Improvement Suggestions for VDMTools

Peter Gorm Larsen, PGL Consult and

Marcel Verhoef, Chess Information Technology

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 and thus validate important properties in a cost-efficient fashion.

The VICE (VDM++ In a Constrained Environment) project was supported by the European Union and it was carried out shortly before Peter Gorm Larsen left IFAD in collaboration with Matra BAe Dynamics. 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's 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 written by Marcel Verhoef of Chess Information Technology (NL) that was presented at the FM'05 Overture workshop, entitled "On The Use of VDM++ for Specifying Real-Time Systems". This paper is also available on-line at

http://www.cs.ru.nl/research/reports/info/ICIS-R05033.html . Marcel is a long-time VDM enthusiast; he made the first version of the type-checker for VDMTools when he was doing his MSc at IFAD back in 1992. In the paper mentioned above, he took the existing VICE version of VDMTools 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 held at Aarhus in October 2005. We reviewed 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 mentioned above. Currently, we are targeting a scientific paper for FM'06 that describes these (language) improvements.

Basically, the improvements 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.

Each improvement category will be detailed further below. After a small description for each of these improvements, we try to make an initial suggestion about a series of milestones towards the realisation of these suggested improvements. Here the order of which suggestion to start off with is naturally essential. This is followed by a more traditional work break down into tasks but since it is at the moment still is very unclear to us how the actual work would be broken up between parties we have not tried to detail this very much at the moment. Finally, this memo is completed by an appendix with a small case study, describing a real-time embedded and distributed system. After introducing the case we first show how a VDM++ model can be produced with the existing VICE VDM++ technology. Afterwards it is shown how the VDM++ model could be formulated and validated with the suggested improvements. Without incorporating the suggested improvements we believe that it is very likely that any developments made with VDMTools for embedded systems has a substantial risks for failure.

Incorporation of ShowVICE functionality into VDMTools

Marcel Verhoef has produced a simple off-line support tool called "ShowVICE". This stand-alone tool is able to display a sequence diagram (with time annotations) of the symbolic execution of the VDM++ model in VICE. 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. This 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. Figure 1 shows an overview of the VICE trace file as it was read back into the tool from a file. 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 4 time units. This graphical feedback is essential to understand the behaviour of the application.

