analysis of realistic timing and schedulability problems. But all these techniques use an input language that is tailored for formal analysis rather than for design, which makes it hard to use by practitioners, even despite the nice graphical front-ends of the latter two tools. Moreover, the improved power of the tools itself has not reduced the inherent problem in this class of languages: even seemingly simple models can lead to large state spaces that are very hard to analyze. Expert advice, which is often lacking in an industrial environment, is often the only solution to find alternative modeling strategies that prevent (or circumvent) such situations.

Solutions are also proposed from the discrete event systems domain, where simulation based techniques are often used instead of model checking. It is often impossible to claim completeness during analysis because the state space of the model is potentially infinite which makes exhaustive simulation impossible. Nevertheless useful results can be obtained using these techniques. For example the Parallel Object Oriented Specification Language (POOSL, [11]) with its support tools SHESim and Rotalumis come from this domain. The language has been formally defined using timed probabilistic labeled transition systems and the simulation algorithm was proven to be correct with respect to the language semantics. The simulation algorithm consists of a two phase execution approach, where either processes execute asynchronously to completion or time passes synchronously for all processes at once. The language is sufficiently expressive to model timing requirements but these requirements have to be encoded explicitly, for example using modeling patterns. Matlab, which is well-known in industry (*http://www.mathworks.com*) provides functionality to specify timing requirements using Simulink and Stateflow. However, the notation is limited to state transition diagrams only and its semantics are determined by the type of mathematical solver used.

3 The VDM++ language

The book by Fitzgerald et al, [3], presents VDM++ using several case studies and deals with concurrency in chapter 12. Note that the real-time language extensions presented here are *not* discussed in the book. The official language reference manual is [12], which is also available at *http://www.vdmbook.com/tools.php.* The real-time language extensions are currently only described in [13,14] and are illustrated by the case study presented in [15]. We will repeat the essentials from these documents here for the sake of completeness and discussion.

Unfortunately, it is not possible to show the case study we performed in the BODERC project for reasons of confidentiality. In stead, the famous "Dining Philosopher" problem of Edsgar Dijkstra [16] will be used as a carrier throughout this paper. The story goes as follows: a group of philosophers joins for dinner at a table and each of them brings one fork. But the philosophers are only allowed to eat if and only if they have two forks, one in each hand. So, they have to borrow a fork from their colleagues to be able to eat. The aim is to find an algorithm that allows all the philosophers to eat, but without fighting about the forks, which are critical resources. A fork can be picked up by only one philosopher at a time. In contrast to the original "Dining Philosopher" problem, where only the fork of the adjacent philosopher at the table can be borrowed, here we do not care which fork is taken. For simplicity, a philosopher may take any fork from the table, as long as it is free. Furthermore, we have added an explicit limit to the number of times a philosopher can eat. This feature was introduced for practical reasons as well. While the original algorithm continues indefinitely, the algorithm presented in this paper always terminates, which makes it easier to study with VDMTools. Last but not least, we will also use the example as a carrier to introduce the notion of time in the specification, which goes beyond the scope of the original example.

3.1 Dealing with concurrency

VDM++ distinguishes *active* and *passive* objects. The latter only react when their operations are called, the former have their own thread of control and after start up they do not necessarily need interaction with other objects to continue. As shown in Figure 1, we will model the table as a passive object and the philosophers as active objects.

The class `Table` has an instance variable `forks` to count the number of available forks on the table. The operations `takeFork` and `releaseFork` are used to either pick up a fork or put it down again on the table. The operation `IamDone` is used to count the number of philosophers that are done eating. The constructor

```
class Table

instance variables
  forks : nat := 0;
  guests : set of Philosopher := {};
  done : nat := 0

operations
  public Table: nat ==> Table
  Table (noGuests) ==
    while forks < noGuests do
      ( guests := guests union
          {new Philosopher(self)};
        forks := forks + 1 )
    pre noGuests >= 2;

  public takeFork: () ==> ()
  takeFork () == forks := forks - 1;

  public releaseFork: () ==> ()
  releaseFork () == forks := forks + 1;

  public IamDone: () ==> ()
  IamDone () == done := done + 1;

  wait: () ==> ()
  wait () == skip;

  public LetsEat: () ==> ()
  LetsEat () ==
   ( startlist(guests); wait() )

sync
   per takeFork => forks > 0;
   per wait => done = card guests;
   mutex(takeFork,releaseFork);
   mutex(IamDone)

end Table
```

