PROCEEDINGS OF THE 9TH OVERTURE WORKSHOP

Electrical and Computer Engineering
Technical Report ECE-TR-2

AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

DATA SHEET

Title: PROCEEDINGS OF THE 9TH OVERTURE WORKSHOP
Subtitle: Electrical and Computer Engineering
Series title and no.: Technical report ECE-TR-2

Authors: Sune Wolff & John Fitzgerald
Department of Engineering – Electrical and Computer Engineering,
Aarhus University

Internet version: The report is available in electronic format (pdf) at the Department of Engineering website http://www.eng.au.dk.

Publisher: Aarhus University©
URL: **http**://www.eng.au.dk

Year of publication: 2012 Pages: 125
Editing completed: June 2011

Abstract: This report contains the proceedings of The 9th Overture Workshop, held in Limerick on 20th June 2011.

Keywords: Overture, Workshop, 2011

Please cite as: Sune Wolff & John Fitzgerald, Proceedings of the 9th Overture Workshop 2012. Department of Engineering, Aarhus University. Denmark.125 pp. - Technical report ECE-TR-2

Front image: Logo, Overture Open Source Community

ISSN: 2245-2087

PROCEEDINGS OF THE 9TH OVERTURE WORKSHOP

Sune Wolff — Aarhus University, Department of Engineering
John Fitzgerald — Newcastle University

Abstract

This report contains the proceedings of the 9th Overture Workshop, held in Limerick on 20th June 2011.

Table of Contents

Introduction

Overture (`www.overturetool.org`) is by now a well established open-source community initiative that has developed a range of modern tools to support the construction and analysis of models expressed in the VDM (Vienna Development Method) family of notations. Similarly, the community's workshops have become a fixture since the first such event was held in 2005.

This volume represents the proceedings of the ninth Overture Workshop, held at LERO, Limerick, Ireland on 20 June 2011, as part of the FM 2011 symposium. As with all the Overture workshops, its purpose was to foster an active community of researchers and practitioners working with VDM in both academia and industry. The organizers were:

- Sune Wolff of Aarhus University in Denmark, and
- John Fitzgerald of Newcastle University, UK.

Members of the Programme Committee were:

- Nick Battle, UK;
- Dines Bjørner, Denmark;
- Cliff Jones, UK;
- Peter Gorm Larsen, Denmark;
- Ken Pierce, UK;
- Nico Plat, Netherlands; and
- Shin Sahara, Japan.

For the ninth workshop, we were delighted to welcome contributions from Augusto Ribeiro, Kenneth Lausdahl and Peter Gorm Larsen on the real-time extensions VDM-RT, addressing issues that, along with those raised in Ken Pierce's contribution on threads, have been highlighted by the ongoing work on co-modelling and co-simulation for embedded systems design in the DESTECS

project (`www.destecs.org`). Also in the control systems domain was a thorough and systematic analysis of the safety analysis of power transmission in construction equipment by Takayuki Mori from Komatsu in Japan.

Papers by Shigeru Kusakabe, Yoichi Omori and Keijiro Araki, and by Hiroshi Mochio and Fuyuki Ishikawa, provided a more methodological strand to the workshop, while Claus Ballegaard Nielsen's paper on VDM's application in analysing a process management tool provided a further application story.

Ken Pierce's paper on proof construction looked to a long-running area for tools development, coupled closely to deep semantic issues, complemented by a presentation from Anne Haxthausen (not in these proceedings), on the Semantics of a VDM Core Language in COQ.

The papers and presentations are available on-line on the Overture project web-site (http://www.overturetool.org), which also includes online proceedings of the previous Overture workshops. We hope that this volume shows the continuing variety of research and application in the community surrounding this formal method.

October 2011

Sune Wolff
John Fitzgerald

List of Participants

Nick Battle Fujitsu UK

John Fitzgerald Newcastle University

Anne Haxthausen Technical University of Denmark

Fuyuki Ishikawa National Institute of Informatics

Shigeru Kusakabe Kyushu University

Peter Gorm Larsen Aarhus School of Engineering

Kenneth Lausdahl Aarhus School of Engineering

Hiroshi Mochio Chikushi Jogakuen University

Takayuki Mori Newcastle University

Claus Nielsen Aarhus School of Engineering

Nico Plat West Consulting B.V.

Ken Pierce Newcastle University

Augusto Ribeiro Aarhus School of Engineering

Shin Sahara CSK Corporation

Marcel Verhoef CHESS Embedded Technology B.V.

Sune Wolff Aarhus School of Engineering

Jim Woodcock University of York

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

Augusto Ribeiro, Kenneth Lausdahl, and Peter Gorm Larsen

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

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

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

1 Introduction

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

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

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

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

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

2 VDM-RT

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

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

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

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

3 Case study

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

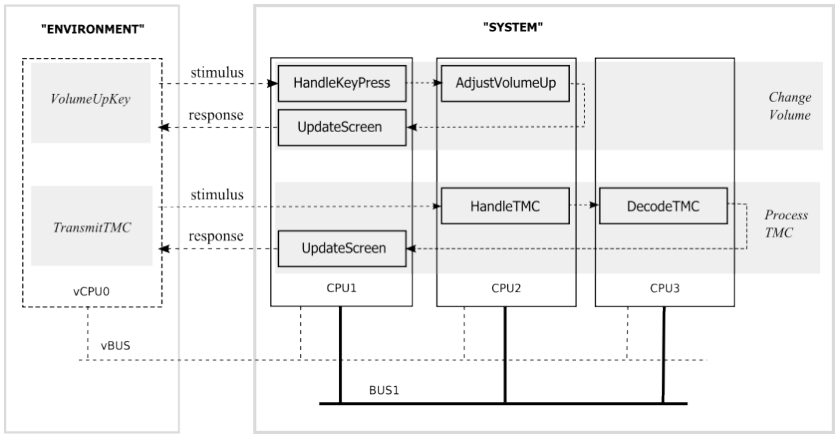

Fig. 1. Overview of the In-car Radio Navigation System

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

```
class Radio

values
  public MAX : nat = 10;

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

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

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

end Radio
```

