# Validation Support for Distributed Real-Time Embedded Systems in VDM++

John S. Fitzgerald
Newcastle University, UK
John.Fitzgerald@ncl.ac.uk

Peter Gorm Larsen
Engineering College of Aarhus, Denmark
pgl@iha.dk

Simon Tjell
University of Aarhus, Denmark
tjell@daimi.au.dk

Marcel Verhoef
Chess and Radboud University Nijmegen, NL
Marcel.Verhoef@chess.nl

## Abstract

*We present a tool-supported approach to the validation of system-level timing properties in formal models of distributed real-time embedded systems. Our aim is to provide system architects with rapid feedback on the timing characteristics of alternative designs in the often volatile early stages of the development cycle. The approach extends the Vienna Development Method (VDM++), a formal object-oriented modeling language with facilities for describing real-time applications deployed over a distributed infrastructure. A new facility is proposed for stating and checking validation conjectures (assertions concerning real-time properties) against traces derived from the execution of scenarios on VDM++ models. We define validation conjectures and outline their semantics. We describe the checking of conjectures against execution traces as a formally-defined extension of the existing VDM++ tool set, and show tools to visualise traces and validation conjecture violations. The approach and tool support are illustrated with a case study based on an in-car radio navigation system.*

## 1 Introduction

The development of real-time embedded control systems to high levels of assurance is a major technical challenge, particularly when software is to be distributed over networked processors. The early development phases of such systems are also often characterised by complexity and volatility of requirements. In this environment, developers require tools that support the rapid evaluation of design models against system-level temporal and functional properties. Such a validation activity helps to identify requirements and design defects before a commitment is made to a particular design strategy. In these early development phases, when developer time is at a premium, the cost effectiveness and ease of use of validation tools is significant, as well as the level of rigour supplied by the modeling language and environment.

Our current work aims to use formal techniques in an accessible and cost-effective manner to support validation for models of distributed and embedded real-time systems. The approach is based on the Vienna Development Method (VDM), an established formal method which has been extended to support modeling of concurrent object-oriented systems (VDM++ [5]) and real-time and distributed systems [20]. In this paper we define new extensions to the modeling language and tools to permit the expression of system-level timing properties and support their validation against timed traces derived from the execution of scenarios on VDM++ models. The novel features of this work are the language and semantics of validation conjectures over timed distributed VDM++ models and their implementation in a proof-of-concept tool. The impact of the work is primarily on model developers and analysts, enabling explicit consideration of system-level properties during modeling process.

Section 2 introduces the current state of VDM++ technology. The extensions to accommodate checking of system-level timing properties, called *validation conjectures*, are shown in Section 3. These consist of language extensions to allow the specification of validation conjectures (Section 4) and formally specified tools extensions to identify conjecture violations in the execution traces (Section 5). We interleave the description of the language and tool extensions with an example based on a distributed in-car radio navigation system, introduced informally in the remainder of this section. Finally the approach and further work are discussed (Section 6).

## 1.1 Example: an In-car Radio Navigation System

Our example, based on an in-car radio navigation system, was introduced in the context of performance analysis [21, 12] and also as a case study in the extension of VDM++ to model timing requirements and distributed architecture [20]. The navigation system consists of several software applications running on a common distributed hardware platform. The design challenge is to develop an architecture capable of satisfying the requirements of the individual applications. In developing such an architecture, the designer will need feedback on system-level timing properties of alternative models. The model presented here reflects one of the proposals that was considered during design. It consists of three CPUs on an internal communication bus (Fig.1).

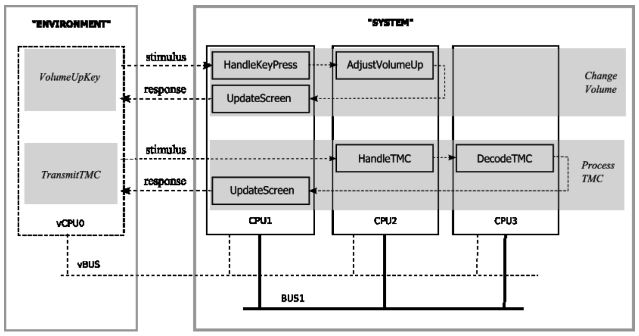


**Figure 1. Informal overview of the case study**

There are two applications, *ChangeVolume* and *ProcessTMC*, represented by the upper and lower groups of gray boxes in Fig. 1. Each application consists of three tasks. The *ChangeVolume* application increases the radio volume in response to a user pressing a "volume up" key. Within this application, the task `HandleKeyPress` takes care of user interface input handling, `AdjustVolumeUp` modifies the volume accordingly and `UpdateScreen` displays the new volume setting on the screen. The *ProcessTMC* application handles Traffic Message Channel (TMC) messages. TMC messages arrive at the `HandleTMC` task where they are checked and forwarded to the `DecodeTMC` task to be translated into human readable text which is displayed on the screen by the `UpdateTMC` task. The environment of the system is modeled by two additional applications which inject stimuli and observe the system response. The $vBUS$ and $vCPU0$ are the virtual bus and CPU on which the environment processes and the environment-system communications take place.