(a) Passive object: Table

```
class Philosopher

instance variables
  theTable : Table;
  turns : nat := 2

operations
  public Philosopher : Table ==> Philosopher
  Philosopher (pt) == theTable := pt;

  Think: () ==> ()
  Think () == skip;

  Eat: () ==> ()
  Eat () == turns := turns - 1;

thread
  ( while (turns > 0) do
      ( Think();
        theTable.takeFork();
        theTable.takeFork();
        Eat();
        theTable.releaseFork();
        theTable.releaseFork() );
    theTable.IamDone() )

end Philosopher
```

(b) Active object: Philosopher

Fig. 1. The Dining Philosophers – concurrency

Table puts a fork on the table, by incrementing the instance variable forks, for each instance of Philosopher that is created. At least two philosophers should show up for dinner because we need two forks at minimum to allow any philosopher to eat. Note that when the Philosopher is created in the constructor, its thread of control is not yet started. The threads of the active objects are started by executing the operation startlist in the operation LetsEat.

The class Philosopher has an instance variable turns to count the number of times (in our case: two) the philosopher wants to eat from the table. The philosopher has two basic activities, either he is thinking, represented by the operation Think or he is eating, represented by Eat. The thread clause specifies the so-called thread of control of the task. Any instance of class Philosopher will execute the algorithm specified here without needing extra external stimuli. The philosopher will first `Think`, then acquire two forks by calling `takeFork`, he will `Eat` and finally give the forks back by calling `releaseFork`. If the philosopher has iterated twice, then the `while` loop is finished and the philosopher will signal that he is ready by calling `IamDone` and the thread will terminate.

Note that the class `Table` does not have a `thread` clause which implies that its public operations are called and executed in the context of the thread of control of another, active, task. If such a passive task is shared among several active tasks, as is the case here (all the philosophers access the same table instance), special care needs to be taken when updating the internal state of the passive object. In the `sync` clause, so-called permission predicates (indicated by the `per` keyword) are given that specify under which circumstances the operations are allowed to be called.

For example `takeFork` is enabled if there are still forks left on the table. If an active task calls this operation while `forks` is zero then the thread of control of the active task is blocked until the value of `forks` becomes positive, for example when another philosopher returns a fork to the table by calling `releaseFork` from its own thread of control.

Similarly, the operation `wait` is blocked until all philosophers are finished eating. This situation is reached when the instance variable `done` is equal to the cardinality of the set of philosophers. Note that `done` is incremented by the operation `IamDone` which is called at the end of the `thread` clause of each philosopher. The `wait` operation is needed to suspend the thread of control of whoever called the operation `LetsEat`, in our case the user-interface of the VDMTools interpreter by calling `print new Table(3).LetsEat()` from the interpreter command-line. This is the appropriate way to start the simulation of the application for the case of 3 philosophers. Note that the instances of class `Philosopher` are *created* in the constructor of class `Table`, but their threads of control are *started* in the operation `LetsEat`.

Last but not least we can specify which public functions are mutual exclusive. In our case, we need to protect the instance variable `forks` to be changed by two threads at the same time. Therefore we have declared `takeFork` and `releaseFork` to be `mutex`. This also implies that both operations are not reentrant. Once a call to either operation is active, a second call to the same function will also block the thread.

The mutual exclusion keyword is actually "syntactic sugar" for a set of permission predicates defined over so-called history guards. For each operation a set of values is maintained that count:

1. how often was *operation name* requested (`#req`: *operation name* $\rightarrow \mathcal{N}$)
2. how often was *operation name* activated (`#act`: *operation name* $\rightarrow \mathcal{N}$)
3. how often was *operation name* finished (`#fin`: *operation name* $\rightarrow \mathcal{N}$)

Based on these numbers, some short-hand functions can be defined:

1. how many instances of *operation name* are currently running
   `#active`(x) = `#act`(x) - `#fin`(x)

2. how many instances of *operation name* are currently pending
   `#waiting`(x) = `#req`(x) - `#act`(x).

Using those definitions, the `mutex` synchronization clause can now be written as:

```
sync
  per takeFork     => #active(takeFork) + #active(releaseFork) = 0;
  per releaseFork => #active(takeFork) + #active(releaseFork) = 0
```

These history counters play an important role in the real-time extensions.