Listing 1.1. Snippet of the Radio class

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

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

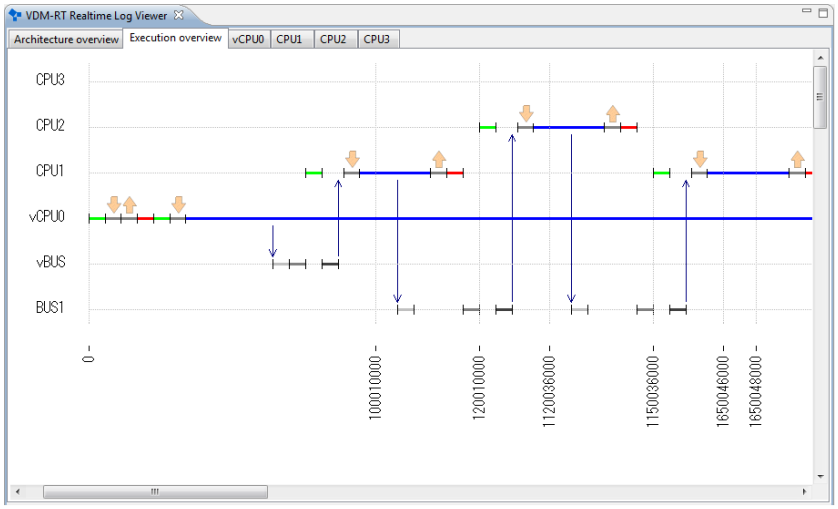

Fig. 2. Log showing one of the executions of the model

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

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

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

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

4 Timing Invariants

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.1 Events

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

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

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

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

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

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

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

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

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

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

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

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

4.2 Invariant Instances

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5 Run-Time Invariant Checking

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

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

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

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

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

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

5.1 Concrete Invariants

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

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

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

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

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

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

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

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

5.2 System Class Extension

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

```
system Sys
...
timing invariants

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

end Sys
```

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

5.3 Results

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

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

6 Concluding Remarks

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

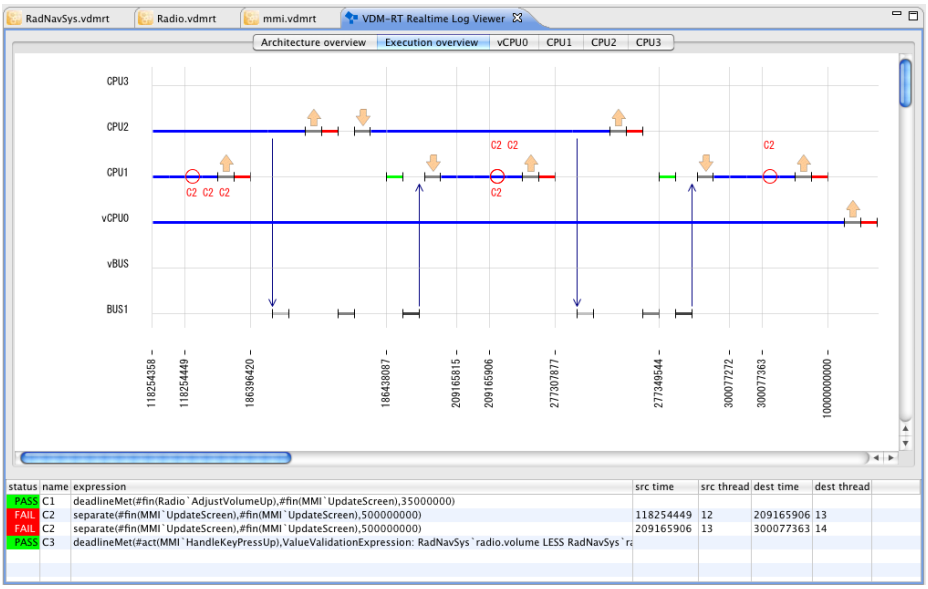

Fig. 3. Timing invariants violations represented in the logger

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

Acknowledgements

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

References

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

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

Automated Exploration of Alternative System Architectures with VDM-RT

Kenneth Lausdahl and Augusto Ribeiro

Aarhus School of Engineering, Dalgas Avenue 2, DK-8000 Aarhus C, Denmark

Abstract. Choosing the optimal deployment of a distributed embedded application onto alternative hardware configurations is often difficult and time consuming. When developing a new product, a company must chose a hardware architecture that ensures both that the system behaves correctly according to its functional and timing specifications but also keeps its production cost at a minimum. The investigation to find this tradeoff between cost and performance can be very expensive if carried out at implementation time. A company can save money and development time if there is a possibility to quickly explore the design alternatives before the start of the implementation. In this paper we describe a method and associated tool support to assist in finding the best system design solution.

1 Introduction

As distributed real-time embedded systems become more and more prevalent around us, new techniques must to be used to ensure lower costs of development while keeping the service quality high. The quality of these kind of systems is normally not only measured by functional correctness but also by timing behaviour correctness. Because timing correctness is so essential when developing such systems, the implementation cost can increase considerably if it is discovered at later stages of the development that the selected hardware architecture cannot fulfil the timing parameters. Typical design questions that cross an architect's mind are [13]:

1. Does the proposed architecture meet the performance requirements of all applications?
2. How robust is the chosen architecture with respect to changes in the application or architecture parameters?
3. Is it possible to replace components by cheaper, less powerful equivalents to save cost while maintaining the required performance targets?

Models of software/hardware can be used to assist the system architect in answering these questions. They have previously been used to explore and validate different deployment architectures even before the implementation cycle starts [14,12]. By doing so, it is possible to gain knowledge, at an early stage, of the product that is being developed, even before any deployment decisions have been made. Usually, companies that wish to develop a distributed embedded system have certain target Printed Circuit Boards (PCBs) in mind, which support a small limited range of CPUs. These PCBs establish the architecture of the hardware and by using models and simulation techniques,

one can gain insight into which PCB should be selected in order to fulfil the project requirements. Furthermore, one can identify which kinds of CPUs and buses are needed to respect the speed/capacity requirements of the application. Having a method and tool support to test out all the interesting PCB/CPU/bus combinations and identify which ones satisfy the system timing invariants, would give the system architect an advantage when making such initial design choices.

The modelling language VDM-RT, enables a system architect to do these kinds of different simulations, but the process of changing the system architecture and application deployment is very cumbersome and lacks both flexibility and tool support. Everything concerning deployment is tightly connected and mixed with the application construction in the *system*[1] class. This makes it difficult and impractical to explore different architectures from a modeller's point of view. When one wants to make such changes, one must either overwrite the **`system`** class, and by doing so losing the previous system, or create a new project and copy all the files, except the system class, and then create a new system class.

In this paper we show how deployment of VDM-RT models can be modified to support exploration of different hardware configurations without changing the model and how this can help a system architect to find the good designs. This can be seen as a part of a larger effort in order to explore the different design alternatives of an embedded distributed system in the style used in the DESTECS project [2]. We consider "the best design" to be any solution that solves the proposed problem. It is then up to the architect to decide which is the best one based on system invariants and additional cost analysis.

The remainder of this paper is set out as follows. In section 2 an introduction to VDM and the VDM-RT dialect is presented. Section 3 illustrates how we separate the model from its deployment without loosing expressiveness. In section 4, the typical design questions are addressed and solutions are presented to show how this work assists the answering of these questions. Section 5 illustrates how this work can be used to explore an in-car-navigation system. Lastly, section 6 concludes this work with remarks and suggestions for future improvements.

