UNIVERSITY OF
NEWCASTLE

University of Newcastle upon Tyne

COMPUTING SCIENCE

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

J. S. Fitzgerald, P. G. Larsen, S. Tjell, M. Verhoef

TECHNICAL REPORT SERIES

No. CS-TR-1017 April, 2007

TECHNICAL REPORT SERIES

No. CS-TR-1017 **April, 2007**

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

J. S. Fitzgerald, P. G. Larsen, S. Tjell, M. Verhoef.

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 implementation of conjectures against execution traces as a formally-defined extension to the existing VDM++ tool set, and show tools to support the visualisation of traces and validation conjecture violations. The approach and tool support are illustrated with a case study based on an in-car radio and navigation system.

© 2007 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne,
Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

Bibliographical details

FITZGERALD, J. S., LARSEN, P. G., TJELL, S., VERHOEF, M..

Validation Support for Distributed Real-Time Embedded Systems in VDM++
[By] J. S. Fitzgerald, P. G. Larsen, S. Tjell, M. Verhoef.

Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Science, 2007.

(University of Newcastle upon Tyne, Computing Science, Technical Report Series, No. CS-TR-1017)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE
Computing Science. Technical Report Series. CS-TR-1017

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 implementation of conjectures against execution traces as a formally-defined extension to the existing VDM++ tool set, and show tools to support the visualisation of traces and validation conjecture violations. The approach and tool support are illustrated with a case study based on an in-car radio and navigation system.

About the author

John Fitzgerald is Reader in Computing Science at Newcastle University. His research concerns the use of formal methods to support the design of systems that reconfigure in response to threats. John leads work on resilience-explicit computing in the ReSIST Network on Resilience in IST and is co-Investigator in the Trustworthy Ambient Systems project. He works closely with the aerospace industry and previously established validation activities in Transitive Ltd, a successful SME in the embedded software market. He is Chairman of Formal Methods Europe.

Peter Gorm Larsen is Professor of Computer Technology and Embedded Systems at The Engineering College of Aarhus, Denmark and an independent consultant. An authority on system modelling, particularly the Vienna Development Method, he has pioneered the development of industrial-strength tool support for model-oriented specification languages, heading the group that initially developed VDMTools(R) now owned by CSK.

Simon Tjell is a doctoral student and member of the Coloured Petri Nets group at the Department of Computer Science in the University of Aarhus. He is working on development of approaches to model-based engineering of embedded systems.

Marcel Verhoef is a consultant in Embedded Systems Architecture and a member of the Innovation Team at CHESS, a Netherlands SME specialising in the development of high-end computer based systems for business critical applications. He is working on developing and improving multi-disciplinary design methodologies for real-time and distributed embedded systems. He is also a doctoral candidate at the Radboud University Nijmegen.

Suggested keywords

REAL-TIME SYSTEMS,
DISTRIBUTED SYSTEMS,
EMBEDDED SYSTEMS,
VALIDATION,
FORMAL METHODS,
MODELING

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

John Fitzgerald[1], Peter Gorm Larsen[2], Simon Tjell[3], and Marcel Verhoef[4]

[1] School of Computing Science, Newcastle University, UK
[2] Engineering College of Aarhus, Denmark
[3] Department of Computer Science, University of Aarhus, Denmark
[4] Chess and Radboud University Nijmegen, NL
`John.Fitzgerald@ncl.ac.uk`, `pgl@iha.dk`,
`tjell@daimi.au.dk`, `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 implementation of conjectures against execution traces as a formally-defined extension to the existing VDM++ tool set, and show tools to support the visualisation of traces and validation conjecture violations. The approach and tool support are illustrated with a case study based on an in-car radio and navigation system.

1 Introduction

The early stages in the life cycle of computer-based systems are often characterized by complexity and volatility of requirements. This is particularly challenging when software is to be distributed over networked processors in embedded and control applications. In this environment, developers require tools that permit the rapid validation of design models against system-level temporal and functional properties. The cost-effectiveness and ease of use of such tools are significant when developer time is at a premium.

Our current work aims to use formal modeling techniques in an accessible and cost-effective manner to support validation for 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++ [1]) and with capabilities for modeling real-time and distributed systems [2]. In this paper we propose further enhancements to the modeling language and tools that permit the expression of system-level timing properties and support their validation against timed traces derived from the execution of scenarios on the abstract VDM++ model.

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).

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 [3,4] and also as a case study in the extension of VDM++ to model timing requirements and distributed architecture [2]. 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).