3.2 Dealing with real-time

The specification in Figure 1 is complete with respect to concurrency, but it does not say anything about time. In the context of the "Dining Philosopher" example, consider the fact that activities like thinking and eating actually take time. But how do we specify that? The `duration` statement was introduced in [14], which has the following syntax:

```
statement = DURATION ( numeral ) statement
          | block statement
          | ... ;
```

This notation allows to specify the execution time, indicated by the *numeral*, of the right-hand side *statement* according to a discrete time clock running at an arbitrary resolution. If we assume a clock resolution of 1 μsec in our model, then the execution time of the statement `duration(15) y := y * 3` is 15 μsec. Let us now reconsider our case study.

```
public takeFork: () ==> ()
takeFork () ==
  duration (5)
    forks := forks - 1;

public releaseFork: () ==> ()
releaseFork () ==
  duration (5)
    forks := forks + 1;
```

(a) handling the fork takes time

```
Think: () ==> ()
Think () ==
  duration (200)
    skip;

Eat: () ==> ()
Eat () ==
  duration (200)
    turns := turns - 1;
```

(b) and so does thinking and eating

Fig. 2. The Dining Philosophers – adding time

The specification has been extended with execution times, as shown in Figure 2, but what can we now do with it? An informal description of the operational semantics is provided here, a more formal description can be found in [13]. As explained in the previous paragraph, history counters are maintained for each operation of each instantiated object. The VDMTools interpreter maintains a log during symbolic execution (simulation) of the model. Each history counter that is encountered during execution is added to the execution log. The discrete time simulation clock is sampled and the inserted log entry is tagged with this time stamp. The execution log is offered as an ASCII output file to the user for further analysis, an excerpt is shown in Figure 3, highlighting the execution trace of a philosopher as specified in Figure 2.

```
req -> Op: Philosopher‘Think  Obj: 4  Class: Philosopher  @ 156
act -> Op: Philosopher‘Think  Obj: 4  Class: Philosopher  @ 156
fin -> Op: Philosopher‘Think  Obj: 4  Class: Philosopher  @ 356
req -> Op: Table‘takeFork  Obj: 3  Class: Table  @ 360
act -> Op: Table‘takeFork  Obj: 3  Class: Table  @ 360
fin -> Op: Table‘takeFork  Obj: 3  Class: Table  @ 365
req -> Op: Table‘takeFork  Obj: 3  Class: Table  @ 369
act -> Op: Table‘takeFork  Obj: 3  Class: Table  @ 369
fin -> Op: Table‘takeFork  Obj: 3  Class: Table  @ 374
req -> Op: Philosopher‘Eat  Obj: 4  Class: Philosopher  @ 376
act -> Op: Philosopher‘Eat  Obj: 4  Class: Philosopher  @ 376
fin -> Op: Philosopher‘Eat  Obj: 4  Class: Philosopher  @ 576
req -> Op: Table‘releaseFork  Obj: 3  Class: Table  @ 580
act -> Op: Table‘releaseFork  Obj: 3  Class: Table  @ 580
fin -> Op: Table‘releaseFork  Obj: 3  Class: Table  @ 585
req -> Op: Table‘releaseFork  Obj: 3  Class: Table  @ 589
act -> Op: Table‘releaseFork  Obj: 3  Class: Table  @ 589
fin -> Op: Table‘releaseFork  Obj: 3  Class: Table  @ 594
```

Fig. 3. Excerpt of an execution trace log file from Figure 2

Note that each time an "act" line is followed by a "fin" line in the logging, the time stamp at the end of the line indeed differs by the specified duration delay of that particular operation, as specified in Figure 2. It can also be seen from this trace that operations which are not explicitly tagged with a duration statement cost time nevertheless. For example, we see that calling an operation on another object (e.g. `Table‘takeFork` on the fourth line) costs 4 time units while calling an operation locally (e.g. `Philosopher‘Eat` on the tenth line) costs 2 time units. The interpreter uses a table to lookup the default execution time costs for each standard VDM++ language construct, if it is not overruled by a `duration` statement. This default execution time cost table can be modified by the user.