Before embarking on a potentially expensive rigorous development process, the system architect may wish to explore whether a given design alternative, such as the one shown above, will respect important system-level constraints. These constraints may be derived from requirements of potentially competing applications. We will use the term *validation conjecture* to mean a property to be checked against a model and the term *validation* to cover the checking process. Some examples of validation conjectures for the in-car navigation example are listed below:

**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 upward and is not currently at the maximum, the audible change should occur within 100 $ms$.

**C4:** The volume is only allowed to be at the maximum level for at most 10000 $ms$.

Validation conjectures may be inconsistent, possibly pointing to inconsistencies between applications' requirements. The validation process should help to identify such issues, and suggest strengthening or weakening of the conjectures against a specific design model.

## 2 VDM++ Technology

VDM++ is an object-oriented and model-based specification language based on the Vienna Development Method. It has a formally defined syntax, static and dynamic semantics which extend those of the ISO Standard VDM-SL language [1]. For a detailed introduction to VDM++, the reader is referred to current texts [5] and the VDM Portal [14].

VDM++ supports the construction of abstract system models composed of class specifications, each of which contains definitions of data types, instance variables and operations. Abstraction in data is provided through the use of unconstrained types and abstract collections such as sets, mappings and sequences. Functionality is modeled abstractly in terms of operations which may be described explicitly, or may be underspecified, and may even be characterised solely by postconditions. Data types may be constrained by predicate invariants and the invocation of operations may be restricted by predicate preconditions. The language is thus not in general executable, but has an executable subset.

Extensions have recently been proposed to VDM++ in order to better support the description and analysis of real-time embedded and distributed systems [18, 19]. These include primitives for modeling deployment to a distributed hardware architecture and support for asynchronous communication.

## 2.1 Example VDM++ model

This section contains extracts from the VDM++ model of the in-car navigation example initially presented in [20]. We focus on the system model rather than the environment model. There are two independent applications that consist of three tasks each. Tasks can be triggered sporadically (by external stimuli or by receiving messages from other tasks) or periodically (checking for available data on an input source or delivering data to an output). Note that task activation by external stimuli can be used to model *interrupt handling*. The interface handling tasks `HandleKeyPress` and `HandleTMC` tasks belong to this category. The other tasks in our system model are message triggered.

Application tasks are modeled by asynchronous operations in VDM++. For example, Fig. 2 shows the definitions of `AdjustVolumeUp` and `HandleTMC`, which are grouped together in the *Radio* class. The `AdjustVolumeUp` operation increases the instance variable representing the volume and then asynchronously invokes the operation `UpdateScreen` in the MMI object. Note the reference to the `RadNavSys` class, which represents the system as a whole.

```
class Radio

values
  public MAX : nat = 10

instance variables
  public volume : nat := 0

operations
  async public AdjustVolumeUp: nat ==> ()
  AdjustVolumeUp ( pno) ==
    if volume <= MAX
    then ( volume := volume + 1;
           RadNavSys`mmi.UpdateScreen(1,pno)
         );
  async public HandleTMC: nat ==> ()
  HandleTMC (pno) ==
      RadNavSys`navigation.DecodeTMC(pno);
   ...
end Radio
```

**Figure 2. The *Radio* class**

At the system level, the model must show the allocation of tasks to computation resources. A special class `CPU` is provided to create computation resources; each resource is characterised by its processing capacity, specified by the number of available cycles per unit of time and the scheduling policy. Throughout this paper, the time unit is milliseconds. For this case study, fixed priority preemptive scheduling is used, although our approach is not restricted to any policy in particular. A special class `BUS` is provided to create communication resources, each characterised by its throughput, specified by the number of messages that can be handled per unit of time and the scheduling policy that is used to determine the order of the messages being exchanged. The granularity of a message can be determined by the user. For example, it can represent a single byte or a complete Ethernet frame, whatever is most appropriate for the problem under study. For this case study, we use First Come First Served scheduling, but again the approach is not restricted to any policy in particular. An overview of the VDM++ system model is presented in Fig. 3.

```
system RadNavSys
instance variables
  -- create the application tasks
  static public mmi := new MMI();
  static public radio := new Radio();
  static public navigation :=
                 new Navigation();

  -- create CPU (policy, capacity)
  CPU1 : CPU := new CPU(<FP>, 22E6);
  CPU2 : CPU := new CPU(<FP>, 11E6);
  CPU3 : CPU := new CPU(<FP>, 113E6);

  -- create BUS
  -- (policy, capacity, topology)
  BUS1 : BUS := new BUS(<FCFS>, 72E3,
                        {CPU1, CPU2, CPU3})

operations
  -- the constructor of the system model
  public RadNavSys: () ==> RadNavSys
  RadNavSys () ==
     ( CPU1.deploy(mmi);
       CPU2.deploy(radio);
       CPU3.deploy(navigation) )
end RadNavSys
```

**Figure 3. System model for the case study**

## 2.2 Tool Support for VDM++

VDM++ is supported by an industry-strength tool set, called VDMTools, which is currently owned and further developed by CSK Systems [2]. VDM++ and VDMTools have been used successfully in several large-scale industrial projects [17, 8, 10, 4]. The tools offer syntax, type and static checking capabilities, code generators, a pretty printer and an application programmer interface. The main support for validation is by means of an interpreter allowing the execution of VDM++ models written within the executable subset of the language.

