Run-Time Validation of Timing Constraints for VDM-RT Models

Augusto Ribeiro, Kenneth Lausdahl, and Peter Gorm Larsen

Aarhus School of Engineering, Dalgas Avenue 2, DK-8000 Aarhus C, Denmark,
{ari,kel,pgl}@iha.dk

Abstract. Development of distributed real-time embedded systems is often a challenging task and validation of the timing behaviour of such systems is typically as important as its functional correctness. VDM-RT is a modelling language with an executable subset that can be used to describe distributed real-time embedded systems. In previous work [5], post-analysis of important timing constraints was achieved by inspecting a log file that results from simulating a VDM-RT model using VDMTools. In this paper we present how validation of such timing constraints actually can be efficiently carried out during run-time using the interpreter from the open source Overture/VDM tool suite.

Keywords: VDM-RT; real-time distributed embedded systems; timing properties validation

1 Introduction

Development of distributed real-time embedded systems is often a challenging task. Typically, real-time embedded systems have timing constrains that should be respected for the system to be considered useful. These timing constrains are obvious for a hard real-time system where the failure to respond within a certain time interval can lead to total system failure but even soft real-time systems can have time constrains. For example, when a user presses the TV remote control to change channel, he expects the channel on the TV to change in an acceptable amount of time.

Using modelling tools to gain better understanding of a system is seen as a good practice [3]. By using simulation, one can gain confidence that a model is doing what it is expected. By being able to define time constrains and validate these constrains in a model during simulation, one could gain even more confidence.

VDM-RT is a modelling language that permits the specification of distributed real-time systems which has an executable subset. In this article, we present a tool enhancement for the VDM-RT interpreter [10] that extends the work presented in [5] and adds the capability of defining timing constrains to a model and validate them during interpretation.

This paper starts off with a short presentation of the relevant aspects in Section 2. Afterwards Section 3 introduces a small case study for an in-car navigation and radio system and illustrates how the existing tools can be used to provide a graphical overview of the interpretation of such an example distributed over multiple CPUs. Then Section 4 introduces the notion of system-wide timing invariants suggested by this article. This

is followed by Section 5 illustrating how such timing invariants can be used concretely in VDM-RT and how the tool support can be updated with visualisation of violation of such timing invariants. Finally Section 6 provides a few concluding remarks about the work presented in this article.

2 VDM-RT

The Vienna Development Method (VDM) [2, 8, 6] was originally developed at the IBM laboratories in Vienna in the 1970s and as such it is one of the longest established formal methods. VDM comes in three different flavours: VDM-SL [12] (VDM Specification Language) an ISO Standard; VDM++ [7] an object oriented extension of VDM-SL that supports concurrency; and more recently VDM-RT [14, 13], an extension to VDM++ to model distributed real-time embedded systems. VDM-RT is supported by Overture Tool [9] and VDMTools [4]. Both tools includes an interpreter capable of running the executable part of VDM-RT but the work described in this article is only built into Overture.

VDM-RT includes the notion of a quantifiable time; there is a system clock which is running from beginning till the end of interpretation. Currently, the maximum precession allowed in the interpreter is 1 nanosecond. It also contains the notion of processing units; the built-in `CPU` class can be used to declare processing unit and its speed (in Hz); different parts of the model are deployed to specified CPUs. CPUs can communicate between themselves through buses. VDM-RT constructs take time to be interpreted, this time is shorter or longer according to the CPU speed. Using the keywords **`cycles`** and **`duration`** it is possible to influence how much time a construct takes to execute. Using **`cycles`** one can say how many CPU cycles an instruction will take to complete; using this keyword will make the speed to complete an instruction inversely proportional to the speed of the CPU. On the other hand, the keyword **`duration`** turns the completion time of an instruction to a constant value; this can be useful to model, for example, a IO access where it takes a constant time independent of the speed of the CPU accessing it. There is also a special kind of CPU, which is present in all the VDM-RT models implicitly, the virtual CPU (vCPU) which per default is infinitely fast and its execution does not affect system timing. When a VDM-RT model is interpreted, a log is produced in which all events related with operations and function calls, object and threads creation, activation and deactivation that happened during the interpretation are registered. This log can be visualized graphically like shown in Figure 2.