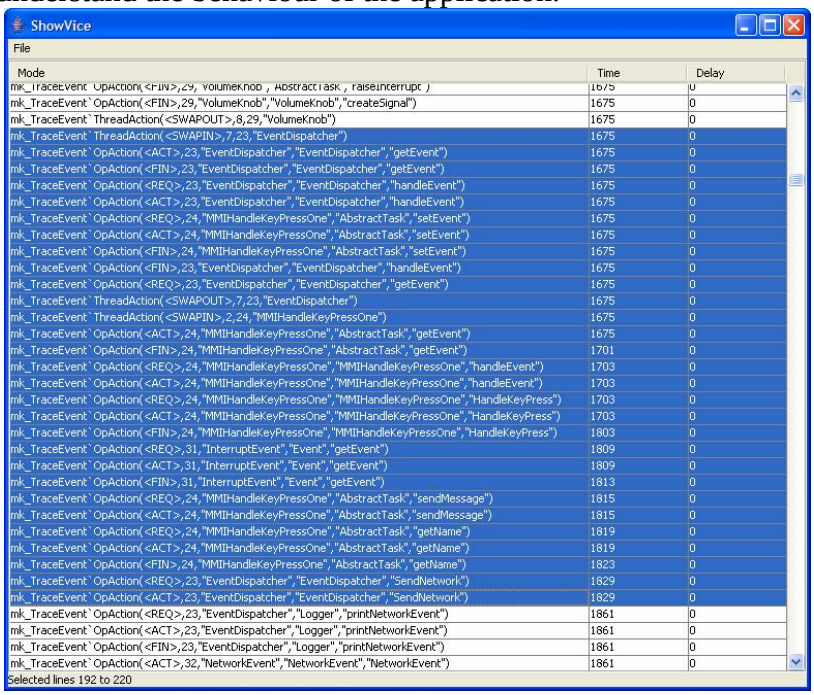

Figure 1 : the “ShowVice” user-interface, showing a parsed log file

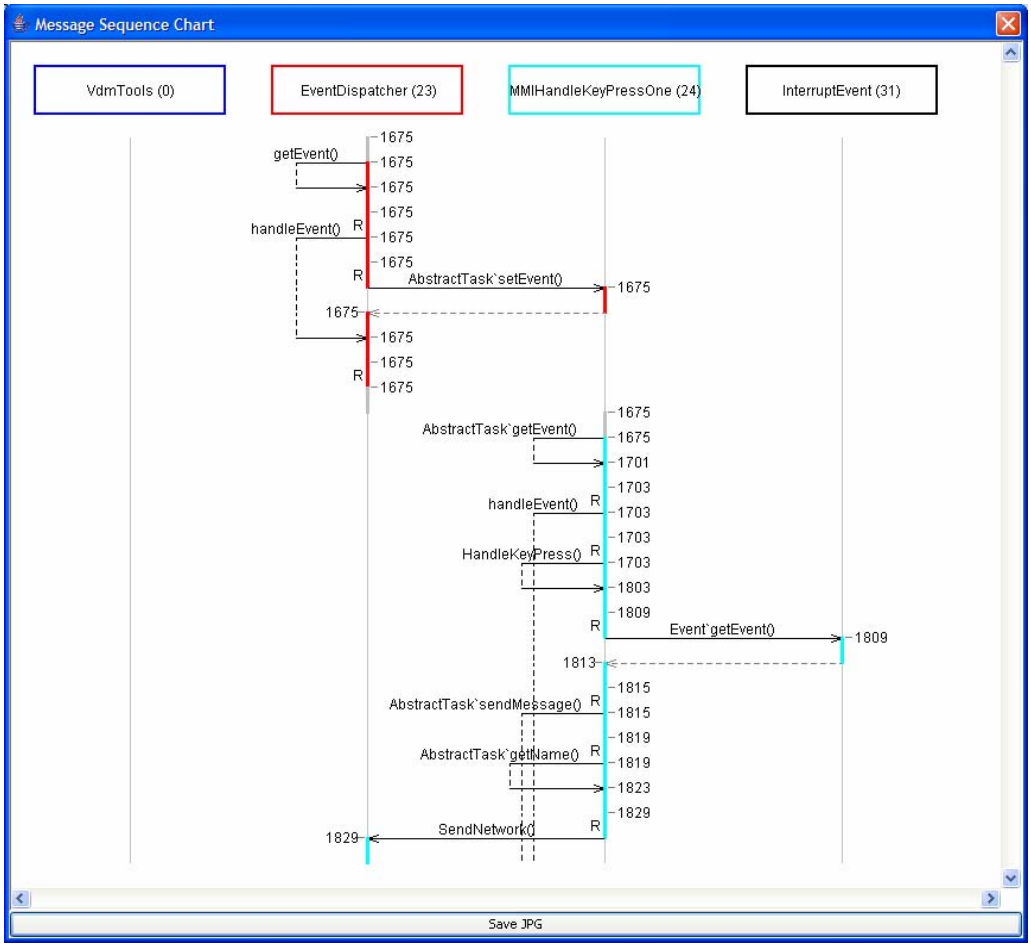

Figure 2 : showing a partial time annotated sequence diagram

When this ShowVICE 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). 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.

Increased Flexibility for Contents of Trace-files

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 tool "ShowVICE" (mentioned above) 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. This could, for example, be made 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 not be expressible without this feature. For example, the "currentTime" clock that is kept inside the interpreter is currently not available, neither at the specification level, nor in the log file.

Increased support for Validation of Deadlocks

Whenever a deadlock is reached in VDMTools, it can be very hard for a user to determine what the cause of the deadlock is. Thus, the suggestion is to enable to 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

Suggestions for improvements to the VDM++ notation

The VICE notation was recently evaluated by Marcel Verhoef in his paper from the Overture workshop at FM'05 explaining the problems when describing embedded real-time systems with VICE. The main problems he identified with the current notation are:

- The VDM++ notation is by nature a synchronous language; 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 (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 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 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 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.

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 implementation in VDMTools. The first draft of the paper is expected to be ready in January 2006. A small case study illustrating the effect of these suggested improvements is listed as an appendix for this memo. However, it is essential to emphasise that this is very much "work in progress".

Enhancing UML linkage for VDMTools

The "Rose link" was originally made to couple VDMTools with Rational Rose. Meanwhile, UML has developed substantially and is now a 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 "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.
- I-Logix have also produced a system engineering process called Harmony that seem very appropriate for the development of embedded real-time systems. This method and 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.