An important principle in the development of VDMTools has been that of 'taking one's own medicine'. Most of the components of VDMTools, including the type checker and interpreter, are specified in VDM and VDM++. For example, the specification of the interpreter embodies the operational semantics of the language. When extensions are proposed, it is necessary to first develop formal specifications for them and integrate them with the existing specifications before developing the implementation. This formal approach has proved particularly valuable in mastering the implementation complexity of some components of the tool set, and has in turn influenced the development of the language. We return to this point when we describe tool extensions to support validation in Section 5.

Scenarios defined by the user are essentially test cases consisting of scripts invoking the model's functionality. The interpreter executes the script over the model and returns observable results as well as an *execution trace* containing, for each internal or bus event, a time stamp and an indication of the part of the model in which it appeared. A separate tool (an Eclipse plug-in) called *showtrace* has been developed for reading execution traces, displaying them graphically so that the user can readily inspect behaviour after the execution of a scenario, and thereby gain insight into the ordering and timing of exchange of messages, activation of threads and invocation of operations.

### 2.3 Example Analysis of Radio Navigation System Model

In order to illustrate how the VDMTools interpreter can be used to examine different scenarios consider Fig. 4. This is a small scenario including two tasks in the environment for changing the volume and sending a TMC message. The `VolumeUpKey` and `TransmitTMC` objects belong to the environment. Each object is started and, once the system has responded to their stimuli, various performance characteristics are evaluated and returned. In addition to yielding a result, the execution of the scenario produces an execution trace. Note that our examples here deal with the execution of a single scenario at a time. Interleaving of scenarios is also possible and can be defined as a separate test case.

```
class World
  operations
    public RunScenario1: () ==>
             map seq of char to perfdata
    RunScenario1 () ==
      ( addEnvironmentTask("VolumeUpKey",
          new VolumeUpKey(10));
        addEnvironmentTask("TransmitTMC",
          new TransmitTMC(10));
        return { name |->
          envTasks(name).getMinMaxAverage()
          | name in set dom envTasks } );
...
end World
```

**Figure 4. Scenario for Radio Navigation System Model**

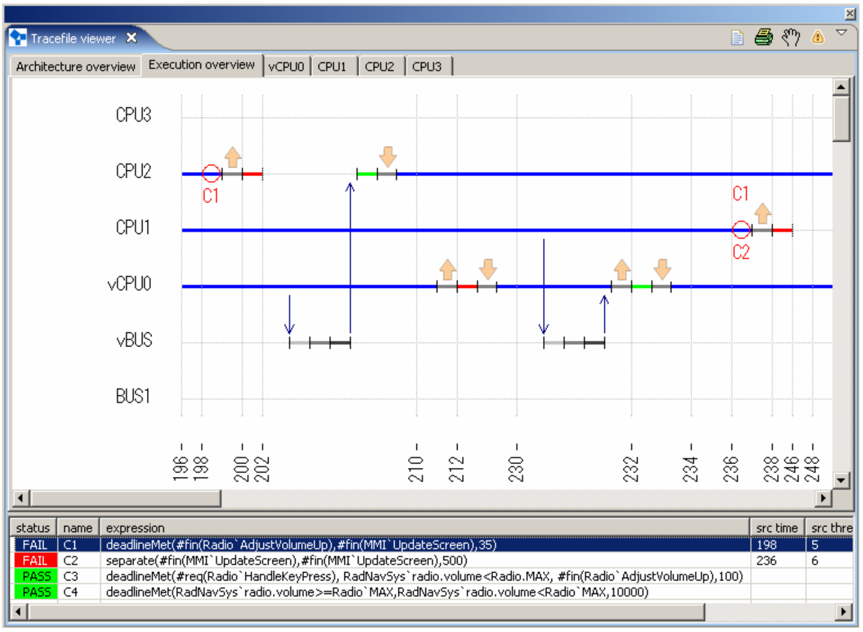
## 3 Extensions to Support Validation

The existing VDM++ and VDMTools framework has been extended so that explicit logical statements of system-level timing properties (validation conjectures) can be checked against execution traces. Fig. 5 shows the *showtrace* output resulting from the analysis of the four case study validation conjectures C1-C4 using the extended framework and the execution scenario defined in Section 2.3.

The main window shows a fragment of the execution trace. The times of significant events are displayed on the horizontal axis. Processing on each architectural unit is shown by coloured horizontal lines (colours are used to denote denote thread activities, including start-up, kill and scheduling). The thin arrows that go to and from buses indicate message passing whereas the fat arrows going up and down indicate thread swapping out and in respectively.

The features supporting validation are the list of conjectures at the bottom of the window and the circular marks on the traces that show conjecture violations. In Fig. 5, all the conjectures have been checked against the execution trace. In the example, based on the scenario shown above, conjectures `C1` and `C2` are violated and `C3` and `C4` have passed. The user can select one of the violated validation conjectures and then the appropriate point in the visualisation is displayed. In Fig. 5 this is done for conjecture `C1` (see the circles with '**C1**' next to them). Here the first `C1` circle indicates the occurrence of the first of the events in the given validation conjecture while the second one indicates the occurrence of the second event violating the constraint (in this case a deadline that is not met). In order to graphically visualise the place where a violation takes place it is easy to do so when it is possible to always relate two event occurrences with each other and that is the case with the different forms of validation conjectures that is supported at the moment.