A special kind of predicates called permission predicates, can act as a guard to operations and can be used to ensure synchronization of concurrent threads. Within these predicates it is possible to use operation history guards. History guards denote the number of requests, activations and completions of the operations. For each of these possible operation states, an event is generated in the log. The VDM-RT syntax to express these events is **`#req`** for request, **`#act`** for activate and **`#fin`** for finish. The *request* event indicates that the interpreter wishes to call the operation. The *activate* event indicates that the requested method was actually activated, this distinction is made because there might exist a delay between request and activation either due to a synchronization condition in the operation or because the CPU executing the thread might not have enough

processing power. The *finish* event indicates that the operation has completed. The relative timing of these events are important in case timing requirements for the system being modelled are needed.

3 Case study

In this section, we introduce a VDM-RT model and the associated existing tool support. The idea behind the model is to describe an in-car navigation radio and check if it is possible to validate its timing requirements. An overview of the system is presented in Figure 1. The environment has three types of interaction with the system, it is possible

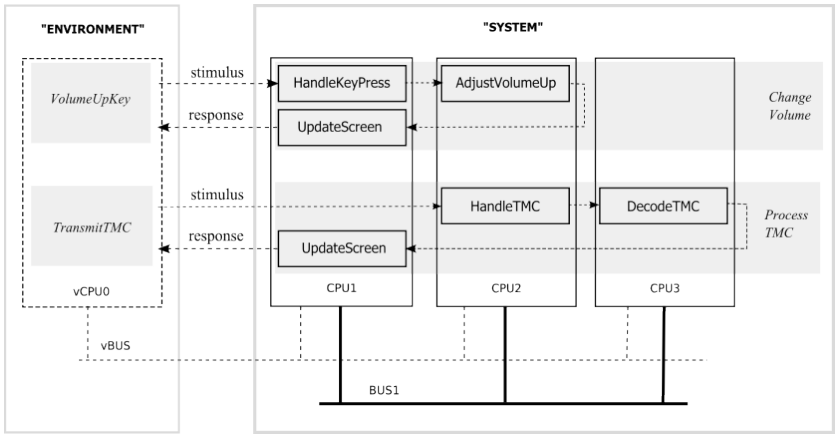

Fig. 1. Overview of the In-car Radio Navigation System

for the system to receive new TMC broadcasts (Traffic Message Channel) and adjust the volume (the volume down interaction is not presented in Figure 1 because it is similar to adjusting the volume up). The system is divided into three major components, the man-machine interface (MMI) in `CPU1`, the radio in `CPU2` and the navigation system in `CPU3`. All these CPUs are connected through a common bus (`BUS1`). Finally all the CPUs have a connection through the vBUS to the vCPU where the environment is present. Listing 1.1 shows how the Radio class is modelled in VDM-RT.

```
class Radio

values
  public MAX : nat = 10;

instance variables
  public volume : nat := 0;
```

```
operations
  async public AdjustVolumeUp : () ==> ()
  AdjustVolumeUp () ==
  ( cycles (1E6) skip;
    if volume < MAX
    then ( volume := volume + 1;
           RadNavSys`mmi.UpdateScreen(1)));

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

end Radio
```

Listing 1.1. Snippet of the Radio class

It has 3 operations (`AdjustVolumeDown` is not presented), the ones to adjust volume and one that handles the incoming TMC signal. The operations illustrate the use of the keyword cycles, in this case it means that 10^6 cycles (`1E6`) are used in the computation of the operation. The rest is on purpose kept very simple, the `AdjustVolume` operations change the volume if they did not reach the limit and notifies the screen to do an update. The `HandleTMC`, relays the decoding of the TMC signal to the navigation unit.

Currently, the tool support available is capable of producing a detailed log of the execution of a VDM-RT model. There is also a tool, the RTLogViewer that allows graphical visualization of such logs. Figure 2 shows RTLogViewer at work. The log