An Initial Incremental Milestone-based Development Plan

In case that the development of real-time embedded systems is the future focus for use of VDMTools, then we believe that it is essential to start the future development as suggested in this memo. If CSK wish to pursue all the suggestions proposed in this memo, we also believe that we are talking about several man years of effort (and a substantial time duration to fully complete the

work) to get the improvements incorporated appropriately into VDMTools as well as in its corresponding documentation. Thus, in order to provide the highest possible guarantee for "value for money" we believe that an incremental development approach will be most beneficial for CSK.

We suggest that the first milestone that should be targeted should include:

- A proof of concept of the suggested VDM++ real-time notations including:
  - Updating the VDMTools parser with the new concrete syntax elements and constructing the modified abstract syntax
  - Updating the dynamic semantics specification with the new language constructs
  - Testing the new dynamic semantics by bootstrapping with the case study from this memo plus a few additional but still relatively simple examples with experiments of different CPU architectures
- Incorporating the updated multi-processor approach in functionality like ShowVICE (either directly inside VDMTools right away or as an external tool depending upon the estimation differences).

We currently estimate that it should be possible to produce such a first proof of concept enabling a better impression of the value of the suggested capabilities with an effort in the order of 2 to 4 man-months. The rather large span in this estimate is due to the fact that it depends on a number of factors such as:

- The ambition level of how much should be included in the proof of concept (e.g. should GANTT like functionality be included in the first version);
- The level of documentation provided for the proof of concept (e.g. is the VDM-SL updates to the dynamic semantics sufficient or shall more detail descriptions be made); and
- The level of interaction required during the development of the proof of concept (e.g. is there a need for a workshop underways or at the end of the development of the proof of concept).

Since the people that needs to be involved in producing such a proof of concept have other duties as well we also believe that it will it will take at least 3 to 4 physical months to get the proof of concept completed. This is measured from the time where CSK decides to initiate the development of this proof of concept.

Before the completing of this first milestone the next milestone should be defined and estimated such that the next phase can be initiated right afterwards if the results have been promising without to much delay. A possible work break down structure is proposed in the following paragraph.

Possible Work packages

No	Work package name	Responsible	Man hour Estimate	Time duration Estimate
1	Enhance visualisation			
	• project management plan	PGL	TBD	
	• requirements specification		TBD	
	• architectural design		TBD	
	• implementation		TBD	
	• systematic test		TBD	
	• documentation update		TBD	

2	Improved static and dynamic analysis of deadlocks		
	• project management plan	PGL	TBD
	• requirements analysis		TBD
	• architectural design		TBD
	• implementation		TBD
	• systematic test		TBD
	• documentation update		TBD
3	Improved VICE notation		
	• project management plan	PGL	TBD
	• requirements analysis		TBD
	• architectural design		TBD
	• implementation		TBD
	• systematic test		TBD
	• documentation update		TBD
4	Enhanced UML support		
	• project management plan	PGL	TBD
	• requirements analysis		TBD
	• architectural design		TBD
	• implementation		TBD
	• systematic test		TBD
	• documentation update		TBD

Appendix: A small case study

The case study presented here 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 E. Wandeler et al (also available at http://www.cs.ru.nl/research/reports/info/ICIS-R05005.html). We refer to this paper for a full description of the case study. Here an attempt was made to describe the case study using VDM++, first with the original VICE extensions and later we will present the proposed extensions to the VICE language. A UML overview class diagram of the case study using the existing VICE technology is shown in Figure 3. In Figure 4 on overview class diagram of the same case study using the suggested improved VICE technology is shown.

The in-car radio navigation system is basically a system that operates several applications at the same time, for example controlling the radio while Traffic Message Channel data is being processed. Timing requirements 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.