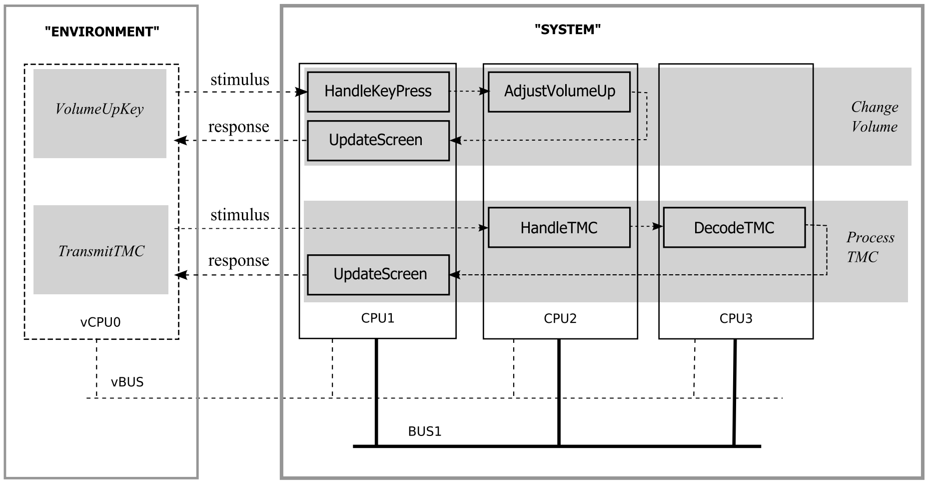

Fig. 1. Informal overview of the case study

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.

Examples of possible requirements from the individual applications are shown below. A system architect would wish to gain confidence that a given design alternative, such as the one shown above, will respect these constraints.

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 $10000\,ms$. This conjecture would be used if, for example, the system has a component designed to automatically lower the volume when it stays too high for too long.

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 as well as a proof theory. VDM++ is largely a superset of the ISO standardized VDM-SL notation [5] . For a detailed introduction to VDM++, the reader is referred to current texts [1] and the VDM Portal [6].

2.1 VDM++ for Timed Distributed Systems

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 characterized 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 [7,8,9]. These include primitives for modeling deployment to a distributed hardware architecture and support for asynchronous communication.

2.2 Example VDM++ model

This section contains extracts from the VDM++ model of the in-car navigation example initially presented in [2]. 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
```

Fig. 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 characterized 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 characterized 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);        -- deploy MMI
        CPU2.deploy(radio);      -- deploy Radio
        CPU3.deploy(navigation) ) -- deploy Navigation
end RadNavSys
```

Fig. 3. The top-level system model for the case study

2.3 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 [10]. VDM++ and VDMTools have been used successfully in several large-scale industrial projects [11,12,13,14]. 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’. A large part of VDMTools is 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 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.

2.4 Example: Analysis of the 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
```

Fig. 4. A scenario for the 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 2.4

The main body of the window shows a fragment from 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). Arrows indicate message passing and thread swapping.

The important new feature supporting validation is the list of conjectures at the bottom of the window. 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 round circles with `C1` written next to it).

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)		

Fig. 5. Trace view showing conjecture violations

This section 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 and introduces the syntax of three basic forms of conjecture supported in the extended framework. 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.

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. 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* 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[5].

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.

It would be possible to construct validation conjectures just using the concepts defined so far. However, in order to promote ease of use, several higher level validation conjecture patterns are defined in order to allow common forms of conjecture to be expressed or composed. Three simple forms of conjecture are currently supported: *separations*, *"required" separations* and *deadlines*. 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

---

[5] The relation $\mathcal{O}$ is similar to the occurrence relation Θ in Real-Time Logic (RTL)[15], 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).

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. These three simple forms are not intended to be exhaustive, but could provide a basis for a language of more sophisticated conjectures. 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:

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

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 might be used to build up a more sophisticated language. 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:

$$\mathit{DeadlineMet}(e, \mathit{true}, e, p + j, \mathit{false}) \wedge \mathit{Separate}(e, \mathit{true}, e, p\text{-}j, \mathit{false})$$

evaluates to true over the same trace.

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)
```

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)
```

C4: *The volume is only allowed to be at the maximum level for at most* 10000 *ms. This could be case if the system was designed to automatically lower the volume.* 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 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. 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 small extracts from this. Note that the VDM-SL text in this section is the formal description of the interpreter extensions, not VDM++ source that the interpreter actually executes.

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. In case of a violation 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 operations to operation histories (`OpHist`), and from instance variable names 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 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;
```

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 occured. 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 that 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 occurences, 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 occurence 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.

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 the potential benefits of object-oriented or modular structuring. Possibly the closest examples are are Timed RSL [16], combinations of Timed CSP with Object-Z [17] and Timed extensions to B such as [18]. 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 [19,20] proposes the statement of 'duration constraints' between events by means of observer state machines or OCL-like expressions.