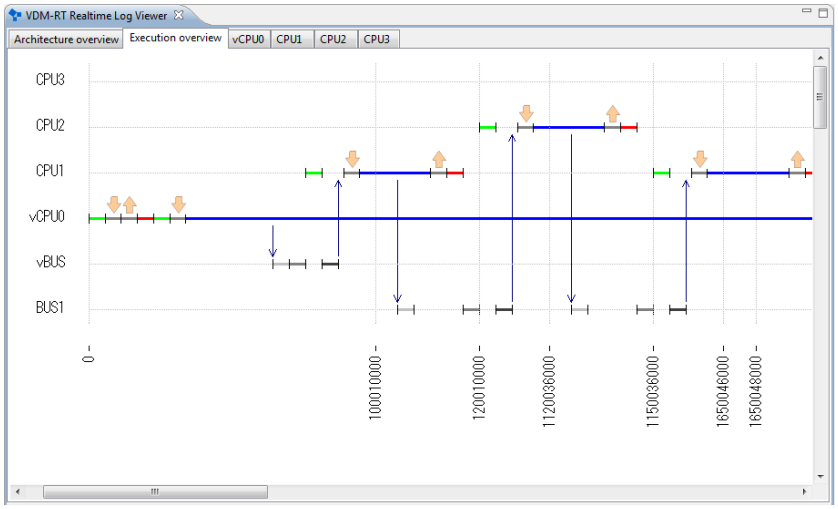

Fig. 2. Log showing one of the executions of the model

contains details such as when certain parts of the model were active and which calls were made at a certain time. Note that the time unit used on the log is nanoseconds (*ns*) as opposed to the time unit used throughout the rest of this article which is milliseconds (*ms*).

A number of system-wide timing invariants need to be added to the in-car navigation system in order to provide a good user interface experience.

C1: *A volume change must be reflected in the display within 35 ms.*
C2: *The screen should be updated no more than once every 500 ms.*
C3: *If the volume is to be adjusted upwards and it is not currently at the maximum, the audible change should occur within 100 ms.*

It can be argued that C1 and C2 are clashing since we demand the screen to update within 35ms after a key press (in C1) and that the screen only updates each 500ms (in C2) but this was chosen on purpose for testing reasons.

4 Timing Invariants

Timing invariants are logical statements that allow a modeller to formulate system-wide timing properties, these properties indicate a relation between two events. These properties have the form of a predicate over events and operate in a three value logic (true, false and unknown). Because these properties are to be verified in a VDM-RT environment we can use in their definition the notion of time. Informally, a property consists of a 6-tuple containing at least the following[1]:

A name: the property name (P);
A relation: a relation between the two events and a time interval ($\sqsubset$);
A trigger: an event that triggers the validation of a conjecture (e_t);
An ending: when the ending event happens, the conjecture can be checked for satisfiability (e_e);
A time interval: the time interval used in the property (i);
A default evaluation: the default evaluation (*true* or *false*) to be returned if the ending event never occurs (d).

We attempt to formally define a property P. To assist us in this task we need the function *time* (t) that returns an event time of occurrence.

$$t(e) = \begin{cases} \text{time} & \text{if } e \text{ occured} \\ \varnothing & \text{if } e \text{ did not occur} \end{cases}$$

Where *time* is the systems time of the occurrence of event e. If both events (trigger and ending) occurred then $t(e_e) \geq t(e_t)$. The current system time is denoted by *curr*. As expected, it is only possible to evaluate if a property holds if the trigger event e_t occurs but it might be possible to evaluate it before the ending event e_e occurs or even if it does not occur at all.

[1] We say at least because extended versions of the property will appear.

$$\mathrm{P}(\sqsubset, e_e, e_t, i, d) \equiv \begin{cases} t(e_e) - t(e_t) \sqsubset i & \text{if } t(e_e) \neq \varnothing \wedge t(e_t) \neq \varnothing \\ d & \text{if } t(e_e) = \varnothing \wedge \mathrm{curr} - t(e_t) > i \end{cases} \quad (1)$$

Where the kind of the property in question determines which relation is ($\sqsubset$) and the default evaluation (d). Because simulation is time framed, it can happen that it terminates before a property can be properly evaluated (the case where $t(e_e) = \varnothing \wedge \mathrm{curr} - t(e_t) \leq i$), when this happens the property evaluation is deemed *inconclusive*.

Now that we have the generic property formally defined, we can by specifying P, $\sqsubset$ and d in definition 1 derive at least three interesting properties.