This paper describes how the extended framework is constructed to support the form of validation illustrated above. The two main elements of Fig. 5 are the validation conjectures and the results of their evaluation. Section 4 gives an informal overview of validation conjectures. The semantics of the conjectures (the result of their evaluation over an execution trace) are embodied in the formal specification of the extended tools, described in Section 5.



| status | name | expression | src time | src thre |
|---|---|---|---|---|
| FAIL | C1 | deadlineMet(#fin(Radio`AdjustVolumeUp),#fin(MMI`UpdateScreen),35) | 198 | 5 |
| FAIL | C2 | separate(#fin(MMI`UpdateScreen),#fin(MMI`UpdateScreen),500) | 236 | 6 |
| PASS | C3 | deadlineMet(#req(Radio`HandleKeyPress), RadNavSys`radio.volume<Radio.MAX, #fin(Radio`AdjustVolumeUp),100) | | |
| PASS | C4 | deadlineMet(RadNavSys`radio.volume>=Radio`MAX,RadNavSys`radio.volume<Radio`MAX,10000) | | |

**Figure 5. Trace view showing conjecture violations**

## 4 Validation Conjectures

Validation conjectures describe the temporal relationships between system-level events that can be observed in an execution trace. An execution trace may be thought of as a finite sequence of records, one for each time unit. We use a discrete abstraction of time by using natural numbers as time values. Each record contains a set of event names for the events that occurred at that time, and a snapshot of the values of the instance variables in the system model at that time.

Events are simply temporal markers; they use up no system resources. Each event has a unique name and may occur many times in an execution trace. However, at any one time, there may be at most one occurrence of a given event. Two kinds of system-level event are detectable in an execution trace generated from a VDM++ model: *operation events* and *state transition events*. Operation events occur when operations are requested, activated, or terminated (denoted `#req(Op)`, `#act(Op)` and `#fin(Op)` respectively). State transition events occur when a predicate over the instance variables of a model becomes true.

Validation conjectures are predicates over execution traces. We will write $\mathcal{O}(e, i, t)$ to indicate that the $i$th occurrence of event $e$ takes place at time $t$. The variable $i$ ranges over the non-zero natural numbers $\mathbb{N}_1$, and $t$, representing time, ranges over the indices of the trace. For example, the simple conjecture

$$\mathcal{O}(\#\mathit{fin}(\mathit{MMI}`\mathit{UpdateScreen}), 1, 50)$$

is true in a trace where the first occurrence of the event marking the termination of the `UpdateScreen` operation is at exactly time unit 50. Note that distinct occurrences of an event must happen at different times and that the occurrence numbers increase incrementally over time[1].

It is often necessary to check a conjecture that relates to the specific values of some instance variables. For example, a designer may wish to check that a variable reaches a certain value at a specified time. In order to do this conveniently, we introduce the notion of a *state predicate*. A state predicate is a predicate over the instance variables of the system model. We will write $\mathcal{E}(p, t)$ to mean that the state predicate $p$ is true of the variables in the execution trace at time $t$.

In stating a conjecture, it may be necessary to mark the times at which a predicate becomes true. In order to support this, we introduce the notion of a *state transition event* which contains a predicate and which occurs at any time when the predicate becomes true.

The concepts defined so far are sufficient to construct useful validation conjectures. In order to allow more efficient detection and appropriate display of violations, we also identify specific forms or pattern of conjecture. When such standard forms are used, we can invoke efficient detection and display functions. In this section, we intro-

[1]The relation $\mathcal{O}$ is similar to the occurrence relation $\Theta$ in Real-Time Logic (RTL)[9], but we do not claim here to be using full RTL. The formal definition of conjecture evaluation is given in VDM-SL for uniformity with the tools framework (Section 5).

duce three such forms: *separations*, *required separations* and *deadlines*. It should be stressed that these are not intended to form a comprehensive language of validation conjectures. However, they illustrate the way in which formal tool support can be tailored to situations in which such standard formats are used.

Intuitively, separation conjectures describe a minimum separation between specified events, should the events occur. Required separations are separations in which the second event is required to occur at or after the minimum separation. Deadline conjectures state that the second event must occur before a deadline is reached after the occurrence of the first event. Below, each conjecture pattern and its semantics are introduced in turn.

A *separation* conjecture is a 5-tuple $\mathit{Separate}(e_1, c, e_2, d, m)$ where $e_1$ and $e_2$ are the names of events, $c$ is a state predicate, $d$ is the minimum acceptable delay between an occurrence of $e_1$ and any following occurrence of $e_2$ provided that $c$ evaluates to true at the occurrence time of $e_1$. If $c$ evaluates to false when $e_1$ occurs, the validation conjecture holds independently of the occurrence time of $e_2$. The Boolean flag $m$ is called the 'match flag', when set to true, indicates a requirement that the occurrence numbers of $e_1$ and $e_2$ should be equal. This allows the designer to record conjectures that describe some coordination between events. For example, we may wish to state that a stimulus and response events occur together in pairs within some time bounds, so the $i$th occurrence of the stimulus is always followed by the $i$th occurrence of the response.