In the previous paragraph we have seen how active objects can be specified using the `thread` construct. However, many real-time systems have one or more threads that exhibit a typical periodic behavior. It could in fact be specified using the `duration` statement, but it would look rather artificial:

```
thread
  while (true)
    ( duration (10) someAction();
      duration (40) skip )
```

This specification would indeed imply a thread with a periodicity of 50 time units, but it would need to be "active" permanently and uninterrupted by other threads. Therefore, a new construct was added to the language where thread periodicity and task duration are explicitly decoupled. Consider the following definition:

```
thread definition = THREAD PERIODIC ( numeral ) ( name )
```

where *numeral* specifies the length of the period in time units and *name* specifies the name of the operation that is called with that frequency, for example:

```
thread periodic (50) (someAction)
```

Note that, compared to the weak previous attempt, we do not have to specify the duration of `someAction` explicitly anymore to guarantee the period of the thread. Every 50 time units the thread will execute `someAction`. It is possible to specify the duration of `someAction` independently by using `duration` statements inside the body of the operation. Note that the duration of the operation might even exceed the period of the thread. This situation is described in more detail in paragraph 4.7. To put the `periodic` statement in the context of the "Dining philosophers" problem, consider the situation where we would only like to specify the eating part of the algorithm, because only that activity involves manipulating the forks. This could be specified as shown in Figure 4.

```
class Philosopher

instance variables
  theTable : Table;
  turns : nat := 2

operations
  public Philosopher : Table ==> Philosopher
  Philosopher (pt) == theTable := pt;

  Eat: () ==> ()
  Eat () ==
    if turns > 0
    then ( theTable.takeFork();
           theTable.takeFork();
           duration (200) turns := turns - 1;
           if turns = 0 then theTable.IamDone();
           theTable.releaseFork();
           theTable.releaseFork() )

thread
  periodic (800) (Eat);

end Philosopher
```

Fig. 4. The philosophers eat every 800 time units

4 Case notes: some lessons learnt

The VDM++ language extensions for real-time have been presented in the previous paragraph and we will move on to discuss the usability of those language extensions. We will first concentrate on the tool support and then move on to the language itself. We have been using the VDMTools VICE edition, version 6.7.27, dated 6 November 2003.

4.1 Tool: multi-threading versus multi-processing

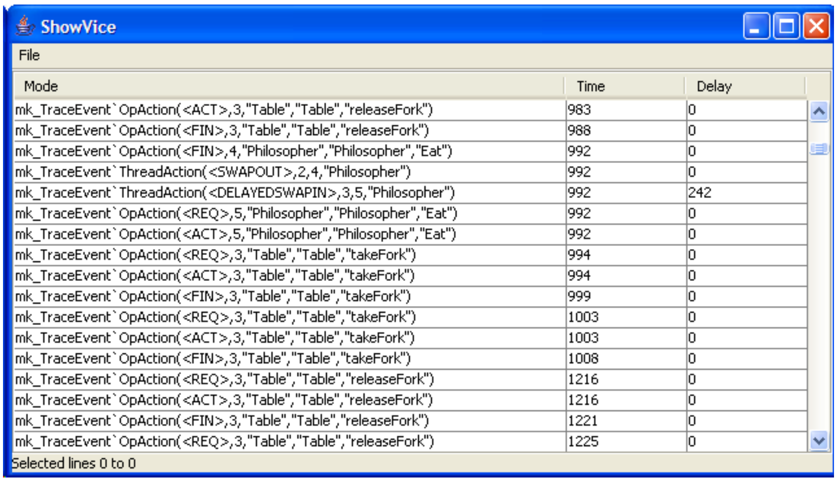
The current implementation in the VICE symbolic interpreter supports the notion of a single CPU. It is therefore only possible to simulate multi-threading (pseudo parallel behavior, interleaving semantics) and not multi-processing (true parallel behavior). According to [13], the operational semantics is sufficiently strong to deal with multi-processing but it simply has not been implemented. The absence of multi-processing in VDMTools causes a lot of problems in practice. First of all, single processor systems are nowadays an exception. Multiple computers connected through a network are common place, also in real-time systems such as for example military command and control systems or air traffic control systems.

Secondly, you have to specify the real-time system *in its environment* and therefore you have to specify the environment too. For that, the multi-processing approach is mandatory; the timing behavior of the real-time system should not be affected in any way by the execution of the environment simulation. In [15] this is partially solved by using `duration(0)` statements for all environment tasks, but it clobbers the specification and makes model maintenance unnecessarily cumbersome and artificial. Moreover, the delivery of external stimuli to the real-time system, specified in the environment model, is often as time critical as the system itself and therefore it should not be hindered by the execution of the real-time system model. It is much more natural to specify the environment as a separate process instead of a separate task.

