Improvement Suggestions for VDMTools

(*Revision* : 1.1 – April 4, 2006)

Peter Gorm Larsen
PGL Consult
Miravej 3, Søften
DK-8382 Hinnerup
Denmark.
http://www.pglconsult.dk
email: `pgl@pglconsult.dk`

Marcel Verhoef
Chess Information Technology BV
P.O. Box 5021
2000 CA Haarlem
The Netherlands.
http://www.chess.nl
email: `Marcel.Verhoef@chess.nl`

Contents

1 Introduction

This memo intends to provide a series of proposals for enhancing VDMTools in its future innovative development. Because of the main emphasis on real-time embedded and distributed systems development from the CSK Corporation, the suggestions made in this memo focus on exactly that: enhancing VDMTools for development of such systems. The intent is that CSK Corporation can make use of the body of knowledge from a collection of VDM tool support experts (mainly placed throughout Europe, both industrial and academic users of VDMTools) and use them as external consultants and/or subcontractors. One can imagine this being arranged under either time-and-material contractual terms or on firm-fixed-price terms for one or more of the improvements.

Historically, VDM and all the other comparable model-oriented formal specification approaches (e.g. RAISE, Z and B) have primarily been used for the development of sequential computer based systems. The majority of formal approaches that have been used for the development of concurrent computer systems use explicit focus on the channels used for communication between the processes (e.g. CCS and CSP). A number of the formal approaches have been extended in different ways to include a notion of time (e.g. Timed CSP and the VICE VDM++ version). However, virtually none of the existing formal approaches are able to deal with combination of concurrency, time and distributed architecture in a way that would be appropriate for the development of real-time embedded and distributed systems. This memo includes suggestions for improving the VICE VDM++ technology with capabilities enabling exactly this combination and we strongly believe that if these suggestions are incorporated into VDMTools a really powerful capability enabling better development of realistic embedded systems will become available. In particular it will become easy for a specifier to experiment with different distributed architectures at a very early stage in the design and thus validate important properties in a cost-efficient fashion.

The VICE (VDM++ In a Constrained Environment) project was supported by the European Union in collaboration with Matra BAe Dynamics and it was carried out shortly before Peter Gorm Larsen left IFAD. This project aimed at improving the VDMTools capabilities in particular in the real-time embedded area and a lot of progress was made, but it was never properly commercialised because of Peter Gorm Larsen 痴 departure from IFAD. The application used for a case study in this project was a missile guidance system with a single processor. Thus, in this project there was not any focus on distribution between different processors. Nowadays there is however significant more emphasis on the distributed aspects of real-time embedded systems. The suggestions made in this memo are thus a continuation of the efforts made in the VICE project.

One of the triggers for this memo is the paper [1] written by Marcel Verhoef of Chess Information Technology (NL) that was presented at the first Overture workshop[1], entitled *"On The Use of VDM++ for Specifying Real-Time Systems"*. Marcel is a long-time VDM enthusiast; he made the first version of the type-checker for VDMTools when he was still a student at the Technical University Delft (NL) doing his MSc thesis project at IFAD in Denmark back in 1992. Later, he used VDMTools on several large scale industrial projects and he is also co-author of the recently published VDM++ book [2]. Currently he is part of the BODERC research project at the Embedded Systems Institute (see *http://www.esi.nl*) where he works on his PhD thesis. His PhD supervisor is professor Frits Vaandrager at the Radboud University Nijmegen (NL).

In [1], Marcel took the 6.7.27 version of the VICE tool and used it to model a large distributed industrial real-time embedded system (a digital control system for a high-volume office printer). The paper lists many areas where both the tool as well as the language could be improved to better fit this class of systems – in his belief many changes are mandatory in order to be sufficiently productive and successful in a commercial environment. The discussion at the conference let to a joint research session with Peter Gorm Larsen held at Aarhus in October 2005. We reviewed

[1]This paper is available at *http://www.cs.ru.nl/research/reports/info/ICIS-R05033.html*.

the problems identified and discussed the suggested solutions. This fruitful session actually let to several more insights that could solve problems that are not even mentioned in the Overture paper.

Currently, we are targeting a scientific paper for FM'06 that describes these (language) improvements. Basically, they can be categorised as follows (in random order):

- Enhancements to the visualisation capabilities of VDMTools;
- Enhancements to the static and dynamic analysis capabilities of VDMTools;
- Enhancements to the VDM++ notation and the corresponding tool support;
- Improvements to the UML coupling.

Prior to presenting the different categories this introduction is followed by listing the tasks that needs to be carried out to in order to demonstrate the proof of concept of the improve suggestions made in this memo. Afterwards, each improvement category will be detailed further below.

Finally, this memo is completed by a case study, describing a typical real-time embedded and distributed system. After introducing the case, we first show how a suitable VDM++ model can be produced with the existing VICE VDM++ technology. Then we shown how the VDM++ model could potentially be formulated and validated with the suggested improvements. At the end of the VDM++ model using the existing VICE technology it is also illustrated why it is hard with the currently available capabilities to discover where potential time requirements are violated and why this is the case. We believe that it is very likely that any developments made with VDMTools for embedded systems has a substantial risks for failure if the suggested improvements are not realised. Hopefully the new approach will be able to make it easier to validate such requirements and when violations are discovered faster to correct the model.

1.1 Proof of Concept of Enhanced VDM++ Real-Time Features

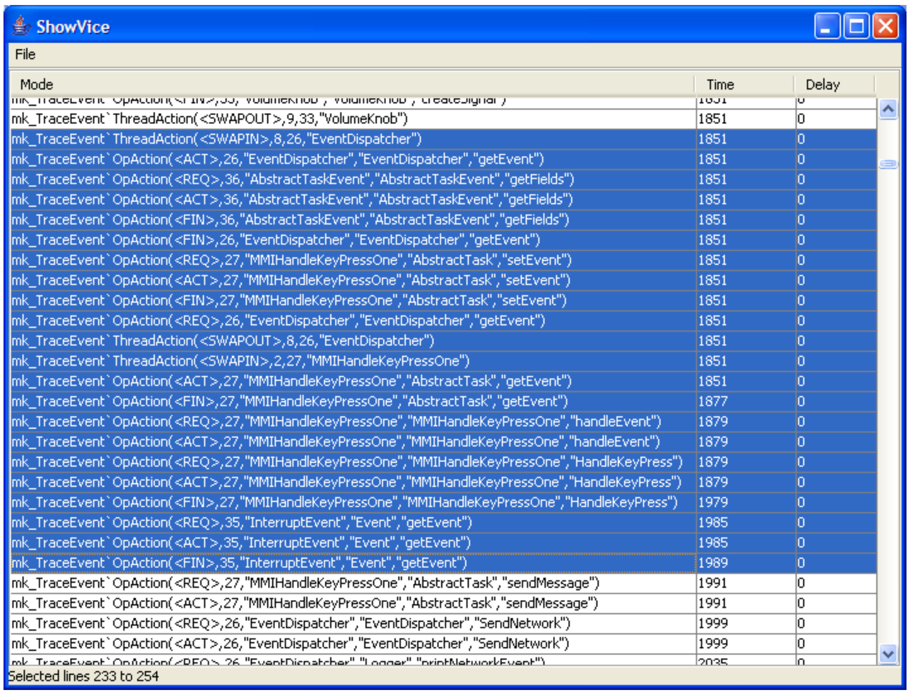
In order to assess whether the approach advocated in this memo for improving the VDMTools technology in the development of embedded real-time and distributed systems, this subsection describes the tasks suggested to be carried out to form a proof of concept. These tasks must produce:

1. An update of the VICE dynamic semantics specification reflecting the suggested semantic modifications in the improved real-time system;
2. A demonstration using the car radio-navigation example shown in this memo about the suggested improvements;
3. An updated scanner and parser that can produce the updated abstract syntax to the dynamic semantics specification;
4. An updated version of ShowVICE that will be able to demonstrate the car radio-navigation example and demonstrate the analysis desirable;
5. A high level product level functionality description that can be used to promote the use of VDMTools for the development of embedded real-time and distributed systems using the newly suggested approach; and
6. A guidelines document that at a more technical level is able to provide guidelines as to how the new sugegsted approach with advantage can be used for the development of embedded real-time and distributed systems. This document will take the existing guidelines document from the VICE project as a starting point.

It is recommended that the first 5 of these delivarables are produced by Peter Gorm Larsen and Marcel Verhoef jointly, whereas the last deliverable just as well could be made by Professor Araki's group (and in this way probably faster be translated into Japanese).

1.2 Enhanced visualisation capabilities

While writing [1], Marcel already produced a simple off-line support tool called *"ShowVICE"*. This stand-alone tool is able to display a UML sequence diagram (with time annotations) of the symbolic execution of the VICE model. This functionality is currently lacking in VDMTools but it is essential for gaining insight into the model, especially when concurrency and real-time issues come into play. Such functionality should definitely become an integrated part of VDMTools and should be enhanced further (for example adding a Gantt-chart view of the active tasks in the model, showing instance variable evolution over time etcetera). This kind of functionality provides important input to the user for validating the correct behaviour of embedded systems. It would also be valuable to have such sequence diagrams even in the traditional VDM++ interpreter (without time information). Two screenshots of the current ShowVice tool are provided in Figure 1 and Figure 2.