A validation conjecture $\mathit{Separate}(e_1, c, e_2, d, m)$ evaluates true over an execution trace if and only if:

$$\begin{array}{l}\forall i_1, t_1 \cdot \mathcal{O}(e_1, i_1, t_1) \wedge \mathcal{E}(c, t_1) \;\Rightarrow\\ \quad \neg\,\exists i_2, t_2 \cdot \mathcal{O}(e_2, i_2, t_2) \wedge t_1 \leq t_2 < t_1 + d \wedge\\ \qquad (m \;\Rightarrow\; i_1 = i_2) \wedge (e_1 = e_2 \;\Rightarrow\; i_2 = i_1 + 1)\end{array}$$

We want to allow this and the following expressions to be used for the specification of conjectures concerning both two occurrences of different event types ($e1 \neq e2$) and two occurrences of the same event type ($e1 = e2$). In the latter case, we distinguish the first and the second instances of the same event type by a requirement to their occurrence numbers ($e_1 = e_2 \;\Rightarrow\; i_2 = i_1 + 1$). The match flag should be used with caution in this case, since it does not make sense to combine the requirement about matching occurrence numbers with an expression about two instances of the same event type.

The *required separation* conjecture is similar to the separation conjecture but additionally requires that the $e_2$ event does occur. A conjecture $\mathit{SepRequire}(e_1, c, e_2, d, m)$ evaluates to true over an execution trace if and only if:

$$\begin{array}{l}\forall i_1, t_1 \cdot \mathcal{O}(e_1, i_1, t_1) \wedge \mathcal{E}(c, t_1) \;\Rightarrow\\ \quad \neg\,\exists i_2, t_2 \cdot \mathcal{O}(e_2, i_2, t_2) \wedge t_1 \leq t_2 < t_1 + d \wedge\\ \qquad (m \;\Rightarrow\; i_1 = i_2) \wedge (e_1 = e_2 \;\Rightarrow\; i_2 = i_1 + 1) \wedge\\ \quad \exists i_3, t_3 \cdot \mathcal{O}(e_2, i_3, t_3) \wedge (m\\ \;\Rightarrow\; i_1 = i_3) \wedge (e_1 = e_2 \;\Rightarrow\; i_3 = i_1 + 1)\end{array}$$

The *Deadline* conjecture places a maximum delay on the occurrence of the reaction event. Again, the *match* option may be used to link the occurrence numbers of the stimulus and reaction events. A validation conjecture $\mathit{DeadlineMet}(e_1, c, e_2, d, m)$ consists of a stimulus event, condition and reaction event; if $c$ holds, $d$ is the maximum tolerable delay between stimulus and reaction. The conjecture evaluates true over an execution trace if and only if:

$$\begin{array}{l}\forall i_1, t_1 \cdot \mathcal{O}(e_1, i_1, t_1) \wedge \mathcal{E}(c, t_1) \;\Rightarrow\\ \quad \exists i_2, t_2 \cdot \mathcal{O}(e_2, i_2, t_2) \wedge t_1 \leq t_2 \leq t_1 + d \wedge\\ \qquad (m \;\Rightarrow\; i_1 = i_2) \wedge (e_1 = e_2 \;\Rightarrow\; i_2 = i_1 + 1)\end{array}$$

These basic forms of validation conjecture can be combined. For example, a conjecture to validate the periodic character of an event might take the form $\mathit{Periodic}(e, p, j)$ where $e$ is the periodic event, $p$ is the period and $j$ the allowable jitter. The conjecture might be defined to be true in a given execution trace if and only if:

$$\begin{array}{l}\mathit{DeadlineMet}(e, \mathit{true}, e, p + j, \mathit{false}) \wedge\\ \mathit{Separate}(e, \mathit{true}, e, p - j, \mathit{false})\end{array}$$

evaluates to true over the same trace.

The three forms of conjecture identified above are by no means exhaustive. For example, one might wish to state that the non-occurrence of an event in a specified period should trigger the occurrence of some other event. Expression of non-standard conjectures will likely entail the use of the basic occurrence relation. Nevertheless, the approach of defining common forms of conjecture is extensible and allows for tailored tool support of the kind discussed in Section 5.

## 4.1 Example Validation Conjectures

It is possible to use the simple language of validation conjectures to state system-level timing properties in the case study. In the following, we give concrete syntax representations for the conjectures C1-C4 introduced in Section 1. In most cases, the state predicate component of the conjecture is omitted and treated as `true`. In all cases, the match component $m$ is omitted and defaults to `false`.

**C1:** *A volume change must be reflected in the display within* 35 *ms.* In the `Radio` class, the `AdjustVolumeUp` operation invokes the `UpdateScreen` operation in the MMI. The conjecture is interpreted formally as a deadline on the completions of `AdjustVolumeUp` and the next screen update `MMI'UpdateScreen`. This is a weak statement in that it does not tie the screen update to the specific volume change event. It could be strengthened by adding operation parameters to the conjecture to link the stimulus and response. This can be done using the formal definitions in Section 5. However, we omit this for simplicity.

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

**C2:** *The screen should be updated no more than once every* 500 $ms$. This is interpreted as a separation constraint on the `MMI`UpdateScreen` screen operation completions.

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

As a result of formalising and validating the conjecture, the architect may observe the inconsistency between C1 and C2 that can arise if a screen update has to be performed in response to a volume change at a time less than 500 $ms$ from the last screen update. It may be appropriate to negotiate a weakening in requirements in such as case.

**C3:** *If the volume is to be adjusted upward and is not currently at the maximum, the audible change should occur within* 100 $ms$. The current volume is modeled by the value of the instance variable `(RadNavSys`radio.volume)`. The request to adjust the volume is interpreted as a deadline. The noticing of the audible change is interpreted as the termination of the `Radio`AdjustVolumeUp` operation.