1. **Deadline Met:** A deadline by definition is a time by which something must be finished. In real-time embedded systems there is typically deadlines that must be respected from when an event happens to its response. In our terminology, it means that the ending event must happen within a certain timeframe from the trigger event. We instantiate 2 and fixate P, $\sqsubset$ and d for the *deadlineMet* property in the following way:

   $$\mathrm{deadline}(\leq, e_1, e_2, i, \mathrm{false}) \quad (2)$$

   Just for better comprehension, the expanded version of definition 2 is presented below:

   $$\mathrm{deadline}(e_1, e_2, i) \equiv \begin{cases} t(e_e) - t(e_t) \leq i & \text{if } t(e_e) \neq \varnothing \wedge t(e_t) \neq \varnothing \\ \mathrm{false} & \text{if } t(e_e) = \varnothing \wedge \mathrm{curr} - t(e_t) > i \end{cases} \quad (3)$$

2. **Separate:** Intuitively, separation properties describe a minimum separation between events if the second event occur at all and it can be defined through specifying 1 in the following way:

   $$\mathrm{separate}(>, e_1, e_2, i, \mathrm{true}) \quad (4)$$

3. **Separate Required:** Intuitively, required separations are separations in which the second event is required to occur after the minimum separation. Again we define it by specifying 1:

   $$\mathrm{separateReq}(>, e_1, e_2, i, \mathrm{false}) \quad (5)$$

   There is only a subtle difference between definitions 4 and 5. The default evaluation ensures the desired result when evaluating the separation properties.
   A peculiar case happens when the ending event does not occur while interpreting a model. Since a model is simulated within a time range (t_n), the ending event could potentially happen some time in the future after the simulation has stopped. In this case, the property would evaluate to *inconclusive* if $t_n - t(e_t) \leq i$ or to the default evaluation (false) otherwise. This case requires attention by the modeller because it is not possible to tell if the ending event would happen in the future and change the evaluation of the property.

4.1 Events

The basic concept of properties have been described and it was mentioned that properties are predicates over events but no definition of event has been provided yet. In this section we will provide a formalization of the notion of events as used in the timing invariants. Events are defined as predicates over certain occurrences that happen in the model during the interpretation. Events can be divided into two types:

Operation events: the VDM-RT semantics defines three identifiable states of an operation: *request*, when an operation is registered to be invoked; *activation*, when an operation is really invoked (the time of *request* and *activation* can be different for several reasons); and finally *finished*, when an operation call is completed.
An *operation event* is an event tied to one of these operation states either at class or object level[2]. So basically when an event is associated with an operation state and a class, this event is registered whenever any object of this class invoking the operation enters that state. On the other hand, an event associated with an object is only registered when the specific object enters that state. Assuming that *opStateSet* is a set that contains tuples of the form $(object, op, state)$ which is populated with the operations that are in a certain state in an object for the current system time (*curr*). We formalize the object level event as:

$$\text{objOpEvent}(object, op, state) \equiv (object, op, state) \in \text{opStateSet} \tag{6}$$

The class level event can be formalized with the help of definition 6 as:

$$\begin{aligned}&\text{classOpEvent}(class, op, state) \equiv \\ &\exists(obj, op, state) \in (\text{opStateSet}).obj \in class \wedge \text{objOpEvent}(obj, op, state)\end{aligned} \tag{7}$$

Predicate events: this kind of events is associated with a predicate, the event occurs when the predicate is true. These predicates must have as argument at least one instance variable that is accessible from the system class, i.e. any variable that is accessible after initialization of the system. Assuming a predicate p with n arguments we formalize predicate events as:

$$\text{predEvent}(p, a_1, \ldots, a_n) \equiv p(a_1, \ldots, a_n) \tag{8}$$

At least one instance variable has to be used as argument because predicate events are only evaluated in case of a variable state change. The reason for this is that evaluated all predicate events at all times could be computationally expensive. By tying a predicate with a variable state change, the number of times the predicate is evaluated is possibly highly reduced.

Timing invariants contain two events, a trigger and an ending, as shown in Section 4. Each trigger and ending event can be formed by a combination of operation and predicate event. Here follows the definition of a timing invariant event (trigger or ending):

$$\text{timInvEvent}(opEv, predEv) \equiv \begin{cases} opEv & \text{if predEv is not defined} \\ predEv & \text{if opEv is not defined} \\ opEv \wedge predEv & \text{otherwise} \end{cases} \tag{9}$$

[2] For practical reasons we limit the object level to instance variables present in the system class

If both events are defined, the *opEv* takes precedence over the *predEv* since it only makes sense to calculate the later if the first one evaluates to true.