Mode	Time	Delay
mk_TraceEvent`OpAction(<FIN>,33,"VolumeKnob","VolumeKnob","createSignal")	1851	0
mk_TraceEvent`ThreadAction(<SWAPOUT>,9,33,"VolumeKnob")	1851	0
mk_TraceEvent`ThreadAction(<SWAPIN>,8,26,"EventDispatcher")	1851	0
mk_TraceEvent`OpAction(<ACT>,26,"EventDispatcher","EventDispatcher","getEvent")	1851	0
mk_TraceEvent`OpAction(<REQ>,36,"AbstractTaskEvent","AbstractTaskEvent","getFields")	1851	0
mk_TraceEvent`OpAction(<ACT>,36,"AbstractTaskEvent","AbstractTaskEvent","getFields")	1851	0
mk_TraceEvent`OpAction(<FIN>,36,"AbstractTaskEvent","AbstractTaskEvent","getFields")	1851	0
mk_TraceEvent`OpAction(<FIN>,26,"EventDispatcher","EventDispatcher","getEvent")	1851	0
mk_TraceEvent`OpAction(<REQ>,27,"MMIHandleKeyPressOne","AbstractTask","setEvent")	1851	0
mk_TraceEvent`OpAction(<ACT>,27,"MMIHandleKeyPressOne","AbstractTask","setEvent")	1851	0
mk_TraceEvent`OpAction(<FIN>,27,"MMIHandleKeyPressOne","AbstractTask","setEvent")	1851	0
mk_TraceEvent`OpAction(<REQ>,26,"EventDispatcher","EventDispatcher","getEvent")	1851	0
mk_TraceEvent`ThreadAction(<SWAPOUT>,8,26,"EventDispatcher")	1851	0
mk_TraceEvent`ThreadAction(<SWAPIN>,2,27,"MMIHandleKeyPressOne")	1851	0
mk_TraceEvent`OpAction(<ACT>,27,"MMIHandleKeyPressOne","AbstractTask","getEvent")	1851	0
mk_TraceEvent`OpAction(<FIN>,27,"MMIHandleKeyPressOne","AbstractTask","getEvent")	1877	0
mk_TraceEvent`OpAction(<REQ>,27,"MMIHandleKeyPressOne","MMIHandleKeyPressOne","handleEvent")	1879	0
mk_TraceEvent`OpAction(<ACT>,27,"MMIHandleKeyPressOne","MMIHandleKeyPressOne","handleEvent")	1879	0
mk_TraceEvent`OpAction(<REQ>,27,"MMIHandleKeyPressOne","MMIHandleKeyPressOne","HandleKeyPress")	1879	0
mk_TraceEvent`OpAction(<ACT>,27,"MMIHandleKeyPressOne","MMIHandleKeyPressOne","HandleKeyPress")	1879	0
mk_TraceEvent`OpAction(<FIN>,27,"MMIHandleKeyPressOne","MMIHandleKeyPressOne","HandleKeyPress")	1979	0
mk_TraceEvent`OpAction(<REQ>,35,"InterruptEvent","Event","getEvent")	1985	0
mk_TraceEvent`OpAction(<ACT>,35,"InterruptEvent","Event","getEvent")	1985	0
mk_TraceEvent`OpAction(<FIN>,35,"InterruptEvent","Event","getEvent")	1989	0
mk_TraceEvent`OpAction(<REQ>,27,"MMIHandleKeyPressOne","AbstractTask","sendMessage")	1991	0
mk_TraceEvent`OpAction(<ACT>,27,"MMIHandleKeyPressOne","AbstractTask","sendMessage")	1991	0
mk_TraceEvent`OpAction(<REQ>,26,"EventDispatcher","EventDispatcher","SendNetwork")	1999	0
mk_TraceEvent`OpAction(<ACT>,26,"EventDispatcher","EventDispatcher","SendNetwork")	1999	0
mk_TraceEvent`OpAction(<REQ>,26,"EventDispatcher","Logger","printNetworkEvent")	2035	0

Selected lines 233 to 254

Figure 1: The main user-interface of "ShowVICE", showing a parsed log file.

Figure 1 shows an overview of the VICE trace file as it was read back into the tool from the log file created by VICE after executing the specification. The user can select the appropriate portion of the log file that it wants to investigate, which is indicated by the highlighted part of the list box.

Figure 2 shows the time annotated sequence diagram of the selected portion of the trace file. In this particular diagram we see that a context switch occurs between the `EventDispatcher` and the `MMIHandleKeyPressOne` tasks. Also note that the execution of, for example, the `Event‘getEvent` operation takes exactly 4 time units. This graphical feedback is essential to understand the behaviour of the application.

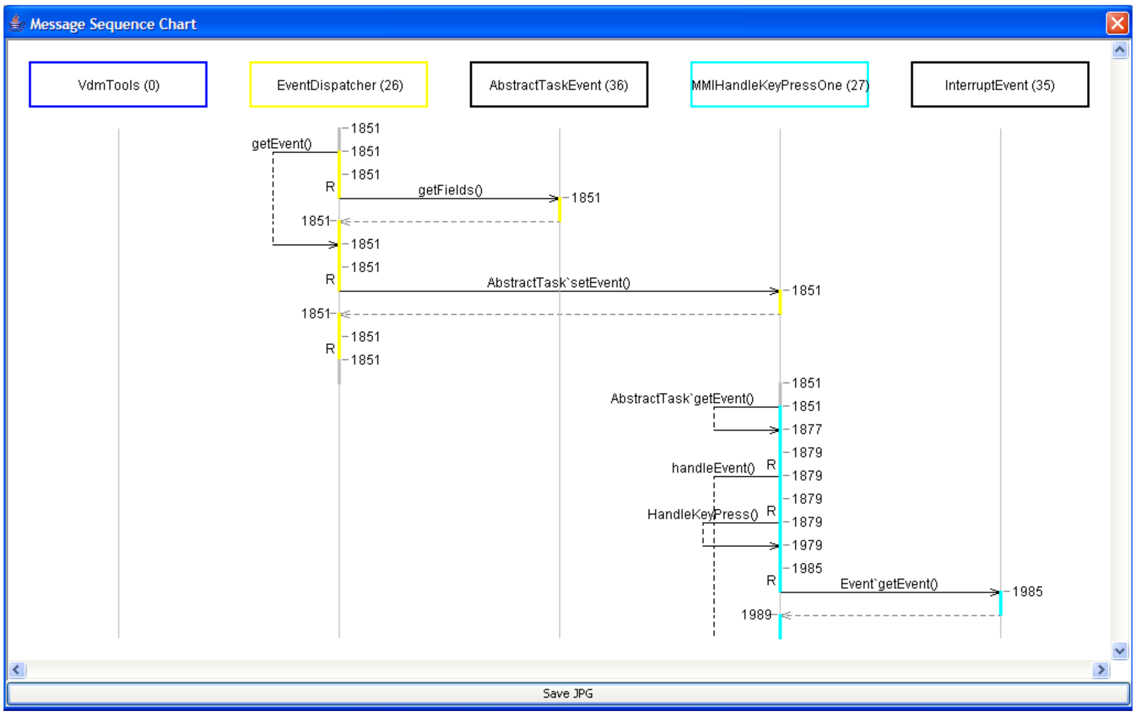


Figure 2: Detail screen showing a partial time annotated sequence diagram.

When this kind of functionality is incorporated inside VDMTools, it would also be extremely valuable to have support for predicates that can be expressed over these traces. Typically these predicates will be expressed with universal quantification and relating different events and the times of their occurrences (inspired by predicates from Real Time Logic [3]). It would be valuable and save time for users if the tool is able to directly demonstrate the counter-examples that did not satisfy the desired predicate(s), preferably graphically as shown in Figure 2. This is also the main reason why this visualisation functionality should be integrated into VDMTools and not kept as an off-line inspection tool, because when a predicate is invalidated, if possible the interpreter should be stopped immediately such that the cause can be analysed by inspecting the state of the model graphically as well as using the debugger command-line. This possibility is obviously not available in the off-line (post-simulation) based analysis.

1.3 Increased flexibility for trace file contents

The trace files produced by the current version of the VICE VDMTools are not in a format that is appropriate for easy post-analysis because a proprietary ASCII format is used. The "ShowVICE" tool thus has to parse these trace events before they can be analysed. This can be solved if the ShowVICE functionality is built into VDMTools as suggested above. But, the contents of each trace event are still quite limited, e.g. the arguments for operations invoked are not presented in the trace file. The reason for that is simple and pragmatic; it would slow down the execution of the interpreter too much if all information available inside the interpreter was dumped with each trace event.

Thus, the ideal solution would be to enable the user to select which information (s)he would like to trace during the simulation and under which circumstances it should do so. This could, for example, be implemented with a small scripting language enabling the user to configure the type and the amount of data to be logged in the trace file and thus be available for subsequent post-analysis (as a VDM value instead of a proprietary ASCII format). Some of the timing requirements for an embedded real-time system may otherwise not be expressible without this feature. An earlier draft of this note has already led to a significant improvement of VICE. The `time` keyword was added to the language, such that simulation wall clock is accessible from within the specification.

1.4 Increased support for analysis of deadlocks

Whenever a deadlock is reached in VDMTools, it can be hard for a user to determine what the cause of the deadlock is. Thus, the suggestion is to enable the user to inspect the state of the interpreter in this situation in a more convenient way. This could for example include:

- The ability to evaluate / debug the guards (synchronisation predicates) for the different selected operations that are active in that context
- Inspect the value of the history counters for the different operations in an easy manner
- The ability to inquire the status of the interpreter scheduler and all tasks

1.5 Improvement of the VDM++ language

The VICE notation was recently evaluated by Marcel in his paper [1] presented at the first Overture workshop explaining the problems when describing distributed embedded real-time systems with VICE. The main challenges he identified with the current notation are:

- The VDM++ notation is by nature synchronous; operation calls are either blocked on an permission predicate or executed in the context of the thread of control of the caller (an active object with its own `thread` clause in the class definition). This is very cumbersome when describing embedded systems, which are typically reactive (and therefore asynchronous) by nature. Of course, an event loop can be specified to simulate the asynchronicity, but it would clobber the specification greatly and lowers the productivity of the user. The specification is intrinsically bigger than necessary and the validation of the model is also more complex because of the increased model size. Users are typically overwhelmed by the amount of run-time data that such a "hand-coded" event loop generates. The language should therefore allow for *asynchronous* operation calls to overcome this problem.
- Similarly, the active object is currently the "owner" of the thread of control. This is ok when all threads are executed on the same processor (this is the current implementation in VICE) but it really complicates matters when distributed systems are modelled. Note that when describing embedded systems, this necessity is paramount, because the environment of the system can always be viewed as the "second processor" that works in full-parallel to and independent from the embedded system; both have their own behaviour and (timing)

requirements that are only influenced by the (timed) exchange of stimuli and responses. Therefore, the ownership of the thread of control should move from the "active class" to the (physical or virtual) processor on which the class instance is deployed. The behaviour of the system is then determined by the scheduling policy on that processor and its total capacity (available computing power). By doing this, it would be possible to describe deployment of software over a distributed computer system, also taking into account that one processor might be slower than another, leading to different response times. This would be a great step forward; as far as we know this has not been done before and it has a dramatic practical impact for describing industrial applications, where most systems nowadays are in fact already multi-processor solutions.