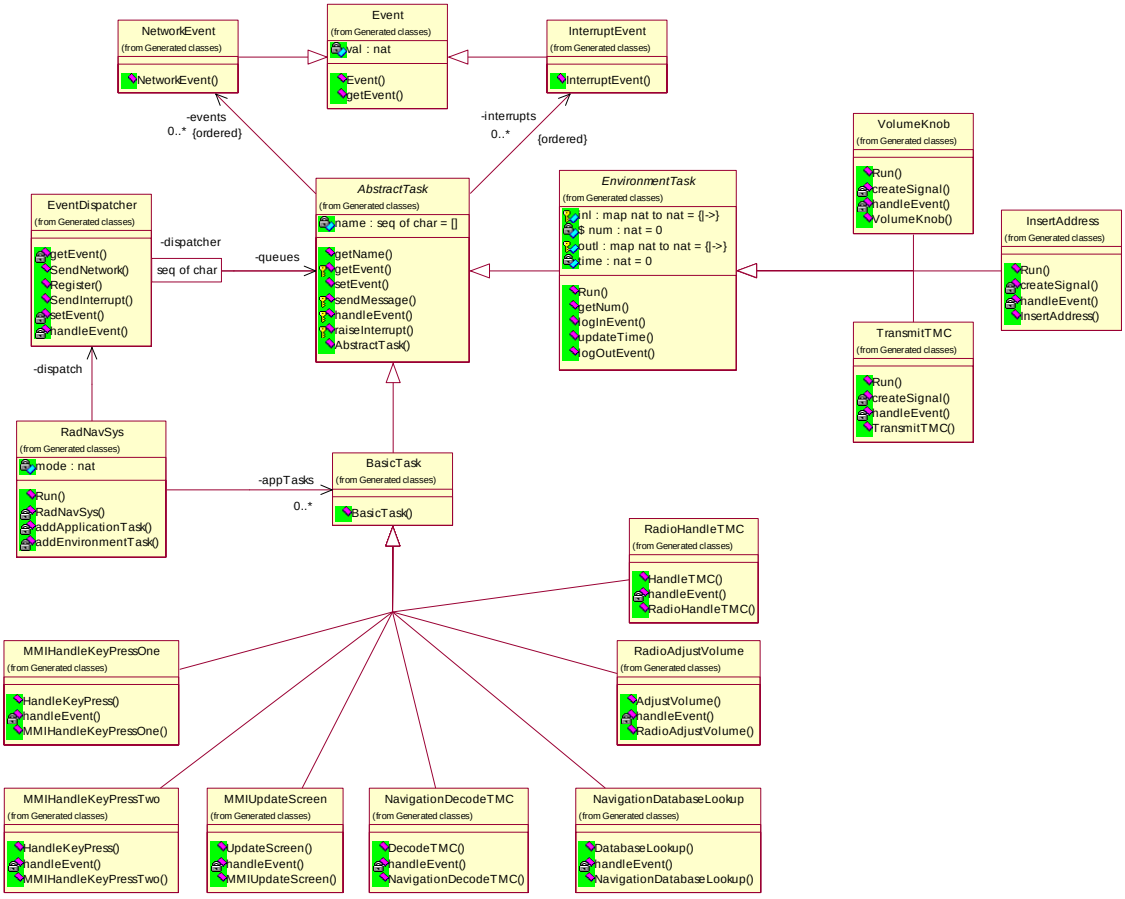

Figure 3 : UML class diagram of the case study in Appendix A

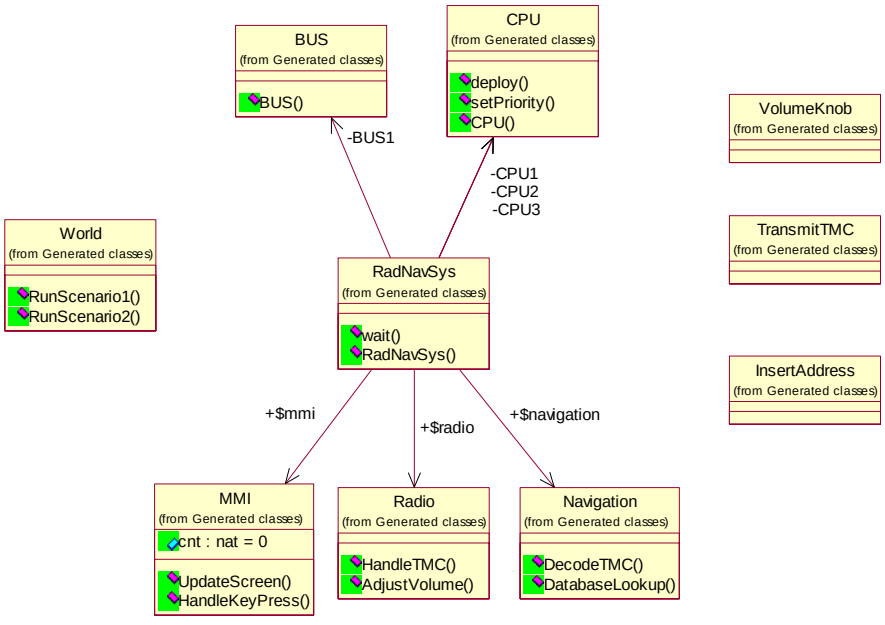

Figure 4 : UML class diagram of the improved case study in Appendix A

Modelling the example using the existing VICE VDM++ Technology

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 system state but most will cause a response back to the environment. 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. 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.