4.2 Invariant Instances

A timing invariant typically needs to be validated more than once for each simulation, for each time the trigger event occurs. These are denominated *invariant instances* because they are instances of the same invariant triggered in different situations. The lifetime of a single instance of an invariant is described below:

1. Before the trigger event occurs, the instance does not exist;
2. If at a certain point in time, the trigger event happens, an instance of the invariant is created in which the time of the trigger event is registered. We denominate these instances *active*;
3. If the ending event occurs, the time of its occurrence will be registered in all[3] the instances of the invariant. The instances are marked as ended and its evaluation can be made. We denominate these instances *decommissioned*. This decommissioning policy is called *non-selective*;
4. If an instance does not hold it remains saved for later display.

Invariant instances represent fully specified versions of the timing invariants presented in definition 1 where all the free variables have been fixed. An arbitrary number of instances of an invariant can exist at a certain point in time during simulation.

Assuming *timInv* is the set of defined timing invariants, *actInst* is the set of active instances, *decoInst* the set of decommissioned instances we can define the transition of states at a given time. Definition 10 describes how invariant instances are created from invariant definitions. The function *isTrigger* checks if the trigger event of an invariant is occurring. The function *createInst* creates an invariant instance from a definition and registers it in the current time.

$$\forall inv \in timInv.\text{isTrigger}(timInv) \implies \text{createInst}(timInv) \cup \text{actInst} \tag{10}$$

Definition 11 describes how an instance passes from active to the decommissioned state. Function *isEnding* is analogous to *isTrigger* but for the ending event.

$$\begin{aligned} \forall inv \in timInv, inst \in \text{instances}(inv, actInv).\text{isEnding}(timInv) \implies \\ \text{actInst} \setminus inst \wedge (\neg\text{isSatisfied}(inst) \implies inst \cup \text{decoInst}) \end{aligned} \tag{11}$$

The function *isSatisfied* checks if an instance of the invariant holds or not. By following this strategy, in the end of an interpretation we will end up with the invariant instances that did not hold in the set *decoInst*.

The *matching* policy The *non-selective* decommissioning policy of invariant instances might not be the proper solution for all cases. With this policy it is not possible to describe that an ending event can only decommission one instance. Assuming that e_t

[3] Further in this section another way of decommissioning the instances is described.

and e_e are trigger and ending events respectively, for a certain invariant P. Considering the following string of events:

$$e_t, e_t, e_e \tag{12}$$

With the policy described before, the following would happen: two instances of the invariant P would be created, one for each e_t then both instances would be decommissioned by the only e_e. One can think of another policy that instead of keeping a set of active instances, keeps a sequence. In this mode, we demand that the trigger and ending events happen in couples for the invariants to be decommissioned. This kind of decommissioning uses a *matching* policy. By doing this, if the string of events presented in definition 12 ocurrs, one instance of the invariant will still be active. Both policies are possible to be implemented and we decided to delegate the policy selection by extending with one more argument to the timing invariant presented in definition 1.

$$\mathrm{P}(\sqsubset, e_e, e_t, i, d, m) \tag{13}$$

The boolean argument m means *match* and decides if the decommissioning of instances is made according to the *matching* policy.

Other policies In Section 4.1, operation events over classes were discussed. Defining a class operation event can lead to the situation where an invariant is triggered by one object of that class and ended by another object of the same class. In certain situations this might not be exactly what the modeller is looking for. Hence one more possible policy of decommissioning of instances is a policy that demands that the trigger and the ending event occur on the same object. Another restriction that might appear natural is to demand that the trigger and ending event occur in the same thread. The choice of the policies is model specific or even invariant specific, it depends on what is the modeller looking for in each individual case.

One more possible extension to definition 13 is to add extra fields to enable more policies. The definition extension will not be made here since these are presented here merely for completion and discussion sake. All the mentioned policies in this paper are possible to implement and they have been present in the development phase of the prototype. The final decommissioning policy chosen for the prototype was the *matching* policy simply because it was the most appropriate fit for the example we chose.

5 Run-Time Invariant Checking

A part of what was described in Section 4 was implemented as a prototype as part of this work. The prototype has been built on top of the open-source VDM-RT interpreter VDMJ [1]. As the validation is made during run-time, an option could be added to the interpreter to stop the execution when an invariant is violated. The prototype merely logs the violations which then can be analysed post to simulation completion.