- Last but not least, we have some ideas that would also allow describing the interconnection between the processors (the network) at a very high-level of abstraction. Inter task communication is now considered instantaneous in VDM++ (or must be encoded explicitly using duration statements) which is far from reality. The network deployment view we envisage gives rise to include communication overhead into the simulation model at relative little extra cost (for both describing and analysing the system). In combination with the asynchronous operation calls, all typical styles of inter processor (and inter task) communication can be described: synchronous, asynchronous and publish/subscribe, while remaining at a high-level of abstraction when writing the specification.

We believe that these changes can be achieved with limited impact to the existing syntax and semantics of the language and the tool. We are currently working on a scientific paper that describes these conceptual changes to the VDM++ language. This should be seen as a "proof of concept" description that could be used as a guideline to the full-blown implementation in VDMTools. The first draft of the paper is expected to be ready in February 2006. A case study illustrating the effect of these suggested improvements is presented in Sections 2 and 3. However, we do emphasise that this is "work in progress". For example, the specification using the improved language has not been mechanically verified because tool support is simply lacking.

1.6 Enhancement of the UML coupling for VDMTools

The "Rose link" was originally made to couple VDMTools with Rational Rose. Meanwhile, UML has developed substantially and is now an updated standard - version 2.0. It would be very advantageous for all users of VDMTools that wish to use VDM++ in conjunction with class diagrams from UML if this new standard would be adopted. The enhancements could for example include:

- Rather than being bound to Rational Rose only, the UML link could be implemented using XMI (the UML model interchange format) such that all UML tools, that support this exchange standard for UML models, could be used (including public domain tools). This would remove the current "vendor lock-in" to Rational Rose for VDMTools. This is in particular important for the real-time and embedded market, where I-Logix Rhapsody seems to have a much larger market penetration than Rational Rose.

- The I-Logix company have produced a system engineering process called "Harmony" that seem very appropriate for the development of embedded real-time systems. This method and the recently released SysML (the Systems Engineering modelling language, also from the OMG in collaboration with the International Council of Systems Engineers INCOSE) should be analysed further for better integration with VDMTools for the early phases of development.

- As a minor point, support for packages and hierarchical classes in UML 2.0 could be taken into account in VDMTools. Similarly, state transition diagrams from UML could easily be imported into a VDM++ model, provided that asynchronous operation calls are supported in the language.

The round-trip engineering approach advocated by the existing VDMTools is still very worthwhile in the context of UML 2.0. The new UML notation still has no agreed syntax (and semantics) for the specification of (for example) operations – which is a natural fit with VDM++, as described for example in the VDM++ book [2]. Do note however that Stephen Mellor is leading an OMG activity currently to define a (new) language for the creation of executable UML models, something that VDM++ already provides but the majority of UML users is probably not aware of. To prevent OMG re-inventing the wheel once more (as they did with OCL), it is suggested that CSK, in collaboration with their academic partners, makes a case for acceptance of VDM++ as a suitable and already existing and industry proven solution for the specification of executable UML models. This should be actively pursued and pushed as soon as possible.

OMG also promotes the Model Driven Architecture approach to systems analysis and design. While the coupling to UML already implements parts of that strategy, the current VDMTools implementation is not well-suited for this. It was just not designed with MDA in mind. The Overture open source initiative (*http://www.overturetool.org*) attempts to solve part of this problem by providing access to a standardized XML format for the abstract syntax of VDM specifications. We suggest that the Overture effort is actively supported from within CSK. Overture is the ideal platform to get academic researchers involved in the development of new tools and ideas as proposed in this note. It can become a breading ground for new ideas and an excellent opportunity for community building.

The above two paragraphs indicate that the current work on UML 2.0 and MDA is pretty much orthogonal to the improvements to VICE proposed in this note. These issues are not hampering each other but the relationship between VDM++, UML and MDA can certainly be strengthened by implementing the suggested language and tool improvements. In our opinion, it will also leverage the market potential for VDMTools.

2 A case study – using VICE

The case study presented in this section is used to analyse the performance of a distributed in-car radio navigation system. The case study was originally modelled using the Modular Performance Analysis technique, as described in the paper *"System Architecture Evaluation Using Modular Performance Analysis – A Case Study"* by Wandeler et al [4]. We refer to this paper[2] for a full description of the case study.

In this note, an attempt was made to describe the same case study using VDM++, first with the original VICE language and later we will present the proposed extensions to the language. A UML class diagram of the case study using the existing VICE technology is shown in Figure 3. In Figure 4, a class diagram of the same case study is shown, but then using the suggested improvements to the VICE language. It is immediately clear from those two diagrams that the suggested improvements significantly reduce the size of the model. But more importantly, the model using the languages extensions describes the system in far more detail than the standard VICE specification. The latter can only describe the single CPU architecture case (and already with some difficulty because the environment tasks are not really running in parallel to the embedded system) while the former is able to describe *all* possible distributed architectures describe in [4]. Note that our discussions with the CSK team have already led to changes in the VICE tool. For example, the `time` keyword was added which allows the user to access the simulation wall clock of the interpreter directly.

The in-car radio navigation system is basically a soft real-time system that executes several concurrent applications at the same time (each consisting of several tasks), for example for controlling the radio while Traffic Message Channel data is being processed. Timing requirements

[2]Available at *http://www.cs.ru.nl/research/reports/info/ICIS-R05005.html*. Additional information can be found at *http://people.ee.ethz.ch/∼leiden05/data/pset/p2.pdf* and at *http://www.mpa.ethz.ch* .

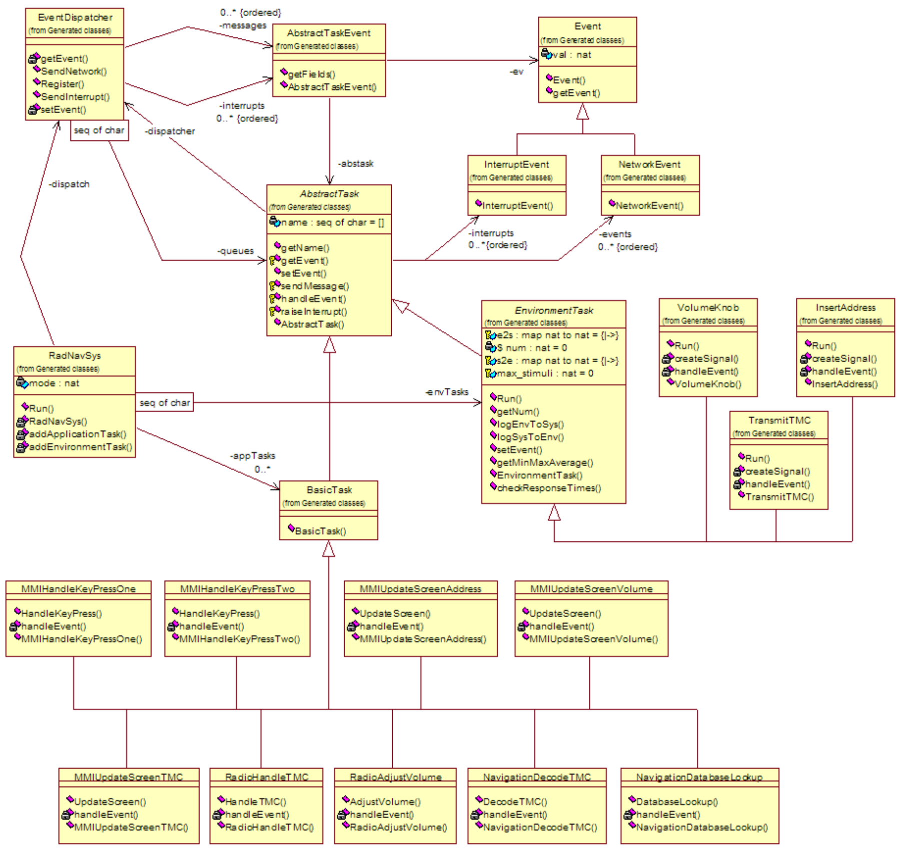


Figure 3: UML class diagram of the case study using standard VICE

are typically specified per application and the question of interest is whether or not these requirements are met under all operating conditions, when deployed on a given architecture. In this note we will consider the situation where all applications are running on a single CPU, because it is the only configuration that we can effectively describe using the existing VICE technology. VICE only supports uni-processor multitasking models, while some architectures considered in the paper above require a notion of multi-processor multitasking. First, we will present the model in the existing VICE notation. In Section 3 we will present the model in the improved VICE notation.

2.1 A mini-framework for reactive systems modelling

The in-car radio navigation system is a typical real-time embedded system; it is waiting for stimuli from the environment and processes those events accordingly. Some stimuli only change the internal system state but most will cause a response back to the environment. In our case study all stimuli will cause an external visible response.

Timeliness requirements are often specified in terms of the elapse time between the stimulus and the response of the system. Such a system is typically described by an event loop; the system is waiting for stimuli and when one arrives, it is identified and the appropriate operation is called and the event loop is immediately resumed. For this to work in real-time systems, the operation calls shall be asynchronous because the event loop should never be blocked by the call because it would potentially delay a high priority event that arrived just after the event that is currently being processed. Since neither VDM++ nor VICE supports asynchronous operation calls, we have to mimic this behaviour by using threads and explicitly describing the event loop.