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

Here again the architect may observe that the formalised conjecture may be weak by allowing many key presses to be serviced by only one volume adjustment event. A revised, stronger, conjecture linking occurrences might be appropriate here.

**C4:** *The volume is only allowed to be at the maximum level for at most* 10000 $ms$. It is interesting to note that we do not distinguish between the initiators in controlling the volume but merely observe the resulting level. The maximum amount of time in which the volume is allowed to be at the maximum level is set at 10000 $ms$.

```
deadlineMet(
  RadNavSys`radio.volume >= Radio`MAX,
  RadNavSys`radio.volume<Radio`MAX,
  10000)
```

## 5 Extended Tool Support for Validation Conjectures

The VDMTools interpreter has been extended to record additional data in the execution trace generated by running a scenario and to use this to evaluate validation conjectures. A further extension to *showtrace* allows violations to be identified and explored. In this section, we describe these extensions, focussing on those aspects that are relevant to the efficient analysis of the validation conjecture forms identified in Section 4.

In accordance with the 'taking one's own medicine' principle for developing VDMTools, we began from the formal specification of the interpreter before developing the code to implement our extensions. The interpreter specification is substantial – over 500 pages of VDM-SL interleaved with informal explanatory text. Thus, VDM is used both as the meta-language as well as the source language in the part described in this section. One additional module called `VC` has been added containing the formal descriptions of data structures and operations for logging execution trace data and for evaluating validation conjectures. The `VC` module is only 400 lines of VDM-SL and below we will show extracts from this.

The intention is that, once a scenario has been executed, it should be possible to evaluate a set of validation conjectures over the execution trace. A further extension of the *showtrace* tool permits the graphical indication of conjecture violations on top of the visualisation of the simulation of the deployed applications. This is intended to speed up the error detection and correction cycle at the abstract design level because the detected violations act as counter examples that are easy to understand.

The `VC` module specifies efficient operations that can determine the validity of a validation conjecture over a given execution trace. If a violation is detected it will yield a tuple of information that uniquely identifies for *showtrace* the point at which the violation takes place. In order to provide for efficient checking of conjectures, the large execution trace file is not searched directly. Instead, the trace is represented in an optimised form. The `VC` module state has the following form:

```
state VCState of
  ophistmap : map AS`Name to OpHist
  instvarhistmap : map AS`Name
                       to InstVarHist
end
```

The state contains a mapping from operation names to operation histories (`OpHist`), and from instance variable names[2] to their histories (`InstVarHist`). The `OpHist` type splits the execution trace for a given operation into traces of the request, activation and finish events. Each event trace is a sequence of records, each containing a timestamp and record of the operation inputs (for a request

[2]The abstract syntax of VDM++ is described in a module called `AS`, in which names of identifiers are denoted by the type `Name`.

event) or result (for a finish event) as well as a thread identifier. The history of changes made to instance variables is stored with the actual value assigned to it (a semantic value, denoted as VAL). Formally:

```
OpHist :: reqs : seq of Req
          acts : seq of Act
          fins : seq of Fin;

Req :: tim   : nat
       arg   : [seq of VAL]
       thrid : nat;

Act :: tim   : nat
       thrid : nat;

Fin :: tim    : nat
       result : [VAL]
       thrid  : nat;

InstVarHist = seq of InstVar;

InstVar :: tim : nat
           val : VAL
           thrid : nat;
```

All these type definitions include a field called thrid that is used to keep track of the thread identification of the requesting thread which is used when the threads and flow of control is presented visually.

Having separated out the execution trace information in this fashion one can check for possible violations of a given validation conjecture. For example, the operation performing the check for a violation of a deadline conjecture on the VC module state is defined as follows:

```
EvalDeadlineMet: DeadlineMet ==>
                 [ nat * ThreadId *
                   [nat] * [ThreadId] ]

EvalDeadlineMet(
  mk_DeadlineMet(ev1,p,ev2,max,match)) ==
    if match
    then
      MatchCheck(ev1,p,ev2,max,<MAX>,false)
    else
      AnyCheck(ev1,p,ev2,max,<MAX>,false);
```

Different checks are made depending upon the match value in the conjecture. If no violation is found, the operation yields a nil value. Otherwise, it returns a tuple indicating the location of a violation. This tuple contains the time and thread identifier indicating the first event in the validation conjecture ev1 and also the time and thread identifier of the second event ev2 (if it does not occur at all the special value nil is used again). Auxiliary operations extract lists of events of interest and use these to evaluate whether a violation occurred. As an example, consider the MatchCheck operation presented below. This operation performs a check for a violation of a conjecture in which the *m* flag is true. Thus, we expect occurrence numbers of the events linked in the conjecture to be the same.

```
MatchCheck: EventExpr * Pred * EventExpr *
  nat * Kind * bool ==> [nat * ThreadId *
    [nat] * [ThreadId]]
MatchCheck(ev1,pred,ev2,delay,kind,req) ==
  let list1 = FindList(ev1),
      list2 = FindList(ev2)
  in
    (for index = 1 to len list1 do
      let t1 = list1(index).tim,
          t2 = if index in set inds list2
               then list2(index).tim
               else <INF>
      in
        (if not PredSatisfied(pred,t1)
         then skip
         elseif t2 = <INF> and req
         then return
                mk_(t1,list1(index).thrid,
                  nil,nil)
         elseif t2 <> <INF> and
                Violation(t1,t2,delay,kind)
         then return
                mk_(t1,list1(index).thrid,
                  t2,list2(index).thrid)
        );
     return nil);
```

Within MatchCheck, the auxiliary operation FindList extracts the list of a particular event's occurrences, so the variable index corresponds to the occurrence number. This checks all instances of the first event ev1 and looks for a violation in the matching occurrence of ev2 (or if no matching is present whether ev2 is required). This operation illustrates violation checking; the other algorithms are similar in nature. The violation information for each validation conjecture is handed over to the *showtrace* tool for visualisation.