We defined the concrete syntax for the timing invariants in VDM-RT as:

```
property(trigger,ending,interval);
```

This syntax is open to discussion and it might need to be extended if the policies need to be expressed in it. The time interval has also some novelty that is noteworthy, it is now possible to specify the time unit used in the interval (s,ms,ns). The concrete syntax of the time interval definition is the following:

```
interval = nat1 ("s" | "ms" | "ns")
```

This notation is used in the examples that appear in the next subsection.

5.1 Concrete Invariants

The invariants first mentioned in Section 3 can now be expressed in the defined syntax.

C1: *A volume change must be reflected in the display within 35 ms.*

```
deadlineMet(
  #fin(Radio`AdjustVolumeUp),
  #fin(MMI`UpdateScreen),
  35 ms)
```

C2: *The screen should be updated no more than once every 500 ms.*

```
separate(
  #fin(MMI`UpdateScreen),
  #fin(MMI`UpdateScreen),
  500 ms)
```

C3: *If the volume is to be adjusted upwards and it is not currently at the maximum, the audible change should occur within 100 ms.*

```
deadlineMet(
  ( #req(MMI`HandleKeyPressUp),
    RadNavSys`radio.volume < Radio`MAX
  ),
  #fin(MMI`AdjustVolumeUp),
  100 ms)
```

The definitions are pretty self-explanatory, **#req** and **#fin** refer to the operation states request and finish respectively. The operation events are all defined over classes and one instance variable event is defined in C3. C3 trigger is a composite of a operation and a variable trigger.

5.2 System Class Extension

We recommend an extension to the system where the modeller could specify the system timing properties. We recommend such extension because these properties could be seen as a kind of system-wide timing invariants which must hold in order for the system to behave correctly. The only difference from traditional VDM invariants is that the violation of these would not cause the interpretation to stop but instead report a timing invariant violation which could after be inspected by the modeller.

```
system Sys
...
timing invariants

deadlineMet(evTrigger1, evEnder1, 400 ms);
...
separate(evTrigger2, evEnder2, 1000 ms);

end Sys
```

With the facilities provided by [11], it is possible to easily test the system in different architectures. By coupling this idea with the recorded time invariants in the system class, it is possible to easily spot which architectures respect the time behavior specification and discard the ones which do not.

5.3 Results

Figure 3 shows the resulting log of an interpretation of the model with the timing invariants. We can see that both C1 and C3 hold through the interpretation while C2 is violated twice. The log shows which invariants were violated or not and for the ones that were violated, it indicates at which point of time it happened and the responsible thread. In the graphical log representation, the places of the violation of C2 are also marked with a circle in red. Having such information readily available and facilities to go to critical points avoids a painstakingly examination of the RTLogs by the modeler, greatly enhancing his ability to reason about the model.

The results of invariants test might not be as simple as only *Pass* or *Not Pass*, like mentioned in Section 4 results might be *Inconclusive* or the invariant might not even be activated once because the trigger event has not occured at all in the chosen scenario, leading to a *Not Activated* result.

6 Concluding Remarks

In this paper we have presented an extension of the VDM-RT notation and the associated interpreter to make validation of system timing properties during run-time that builds up on the theory presented in [5]. The intention of this paper is to both demonstrate that such validation is possible to do at run-time and also to form basis for discussion on the inclusion of timing invariants in the VDM-RT language as a form of