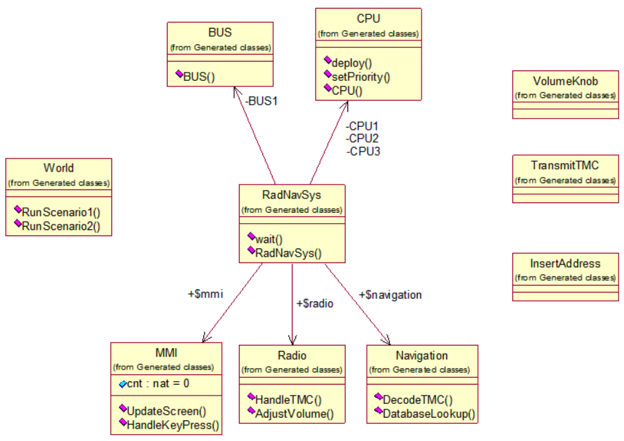


Figure 4: UML class diagram of the case study using the improved VICE notation

The Event class is the abstract base class for the events that are managed by such an event loop. A natural number is used to “trace” each individual event through the system, such that we can specify timing (and temporal) requirements easily.

```
class Event

instance variables
  val : nat

operations
  public Event: nat ==> Event
  Event (pv) == val := pv;

  public getEvent: () ==> nat
  getEvent () == return val

end Event
```

Two different event types are modelled here: interrupt and network events. Interrupts are used to trigger the system from the environment, network events are used to model inter task communication within the system. As we will see later, interrupts *have priority* over network events.

```
class InterruptEvent is subclass of Event

operations
```

```
  public InterruptEvent: nat ==> InterruptEvent
  InterruptEvent (pne) == Event(pne)

end InterruptEvent
```

```
class NetworkEvent is subclass of Event

operations
  public NetworkEvent: nat ==> NetworkEvent
  NetworkEvent (pne) == Event(pne)

end NetworkEvent
```

The `AbstractTask` class provides the basic functionality to handle events. Two separate input queues are maintained, one for interrupts and one for network events. There will be a single `EventDispatcher` instance that will dispatch all system events to the appropriate active object. The EventDispatcher can raise interrupts or send network events by calling the public `setEvent` operation of the applicable `AbstractTask` instance. The operations `getEvent` and `handleEvent` are used to implement the event loop, as we will see later. Note that `getEvent` gives priority to interrupts over network events; calls to the operation will be blocked (because of the synchronisation predicate) until there is at least one event available. The `AbstractTask` can send messages (or raise interrupts) to other `AbstractTask`s by calling `sendMessage` or `raiseInterrupt`. In both cases, the dispatcher will be called to route the message to the receiving task without blocking the caller.

```
class AbstractTask

instance variables
  -- keep the name of the task for easy logging
  name : seq of char := [];

  -- the queue for normal events to be handled by this task
  events : seq of NetworkEvent := [];
  -- the queue of high-priority events to be handled by this task
  interrupts : seq of InterruptEvent := [];

  -- a link to the dispatcher for out-going messages (events)
  dispatcher : EventDispatcher

operations
  public AbstractTask: seq of char * EventDispatcher ==> AbstractTask
  AbstractTask (pnm, ped) == atomic ( name := pnm; dispatcher := ped; );

  public getName: () ==> seq of char
  getName () == return name;

  -- setEvent is a call-back used by the EventDispatcher to insert
  -- events into the appropriate event queue of this AbstractTask instance
  public setEvent: Event ==> ()
  setEvent (pe) ==
    if isofclass(NetworkEvent,pe)
    then events := events ^ [pe]
    else interrupts := interrupts ^ [pe];

  -- getEvent is called by the event loop of this AbstractTask instance to process
  -- incoming events when they are available. note that getEvent is blocked by a
  -- permission predicate (see sync) when no events are available and also
  -- note that getEvent gives interrupts priority over other events
  protected getEvent: () ==> Event
  getEvent () ==
    if len interrupts > 0
    then ( dcl res: Event := hd interrupts;
           interrupts := tl interrupts;
           return res )
    else ( dcl res: Event := hd events;
           events := tl events;
           return res );

  -- handleEvent shall be overloaded by the derived classes to implement
  -- the actual event loop handling. a typical event loop handler would be
  -- thread while (true) do handleEvent(getEvent())
  protected handleEvent: Event ==> ()
  handleEvent (-) == is subclass responsibility;

  -- sendMessage is used to send a message to another task
  -- typically used for inter process communication
  protected sendMessage: seq of char * nat ==> ()
  sendMessage (pnm, pid) == dispatcher.SendNetwork(name, pnm, pid);

  -- raiseInterrupt is used to send a high-priority message
  -- typically used to communicate from environment to the system or vice versa
  protected raiseInterrupt: seq of char * nat ==> ()
  raiseInterrupt (pnm, pid) == dispatcher.SendInterrupt(name, pnm, pid)

sync
  -- setEvent and getEvent are mutually exclusive
  mutex (setEvent, getEvent);
  -- getEvent is blocked until at least one message is available
  per getEvent => len events > 0 or len interrupts > 0

end AbstractTask
```

The `BasicTask` class implements the event loop. It is an active object (with its own thread of control) that is constantly processes incoming events. The task will be blocked when no events are available, due to the synchronisation predicate specified in the base class.

```
class BasicTask is subclass of AbstractTask

operations
  public BasicTask: seq of char * EventDispatcher ==> BasicTask
  BasicTask (pnm, ped) == AbstractTask(pnm, ped);

-- BasicTask just implements the standard event handling loop
-- handleEvent is still left to the responsibility of the subclass of BasicTask
thread
  while (true) do
    handleEvent(getEvent())

end BasicTask
```

The `EnvironmentTask` class is used to model tasks in the environment, outside the scope of the system. EnvironmentTask instances generate the stimuli for the system and observe the responses coming back. Both stimuli and responses are administered using the `logEnvToSys` and `logSysToEnv` operations respectively. The function `checkResponseTimes` is used to check the system timeliness requirements. We will see later how it is used. The operation `getNum` is used to create unique identifiers, one for each stimulus that is generated and inserted into the system. Recent discussions with the CSK team has led to a change in the existing VICE tool already. The `time` keyword was added to refer to the simulation wall clock of the interpreter. This feature is used in the `logEnvToSys` and `logSysToEnv` operations.

```
class EnvironmentTask is subclass of AbstractTask

instance variables
  -- use a unique identifier for each generated event
  static private num : nat := 0;

  -- we limit the number of inserted stimuli
  protected max_stimuli : nat := 0;

  -- administration for the event traces
  -- e2s is used for all out-going stimuli (environment to system)
  -- s2e is used for all received responses (system to environment)
  protected e2s : map nat to nat := {|->};
  protected s2e : map nat to nat := {|->}

functions
  -- checkResponseTimes verifies for each received response whether
  -- or not the elapse time did (not) exceed the user-defined limit
  public checkResponseTimes: map nat to nat * map nat to nat * nat -> bool
  checkResponseTimes (pe2s, ps2e, plim) ==
    forall idx in set dom ps2e &
      ps2e(idx) - pe2s(idx) <= plim
  -- the responses received should also be sent
  pre dom ps2e inter dom pe2s = dom ps2e

operations
  public EnvironmentTask: seq of char * EventDispatcher * nat ==> EnvironmentTask
  EnvironmentTask (tnm, disp, pno) == ( max_stimuli := pno; AbstractTask(tnm,disp) );

  public getNum: () ==> nat
  getNum () == ( dcl res : nat := num; num := num + 1; return res );
  -- setEvent is overloaded. Incoming messages are immediately handled
  -- by calling handleEvent directly, in stead of added to an input queue.
  public setEvent: Event ==> ()
  setEvent (pe) == handleEvent(pe);

  -- Run shall be overloaded to implement the event generation loop
  -- towards the system. typically, it starts a periodic thread
  public Run: () ==> ()
  Run () == is subclass responsibility;

  -- logEnvToSys is used to register when an event was inserted into
  -- the system. note that the 'time' keyword refers to the internal
  -- simulation wall clock of VDMTools
  public logEnvToSys: nat ==> ()
  logEnvToSys (pev) == e2s := e2s munion {pev |-> time};

  -- logSysToEnv is used to register when an event was received from
  -- the system. note that the 'time' keyword refers to the internal
  -- simulation wall clock of VDMTools
  public logSysToEnv: nat ==> ()
  logSysToEnv (pev) == s2e := s2e munion {pev |-> time};

  -- getMinMaxAverage calculates the minimum, maximum and average
  -- response times that were observed during execution of the model
  -- note that getMinMaxAverage is blocked until the number of
  -- system responses is equal to the number of sent stimuli
  -- termination is ensured because only a maximum number of stimuli
  -- is allowed to be inserted in the system, so eventually all
  -- stimuli can be processed by the system. this method only works
  -- when each stimulus leads to exactly one response, which is the
  -- case in this instance
  public getMinMaxAverage: () ==> nat * nat * real
  getMinMaxAverage () ==
    ( dcl min : [nat] := nil, max : [nat] := nil, diff : nat := 0;
      for all cnt in set dom s2e do
        let dt = s2e(cnt) - e2s(cnt) in
          ( if min = nil then min := dt
            else (if min > dt then min := dt);
            if max = nil then max := dt
            else (if max < dt then max := dt);
            diff := diff + dt );
      return mk_(min, max, diff / card dom s2e) )

sync
  -- getNum is mutually exclusive to ensure unique values
  mutex (getNum);
  -- getMinMaxAverage is blocked until all responses have been received
  per getMinMaxAverage => card dom s2e = max_stimuli

end EnvironmentTask
```