2 The VDM Real-Time Dialect

The Vienna Development Method (VDM) [1,8,4] was originally developed at the IBM laboratories in Vienna in the 1970s and as such it is one of the longest established formal methods. The VDM Specification Language is a language with a formally defined syntax, static and dynamic semantics. Models in VDM are based on data type definitions built from simple abstract types such as **`bool`**, **`nat`** and **`char`** and type constructors that allow user-defined product and union types and collection types such as (finite) sets, sequences and mappings. Type membership may be restricted by predicate invariants. Persistent state is defined by means of typed variables, again restricted by invariants. Operations that may modify the state can be defined implicitly, using pre- and post-condition predicates, or explicitly, using imperative statements. Such operations denote relations between inputs and pre-states and outputs and post-states, allowing for non-

[1] The **`system`** class describes the system architecture and its deployment.

determinism. Functions are defined in a similar way to operations, but may not refer to state variables.

Three different dialects exists for VDM: The ISO standard VDM Specification Language (VDM-SL) [5], the object oriented extension VDM++ [6] and a further extension of that called VDM Real Time (VDM-RT) [15,7]. All three dialects are supported by the open source tool called Overture [9].

VDM++ and VDM-RT allow concurrent *threads* to be defined. In VDM-RT, the concurrency modelling can be enhanced by deploying objects on different CPUs with buses connecting them. Operations called between CPUs can be asynchronous, so that the caller does not wait for the call to complete.

VDM-RT has a special **`system`** class where the modeller can specify the hardware architecture, including the CPUs and their bus communication topology; the dialect provides two predefined classes for the purpose, `CPU` and `BUS`. CPUs are instantiated with a clock speed (Hz) and a *scheduling policy*, either *First-come, first-served (FCFS)* or *Fixed priority (FP)*. Only one **`system`** is allowed to be declared at a time for a single model.

The initial objects (artifacts) defined in the model can then be deployed to the declared CPUs using the CPU's `deploy` operations. Buses are defined with a transmission speed (bytes/s) and a set of CPUs which they connect. Object instances that are not deployed to a specific CPU (and not created by an object that is deployed), are automatically deployed onto a *virtual CPU*. The virtual CPU is connected to all real CPUs through a *virtual bus*. Virtual components are used to simulate the external environment for the model of the system being developed.

In figure 1 a graphical representation of an in-car navigation radio system is shown, which illustrates deployment with three CPUs connected by a single bus.

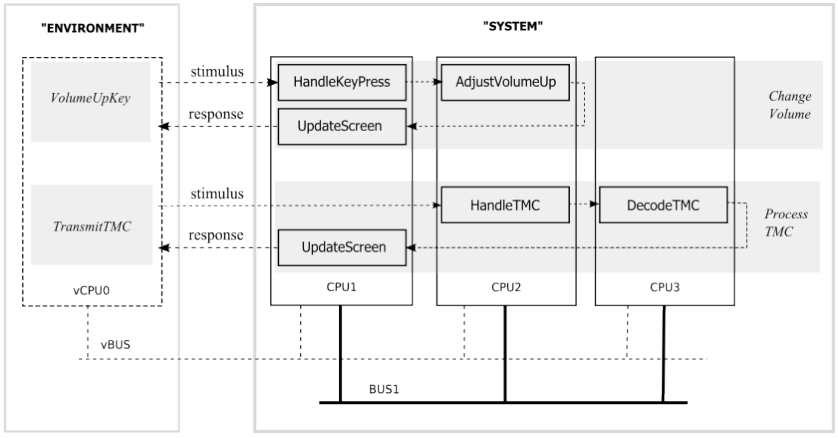

Fig. 1. Overview of the In-car Radio Navigation System

The in-car-navigation system shown in figure 1 is represented as a **`system`** class in listing 1.1. Firstly, the application artifacts are declared as instance variables (`mmi`,

radio and navigation). The definition of the hardware appears after: in this case three CPUs are declared (CPU1, CPU2 and CPU3) with a bus (BUS1) which connects them. Finally, the system architecture and deployment of the artifacts to the specific CPUs appear in the last section.

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

  -- create CPUs (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
  public RadNavSys: () ==> RadNavSys
  RadNavSys () ==
    ( navigation.setMmi(mmi);
      radio.setMmi(mmi);
      radio.setNavigation(navigation);
      mmi.setRadio(radio);

      CPU1.deploy(mmi,"MMI");
      CPU2.deploy(radio,"Radio");
      CPU3.deploy(navigation,"Nav");
      ...
    );
end RadNavSys
```

Listing 1.1. A typical **system** class

Special system invariants based on timing constraints can be validated through post analysis of log files [3]. This enables the modeller to express time constraints on operations and instance variables; e.g. when volumeUp is called then no later than three time units later the volume must be incremented. Log files used for post analysis can be directly created by the VDM interpreter [10] enabling automated checking of such time constraints, allowing a systematic rejection of models which do not meet the time constraints either because the models are wrongly specified or the architecture used is not powerful enough. At the this point in time only post analysis is possible but a version to check system invariants at run-time is being investigated [11].

3 Ensuring Separation Between Software And Hardware

To enable automated exploration of hardware architectures for VDM-RT models, changes must be made to the way deployment is expressed. Currently, the modeller must create new projects with a custom **system** class for each architecture to be explored. This method is inefficient and difficult to automate. The basic problem with the current system definition is the close coupling between system architecture and system deployment. Ensuring a clear separation between architecture and deployment allows a system to be configured and tested against any number of hardware architectures without the hassle of creating new test projects or changing the system architecture.

This section will present a different approach to express deployment from the current VDM-RT **system** class explained in section 2 while preserving the same run-time properties.

This section will present a different approach to express deployment from the current VDM-RT **system** class shown in section 2 while preserving the same run-time properties. Instead of a single **system** class defining the deployment, our approach uses a four level structure to define deployment, keeping a clear separation between the model and the actual deployment. This allows tool automated exploration at all levels:

Abstract Software Architecture: Defines artifacts and how they depend on each other;
Abstract Hardware Architecture: Defines the abstract hardware architecture in terms of nodes and communication channels, i.e. without speeds/capacities or policies;
Configuration: Defines deployment of artifacts presented in Abstract Software Architecture to nodes from the hardware present in the Abstract Hardware Architecture;
Deployment: Defines a concrete deployment using the Configuration; similar to the constructor in the current **system** class.

For each of the levels above, a concrete definition and the relation between them will be presented as VDM-SL types and functions in the following sections.

3.1 Abstract Software Architecture

The Abstract Software Architecture (ASA) is used to describe which application artifacts exist in the system and where inter-artifact calls occur. It represents the software system at its most abstract point where only artifacts of applications are referred e.g. `mmi`, `radio` and `navigation` from section 2. The ASA contains dependencies between the different artifacts representing the inter-artifact calls in the system. This dependency description is used both (1) to check that a hardware architecture contains the required communication channels and (2) for automatic exploration of hardware architectures fitting the software model. Listing 1.2 shows the VDM types used to represent the ASA of a system.