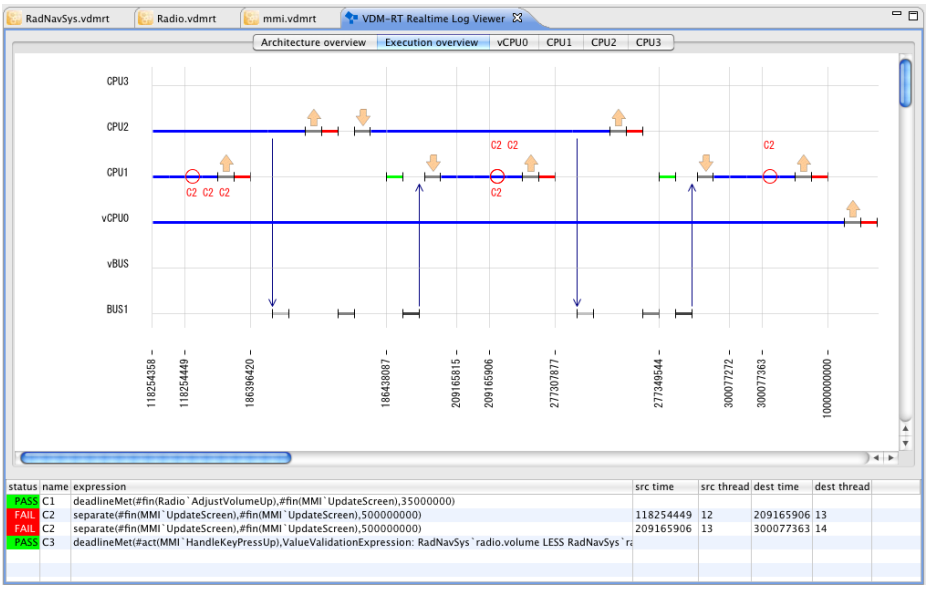

Fig. 3. Timing invariants violations represented in the logger

recording system-wide invariants related with timing which are usually very important when specifying a real-time system. The discussion could also be extended to which kind of the properties should be available for specifying these timing invariants or if their semantics needs to be adjusted. Finally we hope that the workshop can clarify whether it would be worthwhile for the user to be able to select whether violations of timing constraints should be logged or treated as run-time errors.

Acknowledgements

This work was partly supported by the EU FP7 DESTECS Project. We appreciate the input we have had from the different partners on this work. In addition we would like to thank Nick Battle and the anonymous referees for valuable input on this paper.

References

1. Battle, N.: VDMJ User Guide. Tech. rep., Fujitsu Services Ltd., UK (2009)
2. Bjørner, D.: The Vienna Development Method: Software Abstraction and Program Synthesis, Lecture Notes in Computer Science, vol. 75: Math. Studies of Information Processing. Springer-Verlag (1979)
3. Fitzgerald, J.S., Larsen, P.G.: Balancing Insight and Effort: the Industrial Uptake of Formal Methods. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) Formal Methods and Hybrid Real-Time Systems, Essays in Honour of Dines Bjørner and Chaochen Zhou on the Occasion of Their 70th Birthdays. pp. 237–254. Springer, Lecture Notes in Computer Science, Volume 4700 (September 2007), iSBN 978-3-540-75220-2

4. Fitzgerald, J.S., Larsen, P.G.: Triumphs and Challenges for the Industrial Application of Model-Oriented Formal Methods. In: Margaria, T., Philippou, A., Steffen, B. (eds.) Proc. 2nd Intl. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2007) (2007), also Technical Report CS-TR-999, School of Computing Science, Newcastle University
5. Fitzgerald, J.S., Larsen, P.G., Tjell, S., Verhoef, M.: Validation Support for Real-Time Embedded Systems in VDM++. In: Cukic, B., Dong, J. (eds.) Proc. HASE 2007: 10th IEEE High Assurance Systems Engineering Symposium. pp. 331–340. IEEE (November 2007)
6. Fitzgerald, J.S., Larsen, P.G., Verhoef, M.: Vienna Development Method. Wiley Encyclopedia of Computer Science and Engineering (2008), edited by Benjamin Wah, John Wiley & Sons, Inc
7. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object–oriented Systems. Springer, New York (2005), `http://www.vdmbook.com`
8. Jones, C.B.: Systematic Software Development Using VDM. Prentice-Hall International, Englewood Cliffs, New Jersey, second edn. (1990), iSBN 0-13-880733-7
9. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The Overture Initiative – Integrating Tools for VDM. ACM Software Engineering Notes 35(1) (January 2010)
10. Lausdahl, K., Larsen, P.G., Battle, N.: A Deterministic Interpreter Simulating a Distributed Real Time System using VDM. Submitted for publication (2011)
11. Lausdahl, K., Ribeiro, A.: Automated Exploration of Alternative System Architectures with VDM-RT. In: Submitted for publication (2011)
12. P. G. Larsen and B. S. Hansen and H. Brunn N. Plat and H. Toetenel and D. J. Andrews and J. Dawes and G. Parkin and others: Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language (December 1996)
13. Verhoef, M.: Modeling and Validating Distributed Embedded Real-Time Control Systems. Ph.D. thesis, Radboud University Nijmegen (2008), ISBN 978-90-9023705-3
14. Verhoef, M., Larsen, P.G., Hooman, J.: Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006: Formal Methods. pp. 147–162. Lecture Notes in Computer Science 4085 (2006)