2.2 Modelling the system application tasks

In this section, the application tasks are listed. There are six tasks in total. The event loop for each task is simple; the appropriate synchronous function is called and the next task in the sequence is signalled by sending a message to it. Note that the time penalty of the synchronous operation is modelled using the standard VICE duration statement. First the `MMIHandleKeyPressOne` class which is used in scenario 1.

```
class MMIHandleKeyPressOne is subclass of BasicTask

operations
  public MMIHandleKeyPressOne: EventDispatcher ==> MMIHandleKeyPressOne
  MMIHandleKeyPressOne (pde) ==  BasicTask("HandleKeyPress",pde);

  -- we do not specify *what* the operation does
  -- we only specify its execution time
  public HandleKeyPress: () ==> ()
  HandleKeyPress () == duration (100) skip;

  handleEvent: Event ==> ()
  handleEvent (pe) ==
    ( HandleKeyPress();
      -- send message to next task in this scenario
      sendMessage("AdjustVolume", pe.getEvent()) )

end MMIHandleKeyPressOne
```

Next, the `MMIHandleKeyPressTwo` class which is used in scenario 2.

```
class MMIHandleKeyPressTwo is subclass of BasicTask

operations
  public MMIHandleKeyPressTwo: EventDispatcher ==> MMIHandleKeyPressTwo
  MMIHandleKeyPressTwo (pde) == BasicTask("HandleKeyPress",pde);

  -- we do not specify *what* the operation does
  -- we only specify its execution time
  public HandleKeyPress: () ==> ()
  HandleKeyPress () == duration (100) skip;

  handleEvent: Event ==> ()
  handleEvent (pe) ==
    ( HandleKeyPress();
        -- send message to next task in this scenario
      sendMessage("DatabaseLookup", pe.getEvent()) )

end MMIHandleKeyPressTwo
```

The class `MMIUpdateScreenVolume` is used in scenario 1 to handle screen updates caused by the volume changes of the radio.

```
class MMIUpdateScreenVolume is subclass of BasicTask

operations
  public MMIUpdateScreenVolume: EventDispatcher ==> MMIUpdateScreenVolume
  MMIUpdateScreenVolume (pde) == BasicTask("UpdateScreenVolume",pde);

  -- we do not specify *what* the operation does
  -- we only specify its execution time
  public UpdateScreen: () ==> ()
  UpdateScreen () == duration (500) skip;

  handleEvent: Event ==> ()
  handleEvent (pe) ==
    ( UpdateScreen();
        -- scenario finished. signal response back to the environment
      raiseInterrupt("VolumeKnob", pe.getEvent()) )

end MMIUpdateScreenVolume
```

The class `MMIUpdateScreenAddress` is used in scenario 2 to handle screen updates caused by the address lookup actions in the database.

```
class MMIUpdateScreenAddress is subclass of BasicTask

operations
  public MMIUpdateScreenAddress: EventDispatcher ==> MMIUpdateScreenAddress
  MMIUpdateScreenAddress (pde) == BasicTask("UpdateScreenAddress",pde);

  public UpdateScreen: () ==> ()
  UpdateScreen () == duration (500) skip;

  -- we do not specify *what* the operation does
  -- we only specify its execution time
  handleEvent: Event ==> ()
  handleEvent (pe) ==
    ( UpdateScreen();
        -- scenario finished. signal response back to the environment
      raiseInterrupt("InsertAddress", pe.getEvent()) )

end MMIUpdateScreenAddress
```

The class `MMIUpdateScreenTMC` is used in both scenarios to display decoded TMC messages on the screen.

```
class MMIUpdateScreenTMC is subclass of BasicTask

operations
  public MMIUpdateScreenTMC: EventDispatcher ==> MMIUpdateScreenTMC
  MMIUpdateScreenTMC (pde) == BasicTask("UpdateScreenTMC",pde);

  -- we do not specify *what* the operation does
  -- we only specify its execution time
  public UpdateScreen: () ==> ()
  UpdateScreen () == duration (500) skip;

  handleEvent: Event ==> ()
  handleEvent (pe) ==
    ( UpdateScreen();
        -- scenario finished. signal response back to the environment
      raiseInterrupt("TransmitTMC", pe.getEvent()) )

end MMIUpdateScreenTMC
```

The class `RadioAdjustVolume` is used to adjust the volume of the radio.

```
class RadioAdjustVolume is subclass of BasicTask

operations
  public RadioAdjustVolume: EventDispatcher ==> RadioAdjustVolume
  RadioAdjustVolume (pde) == BasicTask("AdjustVolume",pde);

  -- we do not specify *what* the operation does
  -- we only specify its execution time
  public AdjustVolume: () ==> ()
  AdjustVolume () == duration (100) skip;

  handleEvent: Event ==> ()
  handleEvent (pe) ==
    ( AdjustVolume();
      -- send message to next task in this scenario
      sendMessage("UpdateScreenVolume", pe.getEvent()) )

end RadioAdjustVolume
```

The class `RadioHandleTMC` is used to process incoming TMC messages over the radio.

```
class RadioHandleTMC is subclass of BasicTask

operations
  public RadioHandleTMC: EventDispatcher ==> RadioHandleTMC
  RadioHandleTMC (pde) == BasicTask("HandleTMC",pde);

  -- we do not specify *what* the operation does
  -- we only specify its execution time
  public HandleTMC: () ==> ()
  HandleTMC () == duration (1000) skip;
  handleEvent: Event ==> ()
  handleEvent (pe) ==
    ( HandleTMC();
      -- send message to the next task in this scenario
      sendMessage("DecodeTMC", pe.getEvent()) )

end RadioHandleTMC
```

The class `NavigationDatabaseLookup` is used to search for an address in the navigation database.

```
class NavigationDatabaseLookup is subclass of BasicTask

operations
  public NavigationDatabaseLookup: EventDispatcher ==> NavigationDatabaseLookup
  NavigationDatabaseLookup (pde) == BasicTask("DatabaseLookup",pde);

  -- we do not specify *what* the operation does
  -- we only specify its execution time
  public DatabaseLookup: () ==> ()
  DatabaseLookup() == duration (5000) skip;

  handleEvent: Event ==> ()
  handleEvent (pe) ==
    ( DatabaseLookup();
        -- send message to next task in this scenario
      sendMessage("UpdateScreenAddress", pe.getEvent()) )

end NavigationDatabaseLookup
```

The class `NavigationDecodeTMC` is used to decode TMC messages into a human readable format using the information from the database.

```
class NavigationDecodeTMC is subclass of BasicTask

operations
  public NavigationDecodeTMC: EventDispatcher ==> NavigationDecodeTMC
  NavigationDecodeTMC (pde) == BasicTask("DecodeTMC",pde);

  -- we do not specify *what* the operation does
  -- we only specify its execution time
  public DecodeTMC: () ==> ()
  DecodeTMC () == duration (500) skip;

  handleEvent: Event ==> ()
  handleEvent (pe) ==
    ( DecodeTMC();
      -- send message to next task in this scenario
      sendMessage("UpdateScreenTMC", pe.getEvent()) )

end NavigationDecodeTMC
```

2.3 Modelling the environment tasks

There are three environment tasks for our case study. One to insert “key press” events (`VolumeKnob`), one to insert “lookup address” events (`InsertAddress`) and one to insert Traffic Message Channel messages (`TransmitTMC`). The environment inserts these events into the system by raising an interrupt. Note that all environment tasks exhibit the same basic structure. The operation `createSignal` is called periodically by its own the thread of control to generate the stimulus. Note that we *have to* use the statement `duration (0)` explicitly in order to ensure that the execution of the environment task does not influence the notion of time of the application tasks in the system model. As a consequence, we cannot write environment tasks that make use of the duration statement, for example to insert a second stimulus after x time units, because that would influence the notion of time of the system model (which is supposed to run in parallel with its “own” notion of time). This makes life for the specifier rather difficult in some cases. The underlying problem is that the uni-processor multitasking semantics of VICE is *not strong enough* to model both the environment and the system operating at the same time. We need to move from the interleaving semantics to true parallel behaviour to describe this properly, effectively implementing a multi-processor multitasking approach where the environment task is assigned its own independent processor.

Despite this deficiency in VICE, we are still able to express timeliness properties, as can be seen in the `handleEvent` operation. Whenever a response is observed, it is added to the trace information and the post-condition of the `handleEvent` operation states that the time difference between each pair of stimuli and responses shall be less than some (user-defined) value. Note that there is no completeness requirement specified here, we can in fact insert more stimuli than we receive back from the system. This is left out on purpose here; it cannot be checked on-line because a stimulus might be “inside” the system for a very long time and we are never sure that we waited long enough (this is called the quiescence property), unless we specify an explicit time-out value for each expected response. In this specification it is solved by limiting the number of inserted stimuli and waiting for all responses of all inserted stimuli. Since the number of stimuli is limited, we know that eventually the system has enough time to deal with each one. Typically, these kinds of completeness requirements are processed off-line, after the simulation run is completed; similarly for temporal properties enforcing some fixed (partial) order in the stimuli. Another reason for off-line processing might be simulation efficiency, since the log will increase in size the longer we simulate and therefore checking the post-condition every time might take up too much time. When doing post-processing the timeliness, completeness and temporal requirements only have to be checked once for the trace which makes it very efficient.

Furthermore, we have no explicit control over the start time of the first period of a periodic task, it is currently dependent on the settings of the interpreter. This can cause problems because we might know for example that two otherwise independent environment tasks have the same period but always run out of phase by some time margin, for example 10 percent of the period. If the simulator however chooses to activate both periodic tasks right after one another (remember we use duration zero) that might cause a different response time behaviour of the system than when the out-of-phase execution of the environment task was taken into account properly. At the moment, these kind of problems cannot be handled by the VICE extensions.

Another problem is the scheduling of the periodic tasks. Whenever the periodic task is scheduled, it should become the active task immediately, pre-empting any other task running, in order to maintain a consistent clock value. If the periodic task is delayed (or interrupted) by another task that lets time progress (with some duration greater than zero) then the clock will become out of sync because we do not know by how much time we were delayed. If that problem occurs during simulation of the model then the trace files that are built up here immediately become useless. But more importantly, it is currently extremely difficult to find out whether or not that problem actually did occur. So if an invalid trace is detected, is it really an invalid trace or is it a side-effect of the simulation?