```
types
Artifact : seq of char

ASA ::
```

```
  artifacts    : set of Artifact
  dependencies : map Artifact to set of Artifact
inv mk_ASA(artifacts,dependencies) ==
  dom dependencies subset artifacts
  and
  dunion rng dependencies subset artifacts
  and
  forall key in set dom dependencies &
        key not in set dependencies(key);
```

Listing 1.2. Abstract Software Architecture types.

The `Artifact` type denotes a named system instance variables (e.g. `mmi`); the `artifacts` set denotes the set of artifacts which can be deployed; the `dependencies` map denotes the dependencies between the artifacts.

3.2 Abstract Hardware Architecture

From an abstract point of view, a computing system is no more than a set of processing nodes which communicate via channels. We name this representation: Abstract Hardware Architecture (AHA). Listing 1.3 presents VDM types capable of representing an abstract hardware architecture.

```
types
Node ::
  id : nat1;

ComChannel ::
  nodes : set of Node;

AHA ::
  nodes    : set of Node
  channels : set of ComChannels
inv forall c in set channels & c.nodes subset of nodes;
```

Listing 1.3. Abstract Hardware Architecture as a VDM type

A processing node is represented by `Node` which has an identifier and a communication channel is represented by `ComChannel`, which contains the set of nodes it connects. `AHA` defines a hardware architecture containing several nodes and channels connecting them. AHAs can either be automatically generated based on the maximum number of artifacts in the system or manually specified which is often the desired solution for an industry where existing PCBs are available from previous projects.

3.3 Configuration

A configuration describes how a system is deployed to an abstract architecture. This allows a system to be deployed onto a hardware configuration without explicitly spec-

ifying the limitations of the hardware like CPU speed and bus capacity. A configuration defines a relation between artifacts from an ASA and the computing nodes from an AHA. The dependencies stated by an ASA must be reflected in the communication channels of the AHA for the configuration to be valid. This check is done by the function `checkDependencies`. Listing 1.4 defines a configuration of an ASA to an AHA. A configuration can be created either by automatic permutation of artifacts onto the nodes of an AHA or by manually specifying the relations. The latter is the normal case for an industry where specialized nodes such as processors with integrated GPS[2] modules are used, which will require a GPS artifact to be explicitly deployed to a specific node.

```
types
NodeArtifactRelation : map Node to set of Artifact;

Configuration ::
  asa : ASA
  aha : AHA
  relation : NodeArtifactRelation
inv mk_Configuration(asa,aha,relation) ==
  checkDependencies(asa, aha, relation);
```

Listing 1.4. Deployment Configuration

3.4 Deployment

The deployment of a system is the process of restricting the computational power of the nodes and the communication channels. A node must be limited to the computational power of a specific CPU with a maximum number of instructions it can perform per second. The same applies to buses where the transfer rate is limited. Listing 1.5 shows the `Deployment` type which represents a mapping between `Nodes` and `ComChannels` to concrete CPUs and buses.

```
Deployment ::
  config : Configuration
  buses  : map ComChannel to BUS
  cpus   : map Node to CPU
inv mk_Deployment(config,buses,cpus) ==
  (forall channel in set config.aha.channels &
        channel in set dom buses)
  and
  (forall node in set config.aha.nodes & node in set dom cpus)
  and card config.aha.channels = card dom buses
  and card config.aha.nodes = card dom cpus;
```

Listing 1.5. Specifies the type of each computational node and communication channel

[2] Global Positioning System

Computational nodes and communication channels are abstractions of the actual physical implementation where a circuit board is manufactured, which among other things consists of the main components CPUs and buses which VDM-RT can reason about. In listing 1.6 two VDM types are listed. `CPU` represents a computational `Node` where the node is limited from being infinitely fast to a specific frequency slowing down the execution of instructions. The same applies to the `BUS`, which is a limited version of the `ComChannel`, where a transmission speed limits the number of bytes which can be transmitted per second.

```
CPU ::
  id                  : nat1
  speed               : nat1
  brand               : seq of char
  scheduling          : <FP> | <FCFS>;

BUS ::
  id              : nat1
  speed           : nat1
  type            : <FILO>;
```

Listing 1.6. Hardware types

3.5 New Deployment Work-flow

To use the four layered separation described above some changes must be made to the deployment work flow. However, not all of the above levels require the modeller's direct attention, since most of the changes are conceptual separations of system elements. It is important to understand that the output of the separation proposed above can be mapped to the current VDM-RT system class without losing details. The difference is that this clear separation between the different levels, enables the modeller to do exploration at all levels. It also enables tools to be developed to assist this process.

The work-flow in ordinary VDM-RT can be described with the following steps:

1. Defining the VDM-RT model.
2. Identifying the static artifacts of the model.
3. Defining the hardware nodes: `CPU` and `BUS` and instantiating the artifacts.
4. Deploying the artifacts to the CPUs.

This is currently all done in a single class called **`system`** with no clear indication of what is artifacts and what is hardware and deployment.

The work-flow with the new sub divided structure:

Model development: The first step is to develop the actual VDM model as in the current VDM-RT workflow.

System configuration: The modeller configures the artifacts of the system as usual in a VDM-RT system class.

Extract artifacts and dependencies: If all artifact relations are expressed as either artifact constructor arguments or parsed as arguments to operations on artifacts, then this step can be automated. Artifacts will be extracted from the **`system`** class and their dependencies from the system constructor, enabling an ASA to be created[3]:

Composing a new AHA: The ASA defines the artifacts and their required dependencies while the AHA define an abstract hardware architecture which respects the dependencies from the ASA extracted from the artifacts dependencies. Such an AHA can either be automatically generated based on the ASA or it can be manually specified by the user.

Configuration: The configuration defines how each artifact is linked to a node of the given AHA. This can be specified manually by the user or a range of configurations can be generated from the pair (ASA, AHA).

Deployment: The final deployment is the limitation of an AHA. This can again be specified by the user to a single fixed deployment or the user can enter a set of possible CPUs which could be used per node allowing a range of deployments to be generated to explore these different CPU limitations.

Evaluation: Finally, the model can be executed with a single specific deployment and its system invariants can be checked either through post-analysis or at run-time both leading to an accept / reject verdict of the tested deployment. This indicates to the modeller if this configuration is acceptable to the system leaving the decision of which to choose to the modeller.

The steps described above can be expressed through the formula 1.

$$\left(ASA + AHA\right) \rightarrow^{*} Configuration \rightarrow^{*} Deployment \equiv \texttt{system} \qquad (1)$$

The arrow $\rightarrow^{*}$ denotes that many elements can be generated with respect to the left side of the arrow. In the first case one or more *configurations* can exist which configures a particular pair of ASA and AHA. Each configuration defines how the ASA is mapped onto the AHA but does not restrict the hardware in any way. Similar to the *configuration*, one or more *deployments* can exist which restricts a particular *configuration* by limiting each computational node to a specific frequency and each communication channel to a specific speed. Finally it can be seen from the **`system`** in section 2 that the left side of the formula below is equivalent to the information in the system class in VDM-RT.

4 The Exploration of Alternative System Architectures

Exploring alternative system architectures is supported by VDM-RT and in section 3 it has been described how the process of deployment can be split up into levels which can be explored for alternatives. The goal is to provide the means to answer the questions

[3] In this paper we do not deal with references which can be passed between artifacts at run-time which also leads to new dependencies.

stated in the introduction. However because these questions are seen from the modellers point of view, we will try to relate them to the levels of the formula 1 to make it easier to describe how this work provides (partial) answers to these questions.

The requirements extracted from the questions are as follows:

- Exploring alternative artifact distribution on a fixed hardware configuration.
- Exploring alternative hardware configurations for an ASA.
- Exploring alternative deployment parameters for a fixed configuration.

The questions require the exploration to support different distribution of artifacts on a fixed distributed hardware platform; the ability to explore parameters for a specific hardware such as CPU capacity; and finally a way to validate such a system architecture. Futhermore we can add the ability to generate hardware architectures, but this may be mainly of academic value. The requirements stated above are covered in the following subsections. In addition the validation is addressed in section 4.4.

4.1 Exploring Alternative Artifact Distribution On A Fixed Hardware Configuration

To explore alternative artifact distribution, an ASA is required to obtain the artifacts and their dependencies. Since the hardware configuration is fixed, an AHA is also provided by the modeller. This gives the pair (ASA, AHA) as input to the exploration of alternative artifact distribution. Formula 2 illustrates where this takes place in the overall work flow where the underlined part denotes what is produced. The result of the generation of alternative distributions is a set of `Configuration`s all for the same system.

$$\left(ASA + AHA\right) \rightarrow^{*} \underline{Configuration} \rightarrow^{*} Deployment \equiv \textbf{system} \qquad (2)$$

Listing 1.7 shows the signature of a VDM function which produces the desired set of configurations:

```
createAltDisbs : ASA * AHA -> set of Configuration
createAltDisbs(asa, aha) == is not yet specified;
```

Listing 1.7. Signature of a function for generation of Configurations from an ASA.

4.2 Exploring Alternative Hardware Configurations For An ASA

When the goal is to find the optimal hardware configuration for a given system it can often be difficult and time consuming to create all possible combinations of nodes and communication channels. It is however important to understand that this is possibly only of academic value since industrial companies often have of-the-shelf hardware platforms which they want to explore. AHAs can be generated from the number of

unique artifacts from an ASA. The formula 3 shows where in the overall work flow this exploration contributes again using the underlined part as the produced aspects.

$$\left(ASA + \underline{AHA}\right) \rightarrow^* Configuration \rightarrow^* Deployment \equiv \textbf{system} \quad (3)$$

A signature of the VDM function to create the AHAs is shown in listing 1.8. It takes an ASA as input and returns a set of AHAs.

```
createAHAs : ASA -> set of AHA
createAHAs(asa) ==
  let maxNodes = card asa.artifacts
  in
    ...
```

Listing 1.8. Signature of a VDM function for the automatic AHA generation.

4.3 Exploring Alternative Deployment Parameters For A Fixed Configuration

Exploring alternative deployment parameters for an otherwise fixed system is one of the most important requirements because this relates directly to the costs of the final product. If a cheaper CPU can be used in mass production, money can be saved by the manufacturer of such a system. Exploring alternative deployment parameters means that one can come up with all possible limitations of the hardware, reducing either a CPUs computational capacity or limiting the bandwidth of a bus. An unlimited range of such deployments can be generated however this is not useful in practice since only a small number of CPUs and buses can be used in a specific hardware topology. In most cases, a PCB design already exists which supports a fixed number of different CPUs from a specific family. Thus the exploration is based on knowing that a small list of possible CPUs or buses are available to be used as nodes. The exploration generates a set of deployments and takes an otherwise fixed system as input together with a set of available CPUs per node of the AHA and a set of available buses for each communication channel. Formula 4 shows where in the overall work flow this takes place.

$$\left(ASA + AHA\right) \rightarrow^* Configuration \rightarrow^* \underline{Deployment} \equiv \textbf{system} \quad (4)$$

The signature of a VDM function is shown in listing 1.9 which takes a fixed configuration of a system plus two maps where the available CPUs and BUSs are given for the resources in the AHA.

```
exploreDeployParams : Configuration *
                      map Node to set of CPU *
```

```
                          map ComChannel to set of BUS
                          -> set of Deployment
exploreDeployParams(config, nCm, cBm) == is not yet specified;
```

Listing 1.9. Signature of a VDM function for alternative deployment parameter exploration.

4.4 Evaluation Of The Architectures

The ability to automatically determine if a specific deployment is good enough is very important now that we have presented the functionality to automatically generate alternatives as early as the AHA in the work flow. The potentially results is a very large number of deployments, since a split in the flow at an early stage doubles the output of all later steps. Currently, the only way to determine if a deployment is "good enough" is by manually inspecting the execution log through the graphical viewer named Real-Time Log Viewer. This viewer is able to illustrate how the scheduler creates threads, shifts them in and out in relation to time etc. To overcome the challenge of manual inspection work is being done in [11] to enable run-time checking of system invariants. Such invariants can then express time constraints in the system, which is exactly what is needed when deployments have to be validated. If the modeller provides system invariants expressing the critical time constraints of the model then the run-time checking of these invariants will be able to tell us if a given deployment has not violated any invariants and thus be accepted.

5 Case Study: In-car Radio Navigation

This case study is based on an already known case explored in both [3] and [13]. How this new structure can be used to do deployment exploration will be presented. The new way to express a system configuration is shown in listing 1.10, it can be seen that no deployment is included within the system class. This is very similar to the system from section 2.

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

operations
  public RadNavSys: () ==> RadNavSys
  RadNavSys () ==
    ( navigation.setMmi(mmi);
      radio.setMmi(mmi);
      radio.setNavigation(navigation);
      mmi.setRadio(radio);
```

```
    );
end RadNavSys
```

Listing 1.10. In-Car-Navigation system.