Event classes

```
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
```

Task classes

The AbstractTask class provides the basic functionality to handle events. Two 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. 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 AbstractTasks 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

  name : seq of char := [];
  events : seq of NetworkEvent := [];
  interrupts : seq of InterruptEvent := [];
  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;

  public setEvent: Event ==> ()
  setEvent (pe) ==
    if isofclass(NetworkEvent,pe)
    then events := events ^ [pe]
    else interrupts := interrupts ^ [pe];

  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 );

  protected handleEvent: Event ==> ()
  handleEvent (-) == is subclass responsibility;

  protected sendMessage: seq of char * nat ==> ()
  sendMessage (pnm, pid) ==
```

```
    dispatcher.SendNetwork(getName(), pnm, pid);

  protected raiseInterrupt: seq of char * nat ==> ()
  raiseInterrupt (pnm, pid) ==
    dispatcher.SendInterrupt(getName(), pnm, pid)

sync
  mutex (setEvent, getEvent);
  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);

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 (an OUT action) and observe the responses coming back (an IN action). Both stimuli and responses are administered using the *logOutEvent* and *logInEvent* operations respectively. We will see later how these traces are processed, for example to check the system timeliness requirements. The operation `getNum` is used to create unique identifiers, one for each stimulus that is generated and inserted into the system. Note that there in the current version of VDMTools is no way we can refer to the VDMTools internal simulation clock, therefore the notion of time needs to be administered in the specification itself using *updateTime*.

```
class EnvironmentTask
  is subclass of AbstractTask

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

  -- administration for the event traces
  time : nat := 0;

  -- stimuli send to the system, map event identifier to time
  protected outl : map nat to nat := {|->};

  -- responses received from the system, map event identifer to time
  protected inl : map nat to nat := {|->}
```

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

  public Run: () ==> ()
  Run () == is subclass responsibility;

  public updateTime: nat ==> ()
  updateTime (delta) == time := time + delta;

  public logOutEvent: nat ==> ()
  logOutEvent (pev) == outl := outl munion {pev |-> time};

  public logInEvent: nat ==> ()
  logInEvent (pev) == inl := inl munion {pev |-> time}

sync
  mutex (getNum);

end EnvironmentTask
```

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.

```
class MMIHandleKeyPressOne  is subclass of BasicTask

operations

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

  public HandleKeyPress: () ==> ()
  HandleKeyPress () ==
    duration (100) skip;

  handleEvent: Event ==> ()
  handleEvent (pe) ==
    ( HandleKeyPress();
      sendMessage("AdjustVolume", pe.getEvent()) )

end MMIHandleKeyPressOne
```

```
class MMIHandleKeyPressTwo is subclass of BasicTask

operations

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

```
  public HandleKeyPress: () ==> ()
  HandleKeyPress () ==
    duration (100) skip;

  handleEvent: Event ==> ()
  handleEvent (pe) ==
    ( HandleKeyPress();
      sendMessage("DatabaseLookup", pe.getEvent()) )

end MMIHandleKeyPressTwo
```

```
class MMIUpdateScreen  is subclass of BasicTask

operations

  public MMIUpdateScreen: EventDispatcher ==> MMIUpdateScreen
  MMIUpdateScreen (pde) ==
    BasicTask("UpdateScreen",pde);

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

  handleEvent: Event ==> ()
  handleEvent (pe) ==
    ( UpdateScreen();
      raiseInterrupt("VolumeKnob", pe.getEvent()) )

end MMIUpdateScreen
```

```
class RadioAdjustVolume  is subclass of BasicTask

operations

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

  public AdjustVolume: () ==> ()
  AdjustVolume () ==
    duration (100) skip;

  handleEvent: Event ==> ()
  handleEvent (pe) ==
    ( AdjustVolume();
      sendMessage("UpdateScreen", pe.getEvent()) )

end RadioAdjustVolume
```

```
class RadioHandleTMC  is subclass of BasicTask

operations

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

  public HandleTMC: () ==> ()
```

```
  HandleTMC () ==
    duration (1000) skip;

  handleEvent: Event ==> ()
  handleEvent (pe) ==
    ( HandleTMC();
      sendMessage("DecodeTMC", pe.getEvent()) )

end RadioHandleTMC
```

```
class NavigationDatabaseLookup  is subclass of BasicTask

operations

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

  public DatabaseLookup: () ==> ()
  DatabaseLookup() ==
    duration (5000) skip;

  handleEvent: Event ==> ()
  handleEvent (pe) ==
    ( DatabaseLookup();
      sendMessage("UpdateScreen", pe.getEvent()) )

end NavigationDatabaseLookup
```

```
class NavigationDecodeTMC  is subclass of BasicTask

operations

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

  public DecodeTMC: () ==> ()
  DecodeTMC () ==
    duration (5000) skip;

  handleEvent: Event ==> ()
  handleEvent (pe) ==
    ( DecodeTMC();
      sendMessage("UpdateScreen", pe.getEvent()) )

end NavigationDecodeTMC
```