The `VolumeKnob` class which is used in scenario 1.

```
class VolumeKnob is subclass of EnvironmentTask

operations
  public VolumeKnob: EventDispatcher * nat ==> VolumeKnob
  VolumeKnob (ped, pno) == EnvironmentTask("VolumeKnob", ped, pno);

  -- handleEvent receives the responses from the system
  -- and checks whether the response time for the matching
  -- stimulus was less than or equal to 1500 time units
  handleEvent: Event ==> ()
  handleEvent (pev) == duration (0) logSysToEnv(pev.getEvent())
  post checkResponseTimes(e2s,s2e,1500);

  -- createSignal generates the stimuli for the system
  createSignal: () ==> ()
  createSignal () ==
    duration (0)
      if (card dom e2s < max_stimuli) then
        ( dcl num : nat := getNum();
          logEnvToSys(num);
          raiseInterrupt("HandleKeyPress", num) );

  -- start the thread of the task
  public Run: () ==> ()
  Run () == start(self)

thread
  periodic (1000) (createSignal)

end VolumeKnob
```

The `InsertAddress` class which is used in scenario 2.

```
class InsertAddress is subclass of EnvironmentTask

operations
  public InsertAddress: EventDispatcher * nat ==> InsertAddress
  InsertAddress (ped, pno) == EnvironmentTask("InsertAddress", ped, pno);

  -- handleEvent receives the responses from the system
  -- and checks whether the response time for the matching
  -- stimulus was less than or equal to 2000 time units
  handleEvent: Event ==> ()
  handleEvent (pev) == duration (0) logSysToEnv(pev.getEvent())
  post checkResponseTimes(e2s,s2e,2000);

  -- createSignal generates the stimuli for the system
  createSignal: () ==> ()
  createSignal () ==
    duration (0)
      if (card dom e2s < max_stimuli) then
        ( dcl num : nat := getNum();
          logEnvToSys(num);
          raiseInterrupt("HandleKeyPress", num) );

  -- start the thread of the task
  public Run: () ==> ()
  Run () == start(self)

thread
  periodic (1000) (createSignal)

end InsertAddress
```

The `TransmitTMC` class which is used in both scenarios.

```
class TransmitTMC is subclass of EnvironmentTask

operations
  public TransmitTMC: EventDispatcher * nat ==> TransmitTMC
  TransmitTMC (ped, pno) == EnvironmentTask("TransmitTMC", ped, pno);

  -- handleEvent receives the responses from the system
  -- and checks whether the response time for the matching
  -- stimulus was less than or equal to 10000 time units
  handleEvent: Event ==> ()
  handleEvent (pev) == duration (0) logSysToEnv(pev.getEvent())
  post checkResponseTimes(e2s,s2e,10000);

  -- createSignal generates the stimuli for the system
  createSignal: () ==> ()
  createSignal () ==
    duration (0)
      if (card dom e2s < max_stimuli) then
        ( dcl num : nat := getNum();
          logEnvToSys(num);
          raiseInterrupt("HandleTMC", num) );

  -- start the thread of the task
  public Run: () ==> ()
  Run () == start(self)

thread
  periodic (1000) (createSignal)

end TransmitTMC
```

2.4 Linking everything together – the event dispatcher

The EventDispatcher is the most important active object in the specification. It mediates all the events in the model, both to and from the environment and to and from the system. Basically, it receives all events, puts them into temporary queues and greedily dispatches them to the appropriate receivers. When simulating, the user should ensure that it is the highest priority task running using pre-emptive scheduling, such that it can process the events as soon as they become available. The `Logger` base class is shown in the appendix. It provides functionality to output the order of the events that were processed into a file. With the new `time` keyword it is now possible to relate this user defined log to the standard log file that is produced by VICE. Using these two log files in combination can increase the insight into the model tremendously!

```
class EventDispatcher is subclass of Logger

instance variables
  queues : map seq of char to AbstractTask := {|->};
  messages : seq of AbstractTaskEvent := [];
  interrupts: seq of AbstractTaskEvent := []

operations
  -- Register is used to maintain a callback link to all the tasks in the system and
  -- the environment. the link is used by the SendNetwork and SendInterrupt operations
  -- and the event loop of the EventDispatcher (see thread)
  public Register: AbstractTask ==> ()
  Register (pat) ==
    queues := queues munion { pat.getName() |-> pat }
    pre pat.getName() not in set dom queues;

  -- setEvent is used to maintain temporary queues for the event loop of
  -- EventDispatcher. it is called by the SendNetwork and SendInterrupt operations
  -- which are in turn called from the other tasks in the system and the environment.
  setEvent: AbstractTask * Event ==> ()
  setEvent (pat, pe) ==
    if isofclass(NetworkEvent,pe)
    then messages := messages ^ [new AbstractTaskEvent(pat,pe)]
    else interrupts := interrupts ^ [new AbstractTaskEvent(pat,pe)];

  -- getEvent is used to retrieve events from the temporary event queues if they are
  -- available. otherwise getEvent is blocked (see sync) which will also block the
  -- event handler of EventDispatcher
  getEvent: () ==> AbstractTask * Event
  getEvent () ==
    if len interrupts > 0
    then ( dcl res : AbstractTaskEvent := hd interrupts;
           interrupts := tl interrupts;
           return res.getFields() )
    else ( dcl res : AbstractTaskEvent := hd messages;
           messages := tl messages;
           return res.getFields() );
  -- SendNetwork is typically called by a system or an environment
  -- task. The event is logged for post analysis and it is added
  -- to the temporary event queue for handling by the event loop
  public SendNetwork: seq of char * seq of char * nat ==> ()
  SendNetwork (psrc, pdest, pid) ==
    duration (0)
      ( dcl pbt: AbstractTask := queues(pdest);
        printNetworkEvent(psrc, pdest, pid);
        setEvent(pbt, new NetworkEvent(pid)) )
    pre pdest in set dom queues;

  -- SendInterrupt is typically called by a system or an environment
  -- task. The event is logged for post analysis and it is added
  -- to the temporary event queue for handling by the event loop
  public SendInterrupt: seq of char * seq of char * nat ==> ()
  SendInterrupt (psrc, pdest, pid) ==
    duration (0)
      ( dcl pbt: AbstractTask := queues(pdest);
        printInterruptEvent(psrc, pdest, pid);
        setEvent(pbt, new InterruptEvent(pid)) )
    pre pdest in set dom queues;

-- the event handler of EventDispatcher. note that we can simulate the overhead
-- of the operating system, which is typically also running on our system,
-- by changing the duration below. note that the while loop is blocked (and
-- this thread will be suspended) if there are no events in either queue
thread
  duration (0)
    while (true) do
      def mk_ (pat,pe) = getEvent() in
        pat.setEvent(pe)

sync
  -- setEvent and getEvent are mutually exclusive
  mutex(setEvent, getEvent);
  -- the thread shall be blocked until there is at least one message available
  per getEvent => len messages > 0 or len interrupts > 0

end EventDispatcher
```

2.5 Composing the top-level system model

The class `RadNavSys` is the top-level specification for our case-study. It is used to instantiate and start the model. The system can be analysed using two different sets of environment tasks exercising the system. The user can select between the scenarios by supplying the appropriate parameter to the constructor of the class. For example, the system can be simulated by calling `new RadNavSys(1).Run()` on the command-line of VICE.

```
class RadNavSys

types
```

```
  public perfdata = nat * nat * real

instance variables
  dispatch : EventDispatcher := new EventDispatcher();
  appTasks : set of BasicTask := {};
  envTasks : map seq of char to EnvironmentTask := {|->};
  mode : nat

operations
  -- the constructor initialises the system and starts the application
  -- tasks. because there are no events from the environment tasks yet
  -- all tasks will block in their own event handler loops. similarly
  -- the dispatcher task is started and blocked
  RadNavSys: nat ==> RadNavSys
  RadNavSys (pi) ==
    ( mode := pi;
      cases (mode) :
        1 -> ( addApplicationTask(new MMIHandleKeyPressOne(dispatch));
               addApplicationTask(new RadioAdjustVolume(dispatch));
               addApplicationTask(new MMIUpdateScreenVolume(dispatch));
               addApplicationTask(new RadioHandleTMC(dispatch));
               addApplicationTask(new NavigationDecodeTMC(dispatch));
               addApplicationTask(new MMIUpdateScreenTMC(dispatch)) ),
        2 -> ( addApplicationTask(new MMIHandleKeyPressTwo(dispatch));
               addApplicationTask(new NavigationDatabaseLookup(dispatch));
               addApplicationTask(new MMIUpdateScreenAddress(dispatch));
               addApplicationTask(new RadioHandleTMC(dispatch));
               addApplicationTask(new NavigationDecodeTMC(dispatch));
               addApplicationTask(new MMIUpdateScreenTMC(dispatch)) )
      end;
      startlist(appTasks); start(dispatch) )
   pre pi in set {1, 2};

  -- the addApplicationTask helper operation instantiates the callback
  -- link to the task and adds it to the set of application tasks
  addApplicationTask: BasicTask ==> ()
  addApplicationTask (pbt) ==
    ( appTasks := appTasks union {pbt};
      dispatch.Register(pbt) );

  -- the addEnvironmentTask helper operation instantiates the callback
  -- link to the task and and starts the environment task. since the
  -- VDMTools command-line always has the highest priority, the task will
  -- not be scheduled for execution until some blocking operation is
  -- called or the time slice is exceeded
  addEnvironmentTask: EnvironmentTask ==> ()
  addEnvironmentTask (pet) ==
    ( envTasks := envTasks  munion {pet.getName() |-> pet};
      dispatch.Register(pet);
      pet.Run() );

  -- the Run operation creates and starts the appropriate environment tasks
  -- for this scenario. to ensure that the system model has ample time to
  -- make progress (because RadNavSys will be started from the VDMTools
  -- command-line which always has the highest priority) the calls to
  -- getMinMaxAverage will block until all responses have been received
  -- by the environment task
  public Run: () ==> map seq of char to perfdata
  Run () ==
    ( cases (mode):
        1 -> ( addEnvironmentTask(new VolumeKnob(dispatch,10));
               addEnvironmentTask(new TransmitTMC(dispatch,10)) ),
        2 -> ( addEnvironmentTask(new InsertAddress(dispatch,10));
               addEnvironmentTask(new TransmitTMC(dispatch,10)) )
      end;
      return { name |-> envTasks(name).getMinMaxAverage() | name in set dom envTasks } )

end RadNavSys
```