A new grammar for deployment in VDM-RT is proposed in listing 1.11, allowing the deployment elements: AHA, configuration and deployment to be specified. All the elements can be generated through exploration as explained in section 4. The listing 1.11 illustrates how the deployment of the in-car-navigation system can be done with this new syntax. The deployment is specified with all elements, but without the ASA, since it can automatically be extracted from the system class in listing 1.10. Any of the blocks **aha**, **configuration** and **deployment** can be left empty in the grammar, indicating that they should be automatically generated. However by explicitly specifying all blocks only a single deployment will exist as in the original system definition from section 2.

```
aha

Channel1 := {node1, node2, node3}

configuration

node1 := {mmi};
node2 := {radio}
node3 := {navigation}

deployment

node1 := CPU(200MHz, <FP>)
node2 := CPU(100MHz, <FP>)
node3 := CPU(1000MHz, <FP>)
Channel1 := BUS(72E3, <CSMACD>)
```

Listing 1.11. New deployment specification for the In-Car-Navigation system

What if the deployment specified in listing 1.11 is an acceptable deployment but the modeller likes to do future investigation through the third question: *Is it possible to replace components by cheaper, less powerful, equivalents to save cost while maintaining the required performance targets?* One option is to try out deployments where one of the nodes is limited to one of three different CPUs as shown in listing 1.12. It can be seen that the grammar allows nodes to be defined with a set of `CPU`s instead of a single `CPU` this allows the exploration to use permutations of CPUs for each node.

```
deployment

node1 := {CPU(200MHz, <FP>),
          CPU(100MHz, <FP>),
```

```
          CPU(50MHz, <FP>)}
node2 := CPU(100MHz, <FP>)
node3 := CPU(1000MHz, <FP>)
Channel1 := BUS(72E3, <FCFS>)
```

Listing 1.12. Alternative deployment block for exploration of deployment parameters.

When the exploration is done for the **`deployment`** block as shown in listing 1.12, three alternatives will be generated, one with each type of CPU. All these alternatives can then automatically be validated against the same tests to see if all of them fulfils the system invariant[4]. If so the modeller can freely decide which option is the best choice.

6 Concluding Remarks

Choosing the optimal architecture for a system is challenging, not only can it be difficult to determine but VDM-RT currently lacks the ability to allow exploration of alternatives in an efficient way without the need of duplicating the model. This work has proposed a way to enable exploration through separation of model and deployment, where exploration is possible at all levels of the deployment process. The common questions a modeller might ask when choosing a optimal architecture have been addressed and exploration functions proposed. We think that this work will help the system architect to determine an optimal architecture for a given system by enabling easy automated exploration. Such an exploration will be able to create all alternatives of AHA, Configuration and deployments and evaluate them against system invariants. VDM-RT priority settings for functions and operations has not been addressed in this work but will be future investigated in the near future.

The plan is to implement the features described in this paper in the Overture platform such that it can be exploited for automatic co-model analysis in the DESTECS project as well. We expect that this will be completed before the end of 2011.

Acknowledgements

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

References

1. Bjørner, D., Jones, C. (eds.): The Vienna Development Method: The Meta-Language, Lecture Notes in Computer Science, vol. 61. Springer-Verlag (1978)
2. Broenink, J.F., Larsen, P.G., Verhoef, M., Kleijn, C., Jovanovic, D., Pierce, K., F., W.: Design support and tooling for dependable embedded control software. In: Proceedings of Serene 2010 International Workshop on Software Engineering for Resilient Systems. ACM (April 2010)

[4] The system invariants are not included in this paper but can be found in [3].

3. Fitzgerald, J.S., Larsen, P.G., Tjell, S., Verhoef, M.: Validation Support for Real-Time Embedded Systems in VDM++. In: Cukic, B., Dong, J. (eds.) Proc. HASE 2007: 10th IEEE High Assurance Systems Engineering Symposium. pp. 331–340. IEEE (November 2007)
4. Fitzgerald, J.S., Larsen, P.G., Verhoef, M.: Vienna Development Method. Wiley Encyclopedia of Computer Science and Engineering (2008), edited by Benjamin Wah, John Wiley & Sons, Inc
5. Fitzgerald, J., Larsen, P.G.: Modelling Systems – Practical Tools and Techniques in Software Development. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, Second edn. (2009), ISBN 0-521-62348-0
6. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object–oriented Systems. Springer, New York (2005), `http://www.vdmbook.com`
7. Hooman, J., Verhoef, M.: Formal semantics of a VDM extension for distributed embedded systems. In: Dams, D., Hannemann, U., Steffen, M. (eds.) Concurrency, Compositionality, and Correctness, Essays in Honor of Willem-Paul de Roever. Lecture notes in Computer Science, vol. 5930, pp. 142–161. Springer-Verlag (2010)
8. Jones, C.B.: Systematic Software Development Using VDM. Prentice-Hall International, Englewood Cliffs, New Jersey, second edn. (1990), iSBN 0-13-880733-7
9. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The Overture Initiative – Integrating Tools for VDM. ACM Software Engineering Notes 35(1) (January 2010)
10. Lausdahl, K., Larsen, P.G., Battle, N.: A Deterministic Interpreter Simulating a Distributed Real Time System using VDM. Submitted for publication (2011)
11. Ribeiro, A., Lausdahl, K., Larsen, P.G.: Run-Time Validation of Timing Constraints for VDM-RT Models. Submitted for publication (2011)
12. 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. pp. 26–43. School of Computing Science, Newcastle University, Technical Report CS-TR-969 (June 2006)
13. Verhoef, M.: Modeling and Validating Distributed Embedded Real-Time Control Systems. Ph.D. thesis, Radboud University Nijmegen (2008), ISBN 978-90-9023705-3
14. Verhoef, M., Larsen, P.G.: Interpreting Distributed System Architectures Using VDM++ – A Case Study. In: Sauser, B., Muller, G. (eds.) 5th Annual Conference on Systems Engineering Research (March 2007), Available at http://www.stevens.edu/engineering/cser/
15. Verhoef, M., Larsen, P.G., Hooman, J.: Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006: Formal Methods. pp. 147–162. Lecture Notes in Computer Science 4085 (2006)

Facilitating Consistency Check between Specification and Implementation with MapReduce Framework

Shigeru KUSAKABE, Yoichi OMORI, and Keijiro ARAKI

Grad. School of Information Science and Electrical Engineering, Kyushu University
744, Motooka, Nishi-ku, Fukuoka city, 819-0395, Japan

Abstract. We often need well-formed specifications in order to properly maintain or extend a system by members who were not in charge of the original development. In contrast to our expectation, formal specifications and related documents may not be maintained, or not developed in real projects. We are trying to build a framework to develop specifications from a working implementation. Testability of specifications is important in our framework, and we develop executable, or testable, formal specifications in model-oriented formal specification languages such as VDM-SL. We figure out a formal specification, check it with the corresponding implementation by testing, and modify it if necessary. While the specific level of rigor depends on the aim of the project, millions of tests may be performed in developing highly reliable specifications. In this paper, we discuss our approach to reducing the cost of specification test. We use Hadoop, which is an implementation of the MapReduce framework, so that we can expect the scalability in testing specifications. We can automatically distribute the generation of test cases from a property, the interpretation of the executable specification and the execution of its corresponding implementation code for each test data using Hadoop. While straightforward sequential execution for large data set is expensive, we observed scalability in the performance in our approaches.

1 Introduction

While we are supposed to have adequate documents in ideally disciplined projects, we may not have such documents in many actual projects. In spite of potential effectiveness of formal methods, formal specifications and related documents may not be maintained, or not developed. Nonetheless, we often need well-formed specifications in order to properly maintain or extend a system by members who were not familiar with the details of current implementation.