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. 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% 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.

Last but not least, we have to keep track of time ourselves, because the simulation clock, which is maintained inside the simulator, is not available to the user. Since we use duration zero, we can use the *updateTime* operation to build our own clock by increasing the clock value with the period each time the task is activated. But it is once again relying on the scheduler of the interpreter whether or not it will be accurate. 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?

```
class VolumeKnob is subclass of EnvironmentTask
```

```
operations
  public VolumeKnob: EventDispatcher ==> VolumeKnob
  VolumeKnob (ped) == AbstractTask("VolumeKnob",ped);

  handleEvent: Event ==> ()
  handleEvent (pev) == duration (0) logInEvent(pev.getEvent())
  post forall pr in set dom inl &
         exists1 ps in set dom outl &
           pr = ps => outl(ps) - inl(pr) <= 1500;

  createSignal: () ==> ()
  createSignal () ==
    duration (0)
     ( dcl num : nat := getNum();
       updateTime(1000);
       logOutEvent(num);
       raiseInterrupt("HandleKeyPress", num) );

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

thread
  periodic (1000) (createSignal)

end VolumeKnob
```

```
class InsertAddress is subclass of EnvironmentTask

operations
  public InsertAddress: EventDispatcher ==> InsertAddress
  InsertAddress (ped) == AbstractTask("InsertAddress",ped);

  handleEvent: Event ==> ()
  handleEvent (pev) == duration (0) logInEvent(pev.getEvent())
  post forall pr in set dom inl &
         exists1 ps in set dom outl &
           pr = ps => outl(ps) - inl(pr) <= 2000;

  createSignal: () ==> ()
  createSignal () ==
    duration (0)
      ( dcl num : nat := getNum();
           updateTime(1000);
        logOutEvent(num);
        raiseInterrupt("HandleKeyPress", num) );

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

thread
  periodic (1000) (createSignal)

end InsertAddress
```

```
class TransmitTMC is subclass of EnvironmentTask

operations
  public TransmitTMC: EventDispatcher ==> TransmitTMC
```

```
  TransmitTMC (ped) == AbstractTask("TransmitTMC", ped);

  handleEvent: Event ==> ()
  handleEvent (pev) == duration (0) logInEvent(pev.getEvent())
  post forall pr in set dom inl &
         exists1 ps in set dom outl &
           pr = ps => outl(ps) - inl(pr) <= 100000;

  createSignal: () ==> ()
  createSignal () ==
    duration (0)
      ( dcl num : nat := getNum();
           updateTime(1000);
           logOutEvent(num);
        raiseInterrupt("HandleTMC", num) );

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

thread
  periodic (1000) (createSignal)

end TransmitTMC
```