If there were no need for predicates very efficient solvers for propositional logic could be used. However, we need the possibility to express the validation conjectures in a simple form. The checking of predicates is carried out using the *PredSatisfied* operation. It takes a predicate and the time at which the predicate needs to be checked. This is defined as follows:

```
PredSatisfied: Pred * nat ==> bool
PredSatisfied(pred,t) ==
  let mk_Pred(var,op,num) = pred
  in
    if var in set dom instvarhistmap
    then
```

```
      let hist = instvarhistmap(var),
          mk_InstVar(firstt,-,-) = hd hist
      in
        if t < firstt
        then return false
        else (for i = 2 to len hist do
                if hist(i-1).tim <= t and
                   t < hist(i).tim
                then return
                     EvalOp(hist(i-1).val,
                            op,num);
              return
              EvalOp(hist(len hist).val,
                     op,num)
             )
   else return false;
```

The predicate is set to be false if the time requested is before the instance variable in question has been initialised. Otherwise it looks through the list of updates that has been made to the instance variable and evaluates the operator at the time requested.

## 6 Concluding Remarks

There is a considerable body of work extending formal model-oriented specification languages to allow the expression of temporal properties alongside functionality. Some, in common with VDM++, aim to combine the expression of temporal behaviour with object-oriented or modular structuring. Possibly the closest examples are Timed RSL [6], combinations of Timed CSP with Object-Z [3] and Timed extensions to B such as [15]. The majority of these works focus on support for verification of design steps by model checking and proof rather than validation in early development stages. None of them deal explicitly with deployment. In the context of UML, the strand of research on validation of timed system models based on a real-time profile [13, 7] proposes the statement of 'duration constraints' between events by means of observer state machines or OCL-like expressions. Work on the UniFrame framework [16] places a VDM++ model of a distributed real-time system as part of a chain of formalisms for handling functional and non-functional properties. The proposed tool chain includes timed trace analysis based on VDM++ models, although this proposal has not been implemented.

There has been considerable interest in the analysis of formal models by animation based on the direct execution of models (rather than a translated model or execution of code derived from a model). There have been recent industrial successes. For example, animation (extensive testing) of an executable model plays a vital role in the validation of an embedded integrated circuit for cellular telephones [10]. Liu's work based on the SOFL language [11], which contains elements derived from VDM, also supports direct animation of models, in this case from systematically derived execution scenarios.

The aim of our work has been to support the validation of system-level temporal properties in an accessible and cost-effective manner. How far have we gone towards achieving this? The approach has a formal basis, but the goal is pragmatic. Building on an existing modeling framework in VDM++ [20], we have presented a new semantically well-founded facility for stating and checking system-level validation conjectures against traces derived from the execution of models that describe distributed real-time systems. Tool extensions have been defined formally and have been implemented to proof-of-concept level. The outcomes of each trace analysis are made accessible by using a graphical display and exploiting defined validation conjecture forms for which specific violation detection and display features have been defined. Together, these facilities enable the system architect to adjust the functionality, its deployment, the architecture or the conjecture itself.

It should be stressed that the validation of execution traces does not replace formal verification: a failing conjecture is the symptom of a possible design defect but a passing conjecture is not a proof of correctness with respect to the real-time constraints. Our objective here is to provide rapid assessment of the key properties of a design model, helping to identify deficiencies in requirements and system-level conjectures in the very early phases of a high assurance development process.

There are some limitations to the framework as developed so far. First, although our models are loosely specified, each scenario execution produces only one execution trace because the implementation of the operational semantics enforces determinism in order to ensure reproducibility over multiple traces. Within a single execution, we do allow non-determinism. Second, the tools do not yet support on-the-fly evaluation of validation conjectures, although this could readily be accommodated. Third, we need to widen the range of conjecture forms for which specialised checking and display tools are available.

There are several other directions in which the proof of concept work reported here might be extended. Once conjectures have been defined and have been used to validate the execution of the model (which also serves as a validation of the conjectures themselves), they can be used (with event transformation) to validate log files generated by the final implementation of a system. A further important task is to begin to evaluate the reliability of the results as an aid to decision-making among design alternatives. Further, the validation framework discussed here might be extended to support the evaluation of fault tolerance strategies at the architectural level. Experiments combining the interpreter (executing a model of a controller) with a continuous time simulator suggest that it is possible to model faults at the interface between the two without clouding the models of either the controller or the environment process. Finally, we are investigating the provision of automated proof of validation conjectures on models in constrained situations.