We are trying to build a framework to develop specifications for maintenance from the actual code of working implementation plus ill-maintained specifications for development if they exist. Fig. 1 shows our concept.

We expect we can figure out some specifications by using techniques of software engineering while we assume documents used in the development may be unreliable and members familiar with the detail of the development may be unavailable. However, we will rely on testing to check the consistency between the implementation and the specifications in the process of making the specifications close to ideal ones.

We can develop executable, or testable, formal specifications in model-oriented formal specification languages such as VDM-SL. By using the interpretor of VDMTools,

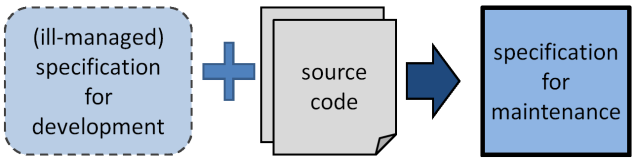

Fig. 1. Developing specifications for maintaining or extending a system from source code plus specifications that are not well managed during the development.

we can test executable specifications in VDM languages to increase our confidence in the specifications. In our framework, we expect executable specifications to play an important role. We develop a formal specification from the running implementation, check it with the result or behavior of the corresponding implementation by testing, and modify and retest it if necessary. In contrast to the usual software development, we modify the specifications to be consistent with the corresponding running code.

In this paper, we discuss our approach to reducing the cost of testing specifications. In the testing phase, millions of tests may be performed in developing highly reliable specifications, while the specific level of rigor depends on the background of the project. Our approach is a brute-force one which adopts recent cloud computing technologies. We use Hadoop, which is an implementation of the MapReduce framework, so that we can expect the scalability in testing specifications. We can automatically distribute the generation of test data from a property, the interpretation of the executable specification and the execution of its corresponding implementation code for each test data using Hadoop. While straightforward sequential execution for large volume of test data is expensive, our preliminary evaluation indicates that we can expect scalability of our approach.

The rest of this paper is organized as follows. We explain our approach to reduce the cost of testing for large data set by using emerging cloud technology in section 2. We outline our framework in section 3. We evaluate our testing framework in section 4. Finally we conclude in section 5.

2 Approach with Cloud Technology

2.1 Elastic platform

Our approach shares the issues of testing with that of usual software development. As the size and complexity of software increase, its test suite becomes larger and its execution time becomes a problem in software development.

Large software projects may have large test suites. There are industry reports showing that a complete regression test session of thousands lines of software could take weeks of continuous execution [5]. If each test is independent with each other, high level of parallelism provided by a computational grid can be used to speed up the test execution [4]. Distributing tests over a set of machines aims at speeding up the test stage by executing tests in parallel [7].

We consider an approach to leveraging the power of testing by using elastic cloud platforms to perform large scale testing. Increasing the number of tests can be effective in obtaining higher confidence, and increasing the number of machines can be effective in reducing the testing time.

The cloud computing paradigm seems to bring a lot of changes to many fields. We believe it also has impact on the field of software engineering and consider an approach to leveraging light-weight formal methods by using cloud computing which has the following aspects [1]:

1. The illusion of infinite computing resources available on demand, thereby eliminating the need for cloud computing users to plan far ahead for provisioning;
2. The elimination of an up-front commitment by cloud users, thereby allowing organizations to start small and increase hardware resources only when there is an increase in their needs; and
3. The ability to pay for use of computing resources on a short-term basis as needed and release them as needed, thereby rewarding conservation by letting machines and storage go when they are no longer useful.

We can prepare a platform of arbitrary number of machines and desired configuration depending on the needs of the project.

2.2 MapReduce

While we can prepare a platform of an arbitrary number of computing nodes and prepare an arbitrary number of test cases, we need to reduce the cost of managing and administrating of the platform and runtime environment.

The MapReduce programming model was created in order to generate and process large data sets on a cluster of machines [3]. Programs are written in a functional style, in which we specify mapper functions and reducer functions, as in `map` and `reduce` (or `fold`) in functional programming language. Fig. 2 shows the concept of `map` and `reduce`. For example, when we calculate square-sum of the elements in a sequence (list in typical functional programming languages), we specify a square function as the function f_M, an add as the function f_R, and 0 as the initial value init. In `map`, the function f_M, square, is applied to every element in the sequence, and in `reduce`, squared values are reduced to a single value by using f_R and init. In MapReduce programming framework, input data set is split into independent elements, and each mapper task processes each independent element in a parallel manner. Data elements are typically data chunks when processing a huge volume of data. The outputs of the mappers are sorted and sent to the reducer tasks as their inputs. The combination of map/reduce phase has flexibility, thus, for example, we can align multiple map phases in front of a reduce phase.

MapReduce programs are automatically parallelized and executed on a large cluster of machines. The runtime system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. Its implementation allows programmers to easily utilize the resources of a large distributed system without expert skills in parallel and distributed systems.

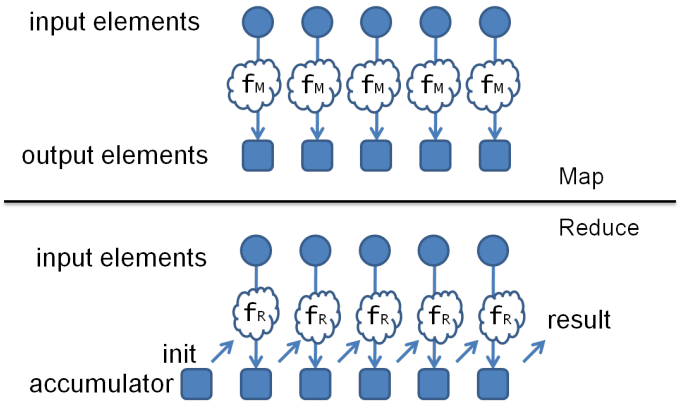

Fig. 2. Concept of the MapReduce programming model.

When using this MapReduce framework, input elements are test data, `f` can be an executable specification in VDM or actual code fragment under test, and output elements are test results.

3 Testing Framework

In this section, we discuss our testing framework, which uses Hadoop to reduce the cost of testing with a large data set. Hadoop is an open source software framework implementing MapReduce programming model [6] written in Java. While our framework can be adjusted to testing in a typical software development, we focus on testing specifications, which are figured out based on a corresponding implementation.

3.1 Property-based testing

Fig. 3 shows the outline of our testing framework. First, we generate test data according to the specified property. We use a property-based testing tool, QuickCheck [2], which supports a high-level approach to testing Haskell programs by automatically generating random input data. QuickCheck defines a formal specification language to state properties, and we can customize test case generation of QuickCheck including the number of test cases. We modify QuickCheck to fit to our approach for testing formal specifications with Hadoop. We try to automatically distribute the generation of test data for a formal specification in addition to the execution of the formal specification. We store the generated data in a file on the Hadoop file system. In the evaluation phase, we pass the test data to mappers, each mapper execute the executable specification and its corresponding implementation code for each test data, and then outputs the comparison result. Reducers receive and aggregate the comparison results.