4.2 Tool: only post-processing of time information

The VICE version of VDMTools only supports post-processing of time information, using the ASCII log file as shown in Figure 3. For any realistically sized problem it is impossible to analyze these files by hand. Tool support is required and some ad-hoc perl scripts were purpose built by IFAD [14,15] for a particular case study. However, if the model changes, then also the tools need modification and this is certainly not very efficient. We developed a general purpose visualization tool called `ShowVice`, a part of the user-interface is shown in Figure 5.

The tool is capable to read the execution trace file of any extended VDM++ model that is executed using the VICE version of VDMTools. The user can select the part of the execution trace that is of interest and produce a time annotated sequence diagram as is shown in Figure 6. As an example, we see the initialization phase of the model presented in Figure 4, which includes the first



Mode	Time	Delay
mk_TraceEvent`OpAction(<ACT>,3,"Table","Table","releaseFork")	983	0
mk_TraceEvent`OpAction(<FIN>,3,"Table","Table","releaseFork")	988	0
mk_TraceEvent`OpAction(<FIN>,4,"Philosopher","Philosopher","Eat")	992	0
mk_TraceEvent`ThreadAction(<SWAPOUT>,2,4,"Philosopher")	992	0
mk_TraceEvent`ThreadAction(<DELAYEDSWAPIN>,3,5,"Philosopher")	992	242
mk_TraceEvent`OpAction(<REQ>,5,"Philosopher","Philosopher","Eat")	992	0
mk_TraceEvent`OpAction(<ACT>,5,"Philosopher","Philosopher","Eat")	992	0
mk_TraceEvent`OpAction(<REQ>,3,"Table","Table","takeFork")	994	0
mk_TraceEvent`OpAction(<ACT>,3,"Table","Table","takeFork")	994	0
mk_TraceEvent`OpAction(<FIN>,3,"Table","Table","takeFork")	999	0
mk_TraceEvent`OpAction(<REQ>,3,"Table","Table","takeFork")	1003	0
mk_TraceEvent`OpAction(<ACT>,3,"Table","Table","takeFork")	1003	0
mk_TraceEvent`OpAction(<FIN>,3,"Table","Table","takeFork")	1008	0
mk_TraceEvent`OpAction(<REQ>,3,"Table","Table","releaseFork")	1216	0
mk_TraceEvent`OpAction(<ACT>,3,"Table","Table","releaseFork")	1216	0
mk_TraceEvent`OpAction(<FIN>,3,"Table","Table","releaseFork")	1221	0
mk_TraceEvent`OpAction(<REQ>,3,"Table","Table","releaseFork")	1225	0

Selected lines 0 to 0

Fig. 5. The user-interface of the ShowVice tool

period of the philosopher with object id 4. The threads of the active objects are highlighted using a different color for each active thread of control. It is easy to see which active object is calling operations on a passive object. Although the history counters are available, they are not shown in the diagram, only the operation request is annotated with the letter 'R'. If an operation is not eligible for execution, because a permission predicate is false, then this thread is swapped out and another thread gets control of the system. At time step 150 in Figure 5 (the second column containing the `Table` instance) we see that an operation is requested but cannot be executed. It turns out to be the operation `wait` which is called from inside the operation `LetsEat` in Figure 1. The `ShowVice` tool is available at *http://www.sf.net/projects/overture*.

4.3 Tool: the symbolic interpreter is slow

The VICE version of VDMTools is very slow compared to other discrete event simulation tools that the author is familiar with. Since the current implementation in VDMTools is closed source, it is not possible to check why there is such a large difference, but for realistic industrial models it is in any case not convenient. Performance much be increased by at least an order of magnitude to guarantee sufficient simulation depth to gain confidence in the model.

4.4 Language: timed consistency predicates

We already mentioned that only post-processing of time information is possible with the current version of the tool. But more surprisingly, it is impossible to *specify* timed consistency predicates because you cannot refer to the current