The purpose of the work reported here has been to enhance the range of tools available to designers of distributed embedded real-time systems in early development stages. The approach is based on the use of abstract system models that have a formal basis and so can benefit from rigorous analysis. The priority has been to provide rapid feedback on the timing properties of abstract system models by exploiting information gathered during the execution of scenarios. This form of validation is not seen as a solitary technique, but can be used in conjunction with other forms of analysis to support design-time trade-off and decision-making.

**Acknowledgments** The authors wish to thank Jozef Hooman, Jeremy Bryans, Cliff Jones and the anonymous reviewers for valuable input on this and previous drafts. Fitzgerald is grateful to the European Network on Resilience for Survivability in IST (ReSIST) and the EPSRC Platform project on Trustworthy Ambient Systems.

## References

[1] D. Andrews, editor. *Vienna Development Method – Specification Language – Part 1: Base language*. International Organization for Standardization, December 1996. International Standard ISO/IEC 13817-1.

[2] CSK. VDMTools homepage. *http://www.vdmtools.jp/en/*, 2007.

[3] J. Derrick. Timed CSP and Object-Z. In D. Bert, J. Bowen, S. King, and M. Walden, editors, *ZB 2003: Formal Specification and Development in Z and B*, Berlin, Germany, June 2003. Springer-Verlag, Lecture Notes in Computer Science volume 2651.

[4] J. Fitzgerald and P. Larsen. Triumphs and Challenges for the Industrial Application of Model-Oriented Formal Methods. In T. Margaria, A. Philippou, and B. Steffen, editors, *Proc. 2nd Intl. Symp. on Leveraging Applications of Formal Methods, Verification and Validation*. IEEE, 2007. To appear. Also Technical Report CS-TR-999, School of Computing Science, Newcastle University.

[5] J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef. *Validated Designs for Object–oriented Systems*. Springer, New York, 2005.

[6] C. George and X. Yong. An Operational Semantics for Timed RAISE. Technical Report 149, United Nations University International Institute for Software Technology, November 1998.

[7] S. Graf and J. Hooman. Correct Development of Embedded Systems. In *Proc. European Workshop on Software Architecture: Languages, Styles, Models, Tools and Applications (EWSA 2004), co-located with ICSE 2004*, pages 241–249. Springer Verlag Lecture Notes in Computer Science Volume 3047, 2004.

[8] J. Hörl and B. K. Aichernig. Validating Voice Communication Requirements Using Lightweight Formal Methods. *IEEE Software*, 13–3:21–27, May 2000.

[9] F. Jahanian and A. K.-L. Mok. Safety Analysis of Timing Properties in Real-Time Systems. *IEEE Transactions on Software Engineering*, SE-12(9):890–904, September 1986.

[10] T. Kurita, T. Oota, and Y. Nakatsugawa. Formal Specification of an Embedded IC for Cellular Phones. In *Proc. Software Symposium 2005*, pages 73–80. Software Engineers Associates of Japan, June 2005. (in Japanese).

[11] S. Liu and H. Wang. An automated approach to specification animation for validation. *Journal of Systems and Software*, 80:1271–1285, 2007.

[12] Martijn Hendriks and Marcel Verhoef. Timed Automata Based Analysis of Embedded System Architectures. In *Proc. 14th Intl. Workshop on Parallel and Distributed Real-Time Systems*. IEEE, 2006. In Procs. IPDPS 2006, DOI 10.1109/IPDPS.2006.1639422.

[13] I. Ober, S. Graf, and I. Ober. Validating Timed UML Models by Simulation and Verification. *Software Tools for Technology Transfer*, 8(2):128–145, 2006.

[14] Overture Group. The VDM Portal. *http://www.vdmportal.org*, 2007.

[15] M. Rached, J.-P. Bodeveix, M. Filali, and O. Nasr. A Timed B method for Modelling Real Time Reactive Systems. In *2nd South-East European Workshop on Formal Methods (SEEFM 05), Ohrid, 18-19 Nov 2005*, pages 181–195. South-East European Research Centre (SEERC), 2006.

[16] Shih-hsi Liu. Validation of Distributed Real-Time and Embedded System Composition in UniFrame. In *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference*, pages 303–304. ACM Press, 2004.

[17] M. Van den Berg, M. Verhoef, and M. Wigmans. Formal Specification of an Auctioning System Using VDM++ and UML – an Industrial Usage Report. In J. Fitzgerald and P. G. Larsen, editors, *VDM in Practice*, pages 85–93, September 1999.

[18] M. Verhoef. On the use of VDM++ for Specifying Real-Time Systems. In J. S. Fitzgerald, P. G. Larsen, and N. Plat, editors, *Towards Next Generation Tools for VDM*, pages 26–43, School of Computing Science, Newcastle University, Technical Report CS-TR-969, June 2006.

[19] M. Verhoef and P. G. Larsen. Interpreting Distributed System Architectures Using VDM++ – A Case Study. In B. Sauser and G. Muller, editors, *5th Annual Conference on Systems Engineering Research*, March 2007.

[20] M. Verhoef, P. G. Larsen, and J. Hooman. Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM 2006: Formal Methods*, pages 147–162. Lecture Notes in Computer Science 4085, 2006.

[21] E. Wandeler, L. Thiele, M. Verhoef, and P. Lieverse. System Architecture Evaluation Using Modular Performance Analysis – A Case Study. *International Journal on Software Tools for Technology Transfer*, 8(6):649–667, 2006.