Event dispatching

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, such that it can process the events as soon as they become available. The Logger base class is not shown here but it basically provides functionality to output the order of the events that were processed into a file. Note that we cannot time stamp these actions because we cannot refer to the VDMTools internal simulation clock, as mentioned before. This is a shame because otherwise we could relate this user defined log to the standard log file that is produced by VICE. It would increase the insight into the model tremendously if we could do this!

```
class EventDispatcher  is subclass of Logger

instance variables

  queues : map seq of char to AbstractTask := {|->};
  messages : seq of (AbstractTask * Event) := [];
  interrupts: seq of (AbstractTask * Event) := []

operations

  public Register: AbstractTask ==> ()
  Register (pat) ==
    queues := queues munion { pat.getName() |-> pat }
  pre pat.getName() not in set dom queues;

  setEvent: AbstractTask * Event ==> ()
  setEvent (pat, pe) ==
    if isofclass(NetworkEvent,pe)
    then messages := messages ^ [mk_(pat,pe)]
```

```
    else interrupts := interrupts ^ [mk_(pat,pe)];

  getEvent: () ==> AbstractTask * Event
  getEvent () ==
    if len interrupts > 0
    then ( dcl res : AbstractTask * Event := hd interrupts;
           interrupts := tl interrupts;
           return res )
    else ( dcl res : AbstractTask * Event := hd messages;
           messages := tl messages;
           return res );

  public SendNetwork: seq of char * seq of char * nat ==> ()
  SendNetwork (psrc, pdest, pid) ==
    duration (0)
     ( dcl pbt: BasicTask := queues(pdest);
       printNetworkEvent(psrc, pdest, pid);
       setEvent(pbt, new NetworkEvent(pid)) )
  pre pdest in set dom queues;

  public SendInterrupt: seq of char * seq of char * nat ==> ()
  SendInterrupt (psrc, pdest, pid) ==
    duration (0)
     ( dcl pbt: BasicTask := queues(pdest);
       printInterruptEvent(psrc, pdest, pid);
       setEvent(pbt, new InterruptEvent(pid)) )
  pre pdest in set dom queues;

  handleEvent: AbstractTask * Event ==> ()
  handleEvent (pat, pe) == pat.setEvent(pe)

thread
  duration (0)
    while (true) do
      def mk_ (pat,pe) = getEvent() in
        handleEvent(pat,pe)

  sync  mutex(setEvent, getEvent);
  per getEvent => len messages > 0 or len interrupts > 0

end EventDispatcher
```

The RadNav system – the top-level specification

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. For example, the system can be simulated by calling "`new RadNavSys(1).Run()`" on the command-line of VICE.

```
class RadNavSys

instance variables

  dispatch : EventDispatcher := new EventDispatcher();
  appTasks : set of BasicTask := {};  mode : nat
```

```
operations

  RadNavSys: nat ==> RadNavSys
  RadNavSys (pi) ==
   ( mode := pi;
     cases (mode) :
       1 -> ( addApplicationTask(new MMIHandleKeyPressOne(dispatch));
              addApplicationTask(new RadioAdjustVolume(dispatch));
              addApplicationTask(new MMIUpdateScreen(dispatch));
              addApplicationTask(new RadioHandleTMC(dispatch));
              addApplicationTask(new NavigationDecodeTMC(dispatch)) ),
       2 -> ( addApplicationTask(new MMIHandleKeyPressTwo(dispatch));
              addApplicationTask(new NavigationDatabaseLookup(dispatch));
              addApplicationTask(new MMIUpdateScreen(dispatch));
              addApplicationTask(new RadioHandleTMC(dispatch));
              addApplicationTask(new NavigationDecodeTMC(dispatch)) )
     end;
     startlist(appTasks);
     start(dispatch) )
    pre pi in set {1, 2};

  addApplicationTask: BasicTask ==> ()
  addApplicationTask (pbt) ==
   ( appTasks := appTasks union {pbt};
     dispatch.Register(pbt) );

  addEnvironmentTask: EnvironmentTask ==> ()
  addEnvironmentTask (pet) ==
   ( dispatch.Register(pet);
     pet.Run() );

  public Run: () ==> ()
  Run () ==
   ( cases (mode):
       1 -> ( addEnvironmentTask(new VolumeKnob(dispatch));
              addEnvironmentTask(new TransmitTMC(dispatch)) ),
       2 -> ( addEnvironmentTask(new InsertAddress(dispatch));
              addEnvironmentTask(new TransmitTMC(dispatch)) )
     end;
     Logger`wait() )

end RadNavSys
```

Modelling the same case-study using the Improved VDM++ Technology

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 together, 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.

```
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 performance
  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 performance
  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 performance
  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 )

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

  sync
    per wait => mmi.cnt > 30

end RadNavSys
```