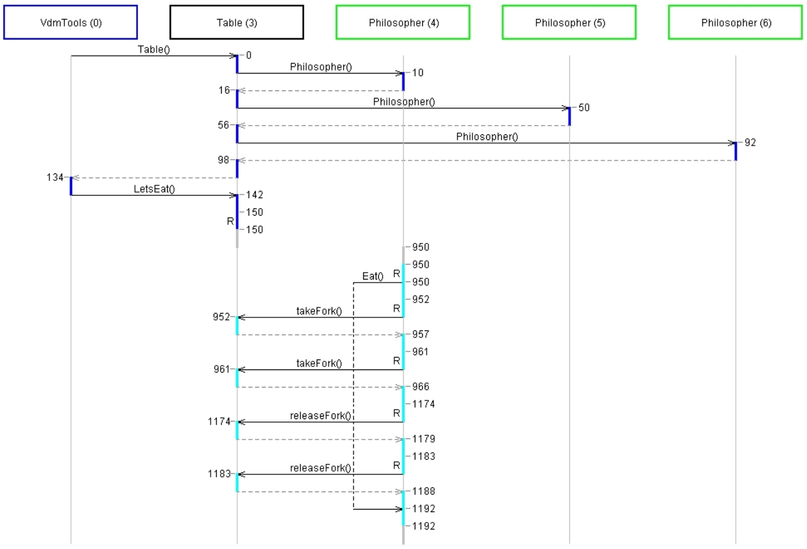


Fig. 6. A partial sequence diagram of Figure 4

value of the discrete time clock, neither implicitly nor explicitly. In the case of the "Dining philosopher" problem, suppose that we want to specify a fairness criteria, for example that each philosopher should eat within a certain time interval because the philosopher would otherwise not survive. How would we deal with that? We could for example introduce extra history counters:

- `#age`: *operation name* → $\mathcal{N}$, which returns the number of time units that passed since `#req` (*operation name*). Its use will be shown in paragraph 4.6.
- `#prev`: *operation name* → $\mathcal{N}$, which returns the number of time units that passed since the previous invocation of *operation name* or the total elapse time since the thread was started if it is the first time the operation was called.

If we would like to specify that the philosophers shall eat within 1000 time units then the specification of `Eat` should be changed as follows:

```
operations
  Eat: () ==> ()
  Eat () == turns := turns - 1
    pre #prev(Eat) <= 1000;
```

Note that this simple timed consistency predicate, which is added to the precondition of the operation `Eat`, would immediately detect at run-time that the default settings for the scheduler in the symbolic interpreter are not fair for our case study (it is possible to assign thread priorities and the size of the maximum time slice assigned to a thread). With the default settings, once a philosopher gets in control, he eats until he is done and only then gives up control. The third philosopher in our model would have died long before his turn is up. If the philosophers would eat in a round-robin fashion, which can be achieved by selecting the right scheduler settings in the VDMTools interpreter, then all philosophers would stay within the specified fairness criterion.

4.5 Language: modifying the thread properties

VDM++ only allows to start a thread (using the `start` and `startlist` operators); it is not possible to explicitly stop, suspend or resume a thread. Thread priorities are set once in an off-line configuration file and cannot be queried or changed at run-time. Consider for example Figure 4. Although the eating task is completed after two periodic iterations, the periodic thread keeps on calling `Eat` nevertheless because there is no possibility to stop the thread! Manipulating the thread properties is a must for real-time systems specification.

4.6 Language: interrupt the thread of control

A key property of real-time systems is that they can react to spurious events from the environment. The normal thread of control is suspended to deal with the event and the thread of control is resumed as soon as the event is handled. It is not possible to interrupt the thread of control of an active class. The thread of control is either waiting for some permission predicate to become true or it is continuously executing some operation. Interrupts can only be specified by explicitly encoding them using permission predicates in a dedicated active object. Modeling patterns are presented in [14,15] to deal with this, but this increases the model complexity unnecessarily. This situation can be improved substantially by introducing a new `interrupt` clause in a class definition that has the following syntax:

```
interrupt clause = INTERRUPT "[" interrupt definition { "," interrupt definition }+ "]" ;
interrupt definition = quoted literal "->" name ;
```

The interrupt clause is an ordered list of interrupt definitions. The order determines the interrupt priority in descending order. An interrupt definition is simply a mapping between an identifier (a quoted literal) and the name of the operation to execute when the interrupt occurs. In addition, we need a few extra statements, in order to raise and mask interrupts.

```
statement = SIGNAL ( quoted literal )
          | ENABLE ( quoted literal | ALL )
          | DISABLE ( quoted literal | ALL )
          | ... ;
```

In our case study, we could use this construct to allow the philosophers to have a beer or drink some wine during dinner:

```
operations
  drinkBeer: () ==> ()
  drinkBeer () == duration(20) skip
    pre #age(drinkBeer) < 5       -- maximum allowed interrupt latency
    post #age(drinkBeer) < 50;    -- interrupt deadline

  drinkWine: () ==> ()
  drinkWine () == duration(40) skip;

interrupt
  [ <BEER> -> drinkBeer, <WINE> -> drinkWine ]
```

Assuming that the interrupts are enabled, we could then cause the thread of control of each philosopher to be interrupted by executing the following statement (from within the scope of the passive class `table`):

```
for all guest in set guests do
  let s in set {<BEER>, <WINE>} in guest.signal(s)
```

The operational semantics for this construct need to be worked out in detail, which goes beyond the scope of this paper. For example, decisions must be made with respect to the interrupt queuing model. Note that interrupt latency and deadline requirements can be specified in the pre- and postcondition respectively, using the `#age` timed consistency predicate defined in paragraph 4.4.