2.6 Executing the model

Before we can execute the model, we have to set up the VICE interpreter. We use VICE in the pre-emptive scheduling mode, with all options enabled (dynamic type, invariant, pre- and post-condition checking). The following task priorities were specified (they are maintained in an external file called `priority.txt`).

```
VolumeKnob:10;
InsertAddress:10;
TransmitTMC:9;
EventDispatcher:8;
MMIHandleKeyPressOne:6;
MMIHandleKeyPressTwo:6;
MMIUpdateScreenVolume:5;
MMIUpdateScreenAddress:5;
MMIUpdateScreenTMC:4;
RadioAdjustVolume:3;
RadioHandleTMC:2;
NavigationDecodeTMC:1
```

The environment tasks are given the highest priority, to ensure that they can indeed inject stimuli whenever their period has expired. Next highest priority is the `EventDispatcher` which orchestrates the message handling between the tasks and the environment. Following are the application tasks, with priorities as specified in the case study definition. We can now execute the model using the VICE interpreter.

```
Initializing specification ...
done
>> priorityfile priority.txt
>> print new RadNavSys(1).Run()
D:/papers/vdmppsem/improve/TransmitTMC.vpp, l. 17, c. 26:
  Run-Time Error 59: The post-condition evaluated to false
>> print e2s, s2e
{ 1 |-> 2047,3 |-> 3935,4 |-> 4131,6 |-> 5119,8 |-> 6107,10 |-> 7095,
  12 |-> 8083,14 |-> 9071,16 |-> 10059,18 |-> 11047 }
{ 1 |-> 25741 }
```

The VICE interpreter stops in the post-condition of the `TransmitTMC` operation. We print out the stimuli (e2s) and response (s2e) mappings and indeed we see that the first stimulus sent actually stayed inside the system longer than 10000 time units. It is nice that we know that we do have a problem in our system but can we find the cause of the problem easily? Actually, that is very hard – even with the `time` extension recently added to VICE. We have to go over the trace files manually, or use dedicated analysis tools. Even for this simple example the trace file becomes approximately 300kb, containing several thousands lines of data. It is clear that powerfull analysis tools are needed to help the user to better understand the (execution data of the) model.

3 The case study revisited – with suggested improvements

In the following specification, we have remodelled the case study, but now using some new syntactic elements that we believe would be beneficial for specifying distributed embedded real-time systems. First of all, we introduce the notion of a "system" as a place-holder for all parts of the model that are "inside" the system.

Furthermore, we introduce a number of standard classes that can be used to reason about deployment of functionality and inter processor communication over a heterogeneous distributed environment. Most notable are the standard classes `CPU` and `BUS`. `CPU` represents the computation unit that uses a particular scheduling policy and has a certain capacity in terms of number of cycles it has available per unit of time. Instances of regular VDM++ classes can be deployed onto such a CPU which implies that their behaviour is dependent on the scheduling regime of that processor. Furthermore, the "performance" of the operations in the regular VDM++ classes are made implicitly dependent of the CPU performance. So VDM++ classes deployed on one processor might exhibit different timing behaviour then when deployed on another processor.

Similarly, the `BUS` class can be used to interconnect CPUs, which means that when an operation from a regular VDM++ class A deployed on CPU B wants to call an operation of class C which is deployed on CPU D then that operation call will cause communication over the bus that connects CPU B and D. This data communication also impacts the system performance because the BUS has limited capacity. Thus, calling operations locally (on the same CPU) might differ a lot from calling operations that are deployed on another processor. Note that the communication means (the `BUS` here) are not mentioned explicitly in the standard VDM++ classes. Thus, this is taken care of implicitly by the VDM++ interpreter such that the specifier does not need to bother with changing the functional part of a description if an alternative hardware architecture is to be investigated. The only thing the user specifies in the "system" is the deployment of the functionality on computation units (CPUs) and the topology of the communication units (BUSses). It could also be possible to visualise this using UML deployment diagrams and such connections could also be made to VDMTools.

3.1 Specifying the system

```
system RadNavSys

instance variables
  -- create an MMI class instance
  static public mmi : MMI := new MMI();
  -- define the first CPU with fixed priority scheduling and 22E6 MIPS
  CPU1 : CPU := new CPU (<FP>, 22E6);

  -- create an Radio class instance
  static public radio : Radio := new Radio();
  -- define the second CPU with fixed priority scheduling and 11E6 MIPS
  CPU2 : CPU := new CPU (<FP>, 11E6);

  -- create an Navigation class instance
  static public navigation : Navigation := new Navigation();
  -- define the third CPU with fixed priority scheduling and 113 MIPS
  CPU3 : CPU := new CPU (<FP>, 113E6);

  -- create a communication bus that links the three CPU's together
  BUS1 : BUS := new BUS (<CSMACD>, 72E3, CPU1, CPU2, CPU3)

operations
  public RadNavSys: () ==> RadNavSys
  RadNavSys ()
    ( -- deploy mmi on CPU1
      CPU1.deploy(mmi);
      CPU1.setPriority("HandleKeyPress",100);
      CPU1.setPriority("UpdateScreen",90);
      -- deploy radio on CPU2
      CPU2.deploy(radio);
      CPU2.setPriority("AdjustVolume",100);
      CPU2.setPriority("DecodeTMC",90);
      -- deploy navigation on CPU3
      CPU3.deploy(navigation);
      CPU3.setPriority("DatabaseLookup", 100);
      CPU3.setPriority("DecodeTMC", 90)
      -- starting the CPUs and BUS is implicit )

end RadNavSys
```

3.2 Specifying the application tasks

The application tasks becomes really simple and concise, see for example the class `MMI` below. Note that we have introduced the keyword `async` in the operations clause to denote that the caller of the operation is not blocked when the call is made. Also note that the class does not use the thread clause any more; asynchronous calls always require their own thread of control and this thread of control is provided by the CPU on which this application task is deployed. The specifier can mix normal (synchronous) and asynchronous operations. Note that the `HandleKeyPress` operation uses the keyword `cycles` instead of `duration`. The keyword `cycles` should be interpreted as a time penalty that is *relative* to the CPU processor speed, while `duration` remains an *absolute* time penalty. Let 痴 assume that an instance of the `MMI` class is deployed on a processor that can deliver 100E6 cycles then `skip` takes 1000 time units in `HandleKeyPress`, while the `UpdateScreen` operation *always* takes 5E5 time units, independent from the CPU on which it is deployed.

```
class MMI

operations
  async public HandleKeyPress: nat * nat ==> ()
  HandleKeyPress (pn, pno) ==
    ( cycles (1E5) skip;
      cases (pn):
        1 -> RadNavSys`radio.AdjustVolume(pno),
        2 -> RadNavSys`navigation.DatabaseLookup(pno)
      end );

  async public UpdateScreen: nat * nat ==> ()
  UpdateScreen (pn, pno) ==
    ( duration (5E5) skip;
      cases (pn):
        1 -> VolumeKnob`HandleEvent(pno),
        2 -> InsertAddress`HandleEvent(pno),
        3 -> TransmitTmc`HandleEvent(pno)
      end )

end MMI
```

Note that explicit event handling is completely removed from the specification. It is replaced by *asynchronous* operation calls. Similarly, the `Radio` class looks as follows:

```
class Radio

operations
  async public AdjustVolume: nat ==> ()
  AdjustVolume (pno) ==
    ( cycles (1E5) skip;
      RadNavSys`mmi.UpdateScreen(1, pno) );

  async public HandleTMC: nat ==> ()
  HandleTMC (pno) ==
    ( cycles (1E6) skip;
      RadNavSys`navigation.DecodeTMC(pno) )

end Radio
```

And the last system class, `Navigation`.

```
class Navigation

operations
  async public DatabaseLookup: nat ==> ()
  DatabaseLookup (pno) ==
    ( cycles (5E6) skip;
      RadNavSys`mmi.UpdateScreen(2, pno) )

  async public DecodeTMC: nat ==> ()
  DecodeTMC (pno) ==
    ( cycles (5E6) skip;
      RadNavSys`mmi.UpdateScreen(3, pno) )

end Navigation
```