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 as (s)he pleases. Note that the *HandleKeyPress* operation uses the keyword `cycles` instead of `duration`. Cycles should be interpreted as a time penalty that is *relative* to the CPU processor speed, duration remains an absolute time penalty. Let's assume that an instance of the MMI class is deployed on a processor that can deliver 100E6 cycles then increasing the counter 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

instance variables
  public cnt : nat := 0

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

```
      end );

  async public UpdateScreen: nat ==> ()
  UpdateScreen (-) ==
    ( duration (5E5) skip;
      VolumeKnob`HandleEvent(cnt) )

end MMI
```

```
class Radio

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

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

end Radio
```

```
class Navigation

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

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

end Navigation
```

The environment tasks from the previous model can be made more concise also. 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, than 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 now 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 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 simplified, mainly because a new operation called *now()* is available that returns the value of the simulation wall-clock in the VDMTools interpreter.

```
class EnvironmentTask

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

  -- stimuli send to the system, map event identifier to time
  protected outl : map nat to nat := {|->};

  -- responses received from the system, map event identifer to time
  protected inl : map nat to nat := {|->}

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

  public logOutEvent: nat ==> ()
  logOutEvent (pev) == outl := outl munion {pev |-> now()};

  public logInEvent: nat ==> ()
  logInEvent (pev) == inl := inl munion {pev |-> now()}

end EnvironmentTask
```

```
class VolumeKnob
  is subclass of EnvironmentTask

operations
  handleEvent: nat ==> ()
  handleEvent (pev) == logInEvent(pev)
  post forall pr in set dom inl &
         exists1 ps in set dom outl &
           pr = ps => outl(ps) - inl(pr) <= 1000

thread
  periodic (1000,0,0,0)
    ( dcl num : nat := getNum();
      logOutEvent(num);
      RadNavSys`mmi.HandleKeyPress(1) )

end VolumeKnob
```

```
class InsertAddress
  is subclass of EnvironmentTask

thread
  periodic (1000,0,0,0)
    ( dcl num : nat := getNum();
      logOutEvent(num);
      RadNavSys`mmi.HandleKeyPress(2) )

end InsertAddress
```

```
class TransmitTMC

thread
  periodic (1000,500,0,0)
    ( dcl num : nat := getNum();
      logOutEvent(num);
      RadNavSys`radio.HandleTMC() )

end TransmitTMC
```

Our 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

operations
  public RunScenario1 : () ==> ()
  RunScenario1 () ==
    ( start(new RadNavSys());
      startlist({new VolumeKnob(), new TransmitTMC()});
      RadNavSys`wait() );

  public RunScenario2 : () ==> ()
  RunScenario2 () ==
    ( start(new RadNavSys());
      startlist({new InsertAddress(), new TransmitTMC()});
      RadNavSys`wait() );

end World
```

Scenario is run by interpreting `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.

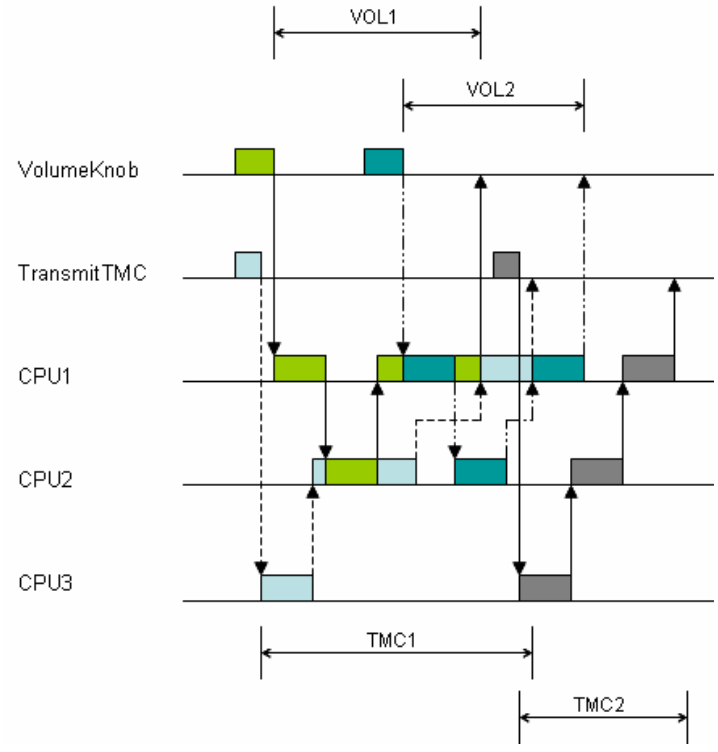

Figure 5 : a GANTT chart of the improved model execution