4.7 Language: dealing with execution time uncertainty and deadlines

The `periodic` and `duration` statements where added to the language but they seem to be incomplete. Periodic behavior in real-time systems is typically described by three parameters: (1) the period, (2) activation latency and (3) the deadline. Activation latency is the amount of uncertainty on the exact scheduling moment. A task with average period 20 and latency 3 will actually occur between time units 17 and 23. Note that actual activation moment is a non-deterministic choice from the time interval $17, \ldots, 23$.

The deadline is the amount of time between the start of the periodic task and the finishing of a single periodic cycle, in the case of Figure 4, the elapse time between `#act`(`Eat`) and `#fin`(`Eat`). The `#age` history counter, which was introduced in paragraph 4.4, can be used to specify the deadline in the postcondition of the periodic thread operation, similar to the `drinkBeer` example in the previous paragraph.

Note that it is explicitly assumed in [13] that the execution time of the task is much smaller than the period, which relaxes the necessity for deadline specification. Currently, it is not detected automatically whether or not the previous invocation of the periodic task has completed already. Note that if an operation is not re-entrant (robust against multiple simultaneous invocations) that this needs to be encoded explicitly by creating a permission predicate for that operation, stating `#act(x) <= 1`. This permission predicate allows at most one active instance of operation x at any time. If the interpreter attempts to call x while it is still active, it will detect a deadlock and the simulation will stop.

To improve the usability of the `periodic` statement, the suggestion is to expand the syntax, where arguments are used to specify the period and latency respectively:

```
thread definition =

        -- pure periodic thread with period (1st argument)
        THREAD PERIODIC ( numeral ) ( name )

        -- periodic thread with period (1st) and latency (2nd)
      | THREAD PERIODIC ( numeral "," numeral ) ( name )
```

Surprisingly, the VDM++ language designers never considered a syntactic construct to define so-called sporadic threads. A sporadic thread is a thread for which only the *minimal* period is specified. It could be specified as follows:

```
thread definition = THREAD SPORADIC ( numeral ) ( name )
```

```
thread
  sporadic (1000) (dropFork)
```

The operation `dropFork` is periodically called, where the time between two execution moments of the task is at least 1000 time units. Note that the actual activation moment of the operation *name* is a non-deterministic choice from the time interval $1000, \ldots, \infty$.

With the `duration` statement, a similar situation arises as with the `periodic` statement. Duration is normally characterized by two parameters: the best-case (BCET) and worst-case execution time (WCET). Hence, the proposal is to allow duration with two parameters. Furthermore, it will be much more comfortable to allow an *expression* instead of a *numeral* as an argument to the duration statement, such that context dependent execution times can be specified naturally instead of forcing the use of a `cases` statement.

```
statement = DURATION ( expression ) statement
          | DURATION ( expression "," expression ) statement
          | ... ;
```

The type checker should ensure that the type of the expression still yields a numeral value. And at run-time the interpreter should verify that `BCET` $\leq$ `WCET`. The duration of the operation is a non-deterministic choice from the interval `BCET`,...,`WCET`. The impact of the non-deterministic extensions proposed here on the operational semantics needs to be analyzed further (for example approximation by a probabilistic variable), but since it has been done before in other languages (such as POOSL, which uses timed probabilistic labeled transition systems for its operational semantics), we do not expect that it will pose big problems.

5 Conclusions

Despite the criticism mentioned in the previous paragraph, the real-time extensions to VDM++ are very valuable, even despite the current poor tool support.

The extensions make it possible to annotate a VDM++ specification with timing information in a very natural and pragmatic way. The notation is conceptually simple, which makes it easy to explain and use. This is already a major step forward compared to current industrial practice.

The richness of the core VDM++ language in combination with the simplicity of the real-time language extensions are easier to use than languages such as for example POOSL and Stateflow. The former is certainly as expressive, but encoding the notion of time in the specification is not trivial. It also has a duration statement, which is called `delay`, but it requires in-depth knowledge of the operational semantics of the language to use it correctly. The style is more encoding rather than specifying. In the latter case the notation used is restricted to state transition diagrams in combination with the Matlab language. This language is also rich, but it is not very appealing to the software engineer because the notation is quite a step away from traditional software engineering techniques.