3.3 Specifying the environment tasks

The environment tasks from the previous model can also be made more concise. Note that here we still use the original VDM++ thread clause definitions, using the `periodic` keyword. When the thread clause is used for a class that is not explicitly deployed on a processor, then the interpreter will automatically assume that it is a separate task which is assigned its own "virtual" processor that runs in parallel to the system. The body of the periodic thread is now simplified to an asynchronous call of an operation inside the system. The periodic keyword has four parameters instead of one. The first parameter specifies the period, the second parameter specifies the number of time units that should pass from the moment the task is started until the start of the first period (the offset). The third and fourth parameters are used to specify jitter. Jitter is the amount of variance that is allowed on the period, the fourth parameter is used to specify the minimum amount of time in between two periodic events – this is important if the allowed jitter is big compared to the period, for example to describe so-called "burst behaviours". With these four parameters it is possible to describe any kind of (semi-)periodic process. In the example below, we have used purely periodic environment tasks only, except the `TransmitTMC` task, which has an offset in the start-up of the first period. Note that the use of the `duration (0)` statements is not longer required. In fact they now *can* be used in the environment because their evaluation does not longer affect the notion of time in the system model! Finally note that the new definition of `EnvironmentTask` is also simplified.

```
class EnvironmentTask

instance variables
  -- use a unique identifier for each generated event
  static private num : nat := 0;

  -- we limit the number of inserted stimuli
  protected max_stimuli : nat := 0;

  -- administration for the event traces
  -- e2s is used for all out-going stimuli (environment to system)
  -- s2e is used for all received responses (system to environment)
  protected e2s : map nat to nat := {|->};
  protected s2e : map nat to nat := {|->}

functions
  -- checkResponseTimes verifies for each received response whether
  -- or not the elapse time did (not) exceed the user-defined limit
  public checkResponseTimes: map nat to nat * map nat to nat * nat -> bool
  checkResponseTimes (pe2s, ps2e, plim) ==
    forall idx in set dom ps2e &
      ps2e(idx) - pe2s(idx) <= plim
  -- the responses received should also be sent
  pre dom ps2e inter dom pe2s = dom ps2e

operations
  public EnvironmentTask: nat ==> EnvironmentTask
  EnvironmentTask (pno) == max_stimuli := pno;

  public getNum: () ==> nat
  getNum () == ( dcl res : nat := num; num := num + 1; return res );

  -- Run shall be overloaded to implement the event generation loop
  -- towards the system. typically, it starts a periodic thread
  public Run: () ==> ()
  Run () == is subclass responsibility;

  -- logEnvToSys is used to register when an event was inserted into
  -- the system. note that the 'time' keyword refers to the internal
  -- simulation wall clock of VDMTools
  public logEnvToSys: nat ==> ()
  logEnvToSys (pev) == e2s := e2s munion {pev |-> time};

  -- logSysToEnv is used to register when an event was received from
  -- the system. note that the 'time' keyword refers to the internal
  -- simulation wall clock of VDMTools
  public logSysToEnv: nat ==> ()
  logSysToEnv (pev) == s2e := s2e munion {pev |-> time};

  -- getMinMaxAverage calculates the minimum, maximum and average
  -- response times that were observed during execution of the model
  -- note that getMinMaxAverage is blocked until the number of
  -- system responses is equal to the number of sent stimuli
  -- termination is ensured because only a maximum number of stimuli
  -- is allowed to be inserted in the system, so eventually all
  -- stimuli can be processed by the system. this method only works
  -- when each stimulus leads to exactly one response, which is the
  -- case in this instance
  public getMinMaxAverage: () ==> nat * nat * real
  getMinMaxAverage () ==
    ( dcl min : [nat] := nil, max : [nat] := nil, diff : nat := 0;
      for all cnt in set dom s2e do
        let dt = s2e(cnt) - e2s(cnt) in
          ( if min = nil then min := dt
            else (if min > dt then min := dt);
            if max = nil then max := dt
            else (if max < dt then max := dt);
            diff := diff + dt );
      return mk_(min, max, diff / card dom s2e) )

sync
  -- getNum is mutually exclusive to ensure unique values
  mutex (getNum);
  -- getMinMaxAverage is blocked until all responses have been received
  per getMinMaxAverage => card dom s2e = max_stimuli

end EnvironmentTask
```

```
class VolumeKnob
  is subclass of EnvironmentTask

operations
```

```
  static public handleEvent: nat ==> ()
  handleEvent (pev) == logSysToEnv(pev)
  post checkResponseTimes(e2s,s2e,1000);

  public Run: () ==> ()
  Run () == start(self)

  createSignal: () ==> ()
  createSignal () ==
    ( dcl num : nat := getNum();
      logEnvToSys(num);
      RadNavSys`mmi.HandleKeyPress(1,num) )

thread
  periodic (1000,0,0,0) (createSignal)

end VolumeKnob
```

```
class InsertAddress
  is subclass of EnvironmentTask

operations
  static public handleEvent: nat ==> ()
  handleEvent (pev) == logSysToEnv(pev)
  post checkResponseTimes(e2s,s2e,2000);

  public Run: () ==> ()
  Run () == start(self);

  createSignal: () ==> ()
  createSignal () ==
    ( dcl num : nat := getNum();
      logEnvToSys(num);
      RadNavSys`mmi.HandleKeyPress(2,num) )

thread
  periodic (1000,0,0,0) (createSignal)

end InsertAddress
```

```
class TransmitTMC

operations
  static public handleEvent: nat ==> ()
  handleEvent (pev) == logSysToEnv(pev)
  post checkResponseTimes(e2s,s2e,10000);

  public Run: () ==> ()
  Run () == start(self)
  createSignal: () ==> ()
  createSignal () ==
    ( dcl num : nat := getNum();
      logEnvToSys(num);
      RadNavSys`radio.HandleTMC(num) )

thread
  periodic (1000,500,0,0) (createSignal)

end TransmitTMC
```

3.4 Executing the model

The top-level specification is now also dramatically simplified; basically there are two operations, one for each scenario. In each operation, first the system instance is created and started. Then the environment tasks are created and started and we wait until the simulation is ready.

```
class World

types
  public perfdata = nat * nat * real

instance variables
  envTasks : map seq of char to EnvironmentTask := {|->};

operations
  addEnvironmentTask: seq of char * EnvironmentTask ==> ()
  addEnvironmentTask (pnm, penv) ==
    ( envTask := envTask munion { pnm |-> penv };
      penv.Run() );

  public RunScenario1 : () ==> map seq of char to perfdata
  RunScenario1 () ==
    ( start(new RadNavSys());
      addEnvironmentTask("VolumeKnob", new VolumeKnob(10));
      addEnvironmentTask("TransmitTMC", new TransmitTMC(10));
      return { name |-> envTasks(name).getMinMaxAverage() | name in set dom envTasks } );

  public RunScenario2 : () ==> map seq of char to perfdata
  RunScenario2 () ==
    ( start(new RadNavSys());
      addEnvironmentTask("InsertAddress", new InsertAddress(10));
      addEnvironmentTask("TransmitTMC", new TransmitTMC(10));
      return { name |-> envTasks(name).getMinMaxAverage() | name in set dom envTasks } );

end World
```

Scenario 1 is run by executing the command `print new World().RunScenario1()`. A typical execution trace is shown in Figure 5. The load on the processors can be deduced from the GANTT chart. In the same fashion one can produce similar diagrams for the load on the channels (or busses) between the processors. The response times can be analysed quickly, mainly because the amount and the abstraction level of the data is in the VICE log file is significantly reduced.

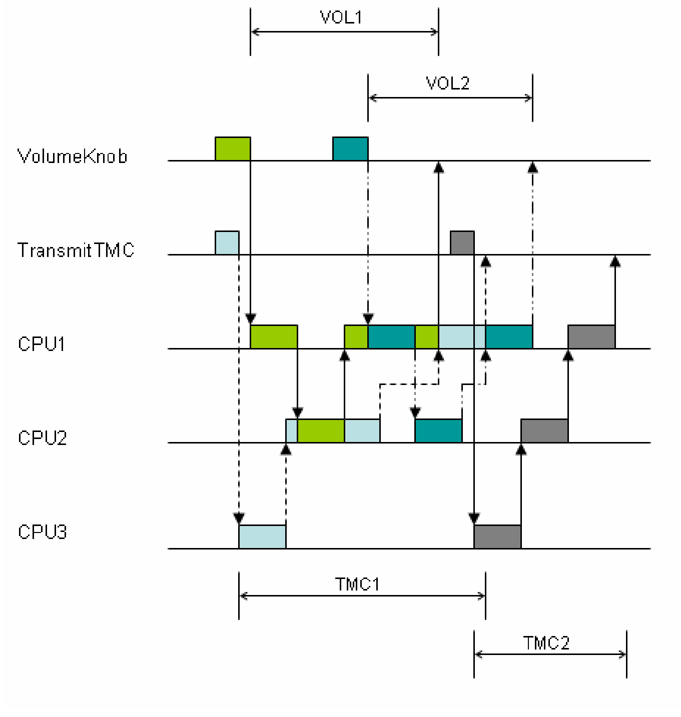


Figure 5: A GANTT chart view of the improved model execution

References

[1] Verhoef, M.: On the Use of VDM++ for Specifying Real-time Systems. In Fitzgerald, J., Larsen, P.G., Plat, N., eds.: Proceedings of the first Overture Workshop – FM'05, Claremont Tower, Newcastle Upon Tyne, NE1 7RU UK, Centre for Software Reliability, University of Newcastle Upon Tyne (2005)

[2] Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object–oriented Systems. Springer, New York (2005)

[3] Jahanian, F., Mok, A.K.L.: Safety Analysis of Timing Properties in Real-Time Systems. IEEE Transactions on Software Engineering **SE-12** (1986) 890–904

[4] Wandeler, E., Thiele, L., Verhoef, M., Lieverse, P.: System Architecture Evaluation Using Modular Performance Analysis - A Case Study. In Margaria, T., Steffen, B., Philippou, A., Reitenspiess, M., eds.: International Symposium on Leveraging Applications of Formal Methods – ISOLA 2004 – preliminary proceedings, Department of Computer Science - University of Cyprus (2004) 209–220

Appendix – Auxiliary classes

```
class Logger

instance variables
  -- using the VDMTools standard IO library to create a trace file
  static io : IO := new IO();
  static mode : <start> | <append> := <start>

operations
  -- printNetworkEvent writes a time trace to the file mytrace.txt
  -- this file can be used for application specific post analysis
  public printNetworkEvent: seq of char * seq of char * nat ==> ()
  printNetworkEvent (psrc, pdest, pid) ==
    def - = io.fwriteval[seq of (seq of char | nat)]
      ("mytrace.txt", ["network", psrc, pdest, pid, time], mode)
      in mode := <append>;

  -- printInterruptEvent writes a time trace to the file mytrace.txt
  -- this file can be used for application specific post analysis
  public printInterruptEvent: seq of char * seq of char * nat ==> ()
  printInterruptEvent (psrc, pdest, pid) ==
    def - = io.fwriteval[seq of (seq of char | nat)]
      ("mytrace.txt", ["interrupt", psrc, pdest, pid, time], mode)
      in mode := <append>;

end Logger
```

```
class AbstractTaskEvent

instance variables
  abstask : AbstractTask;
  ev : Event

operations
  public AbstractTaskEvent: AbstractTask * Event ==> AbstractTaskEvent
  AbstractTaskEvent (pat, pev) == (abstask := pat; ev := pev);

  public getFields: () ==> AbstractTask * Event
  getFields () == return mk_ (abstask, ev)

end AbstractTaskEvent
```