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 and this has led to a focus on tools. Building on an existing modeling framework in VDM++, we have presented a new facility for stating and checking validation conjectures against traces derived from the execution of models that describe distributed real-time systems. Tools extensions have been defined formally and have been implemented to proof-of-concept level. Accessibility has been addressed by providing template validation conjectures which may be combined, and by providing graphical browsing of execution traces in which the conjecture violations are identified. 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. Nevertheless, when development resources, especially time, are short, rapid validation conjecture checking of the kind that we propose does offer a means of assessing key properties of the formal model.

There are some limitations to the framework as developed so far. First, executions are deterministic, so each scenario execution produces only one possible execution trace. Second, the tools do not yet support on-the-fly evaluation of validation conjectures, although this could readily be accommodated.

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 Jeremy Bryans for valuable comments when writing this paper. JSF is grateful to the European Network on Resilience for Survivability in IST (ReSIST).

References

1. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object–oriented Systems. Springer, New York (2005).
2. 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, Lecture Notes in Computer Science 4085 (2006) 147–162.
3. Wandeler, E., Thiele, L., Verhoef, M., Lieverse, P.: System Architecture Evaluation Using Modular Performance Analysis – A Case Study. International Journal on Software Tools for Technology Transfer **8**(6) (2006) 649–667.

4. 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.
5. Andrews, D., ed.: Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language. International Organization for Standardization (December 1996) International Standard ISO/IEC 13817-1.
6. Overture Group: The VDM Portal. *http://www.vdmportal.org* (2007).
7. Verhoef, M.: On the use of VDM++ for Specifying Real-Time Systems. In Fitzgerald, J.S., Larsen, P.G., Plat, N., eds.: Towards Next Generation Tools for VDM: Contributions to the First International Overture Workshop, Newcastle, July 2005, School of Computing Science, Newcastle University, Technical Report CS-TR-969 (June 2006) 26–43.
8. Verhoef, M., Larsen, P.G.: Enhancing VDM++ for Modeling Distributed Embedded Real-time Systems. Technical Report (to appear), Radboud University Nijmegen (March 2006) A preliminary version of this report is available on-line at *http://www.cs.ru.nl/∼marcelv/vdm/*.
9. Verhoef, M., Larsen, P.G.: Interpreting Distributed System Architectures Using VDM++ – A Case Study. In Sauser, B., Muller, G., eds.: 5th Annual Conference on Systems Engineering Research. (March 2007).
10. CSK: VDMTools homepage. *http://www.vdmtools.jp/en/* (2007).
11. Van den Berg, M., Verhoef, M., Wigmans, M.: Formal Specification of an Auctioning System Using VDM++ and UML – an Industrial Usage Report. In Fitzgerald, J., Larsen, P.G., eds.: VDM in Practice. (September 1999) 85–93.
12. Hörl, J., Aichernig, B.K.: Validating Voice Communication Requirements Using Lightweight Formal Methods. IEEE Software **13–3** (May 2000) 21–27.
13. Kurita, T., Oota, T., Nakatsugawa, Y.: Formal Specification of an Embedded IC for Cellular Phones. In: Proc. Software Symposium 2005, Software Engineers Associates of Japan (June 2005) 73–80 (in Japanese).
14. Fitzgerald, J., Larsen, P.: 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. IEEE (2007) To appear. Also Technical Report CS-TR-999, School of Computing Science, Newcastle University.
15. Jahanian, F., Mok, A.K.L.: Safety Analysis of Timing Properties in Real-Time Systems. IEEE Transactions on Software Engineering **SE-12**(9) (September 1986) 890–904.
16. George, C., Yong, X.: An Operational Semantics for Timed RAISE. Technical Report 149, Unoted Nations University International Institute for Software Technology (November 1998).
17. Derrick, J.: Timed CSP and Object-Z. In Bert, D., Bowen, J., King, S., Walden, M., eds.: ZB 2003: Formal Specification and Development in Z and B, Berlin, Germany, Springer-Verlag, Lecture Notes in Computer Science volume 2651 (June 2003).
18. Rached, M., Bodeveix, J.P., Filali, M., Nasr, O.: A Timed B method for Modelling Real Time Reactive Systems. In: 2nd South-East European Workshop on Formal Methods (SEEFM 05), Formal Methods: Challenges in the Business World, Ohrid, 18-19 Nov 2005, South-East European Research Centre (SEERC) (2006) 181–195.
19. Ober, I., Graf, S., Ober, I.: Validating Timed UML Models by Simulation and Verification. Software Tools for Technology Transfer **8**(2) (2006) 128–145.
20. Graf, S., Hooman, J.: 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, Springer Verlag Lecture Notes in Computer Science Volume 3047 (2004) 241–249.