5.1 Future work and outlook

First of all, effort should be spent to automatically derive abstract concurrency and time specifications directly from VDM++ models. These abstract models could then be checked using state of the art model checkers, which are inherently much more potent to find errors than just discrete event simulation. The capability to generate counter examples to failure cases found in the specification is very powerful and certainly helps to increase the confidence in the model. Furthermore, this approach would also make model checking techniques more acceptable in industry because the abstract models are automatically derived from a model that is closer to their world. This philosophy has proven to work when the Prover tool was integrated with VDMTools in the Prosper project to discharge proof-obligations from the type checker (semi-)automatically.

Secondly, it makes sense to *stop* effort in extending the operational semantics of VDM++ for real-time. Instead, existing Discrete Event System simulators should be adopted because the amount of effort needed to implement the additional features that would make the tool really industrial usable can never compete with existing solutions that are already well accepted by industry. Two notable cases are SystemC and TrueTime. SystemC is an open-source platform for hardware/software co-design based on a C++ library and a high-speed simulation kernel. TrueTime is an extension to Matlab/Simulink that allows simulation of distributed real-time systems. It should be explored whether the already existing C++ code generator in VDMTools can be used to integrate with these solutions. Alternatively, the Java code generator could potentially be used together with the Ptolemy tool, providing an cheap alternative for Matlab/Simulink. The implicit advantage of this strategy is that VDM technology is opened up to two very large application domains: the hardware engineering community and the systems control community.

References

1. Jackson, D.: (invited talk) Lightweight Formal Methods. In: FME2001: Formal Methods for Increasing Software Productivity. Volume 2021 of Lecture Notes in Computer Science., Springer (2001)
2. Tiwari, A., Shankar, N., Rushby, J.: Invisible formal methods for embedded control systems. Proceedings of the IEEE **91** (2003) 29–39
3. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object–oriented Systems. Springer, New York (2005)
4. Oliveira, M., Cavalcanti, A., Woodcock, J.: Refining Industrial Scale Systems in Circus. In East, I.R., Duce, D., Green, M., Martin, J.M.R., Welch, P.H., eds.: Communicating Process Architectures 2004. (2004) 281–310
5. Moore, G.E.: Cramming more components onto integrated circuits. Electronics **38** (1965)
6. Sutter, H.: The free lunch is over: A fundamental turn toward concurrency in software. Dr Dobb's Journal **30** (2005)
7. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment. Journal of the ACM **20** (1973) 46–61
8. Buttazzo, G.C.: Hard Real-Time Computing Systems – Predictable Scheduling Algorithms and Applications. Kluwer Academic Publishers (1997)
9. Kopetz, H.: Real-Time Systems – Design Principles for Distributed Embedded Applications. Kluwer Academic Publishers (1997)
10. Selic, B., Gullekson, G., Ward, P.T.: Real-time Object-Oriented Modeling. John Wiley & Sons, Inc (1994)
11. van der Putten, P.H., Voeten, J.P.: Specification of Reactive Hardware/Software Systems. PhD thesis, Eindhoven University of Technology, Eindhoven NL (1997)
12. CSK: VDMTools – The VDM++ Language. CSK corporation, Japan (2005)
13. Mukherjee, P., Bousquet, F., Delabre, J., Paynter, S., Larsen, P.G.: Exploring timing properties using VDM++. In Bicarregui, J., Fitzgerald, J., eds.: Proceedings of the Second VDM Workshop. (2000)
14. IFAD: Development Guidelines for Real-Time Systems Using VDMTools. Institute for applied computer science, Forskenparken 10, DK-5230 Odense M, Denmark (2000)
15. Mukherjee, P.: A Counter Measures Example Illustrating the VICE Approach. Institute for applied computer science, Forskenparken 10, DK-5230 Odense M, Denmark (2000)
16. Ben-Ari, M.: Principles of Concurrent and Distributed Programming. Prentice Hall International (1990)

Acknowledgments

The author wishes to thank Erik Gaal (Philips Applied Technologies) for his constructive comments during the experiments and the writing of this paper. Thanks to Jozef Hooman, Frits Vaandrager, Oana Florescu and Peter Gorm Larsen for providing comments to early drafts of this paper.