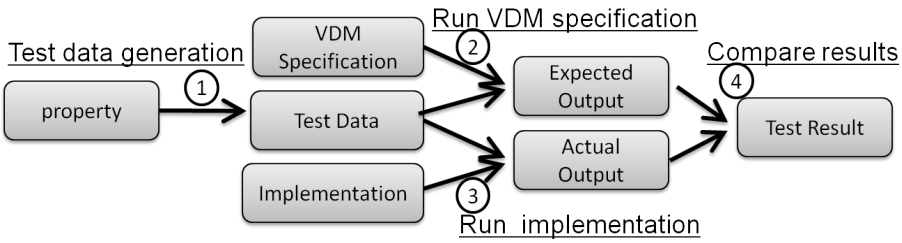

Fig. 3. Outline of our approach to property-based testing.

Table 1. Configuration of the platform

	NameNode	JobTracker	Slave
CPU	Xeon E5420	Xeon E5420	Xeon X3320
	2.50GHz 4core	2.50GHz 4core	2.50GHz 4core
Memory	3.2GB	8.0GB	3.2GB
Disk	2TB	1TB	140GB

3.2 Hadoop Streaming

In the Hadoop framework, we write mapper and reducer functions in Java by default. However, the Hadoop distribution contains a utility, Hadoop Streaming, which allows us to create and run jobs with any executable or script that uses standard input/output as the mapper and/or the reducer. The utility will create a mapper/reducer job, submit the job to an appropriate cluster, and monitor the progress of the job until it completes. When an executable is specified for mappers, each mapper task will launch the executable as a separate process when the mapper is initialized. When an executable is specified for reducers, each reducer task will launch the executable as a separate process then the reducer is initialized. This Hadoop Streaming is useful in implementing our testing framework. We execute specifications in VDM with the combination of the command-line interface of VDMtools and the mechanism of Hadoop Streaming.

4 Performance evaluation

4.1 Evaluation of a formal specification for a large data set

In order to examine the effectiveness of our testing framework using Hadoop, we measured performance in testing a specification of the Enigma machine given in [8] for a large data set. The Enigma cipher machine is basically a typewriter composed of three parts: the keyboard to enter the plain text, the encryption device and a display to show the cipher text. Both the keyboard and the display consist of 26 elements, one for each letter in the alphabet.

The configuration of the platform is shown in Table 1. We show the result of elapsed time in Fig. 4. As we can see from the results, the elapsed time of the Hadoop version

reduced when the number of tests was over four hundreds. Since the Hadoop framework is designed for large scale data processing, we see no advantage in elapsed time for small set of test data. Each computation node has four processor cores, and we can achieve speedup even on a single node as Hadoop can exploit thread-level parallelism on multi-core platforms.

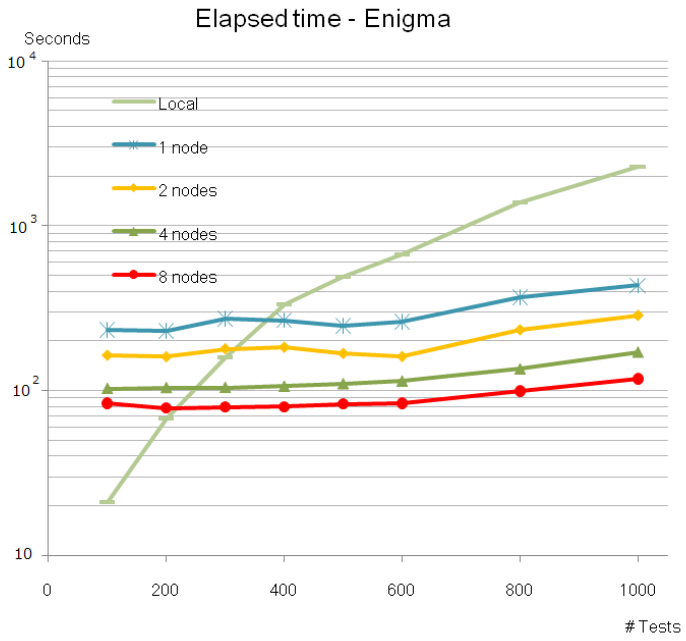

Fig. 4. Elapsed time for VDM enigma specification in increasing the number of tests on the various number of nodes.

4.2 Evaluation of our testing framework for large data set

In order to examine the effectiveness of our approach, we measured elapsed time in testing with an implementation of a small address book system and the corresponding specification in VDM on Hadoop, for a variable number of test cases. The property we used is idempotency, which is an invariant to check if the sorting operation obeys the basic rule: applying sort twice has the same result as applying it only once.

We show the result in Fig. 5. As we see in Fig. 5, until the number of tests exceeds about 300, the total elapsed time of the Hadoop version is more than that of non-Hadoop version while the former uses eight nodes and the latter uses only a single local node. The results of Hadoop version include some overheads such as distributing test data and collecting evaluation results over the network. After that point, the gap between the two

version becomes wider (i.e. the Hadoop version becomes faster) as the number of test data increases. Thus, our approach is suitable for a large scale test data set.

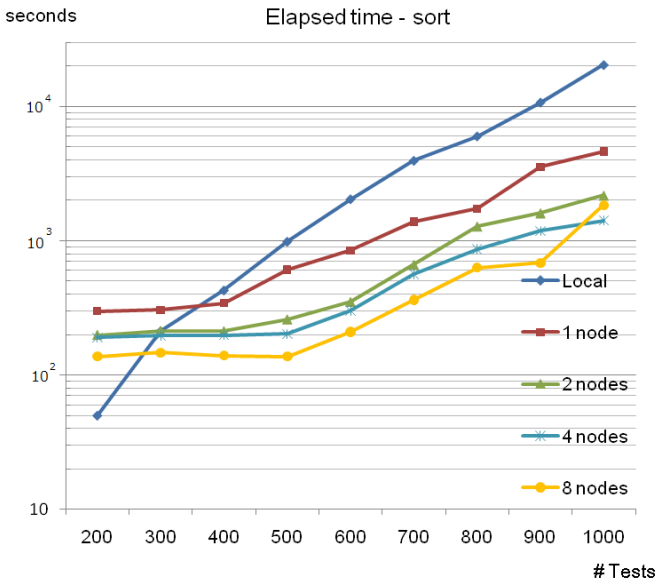

Fig. 5. Elapsed time in increasing the number of test data for idempotency on the various number of nodes.

We show the result on speedup in Fig. 6. The speedup ratio is calculated as *(time for N Hadoop nodes) / (time for local single node)*. As we see in Fig. 6, the increase of the number of slave machines is generally effective in reducing testing time. However, the speedup ratio against the number of slaves does not seem ideal. There are some troughs in the graph. While one of the reasons is overhead of using Hadoop, we will investigate further to achieve more efficient environment.

5 Concluding Remarks

In this paper, we presented preliminary evaluation results of our approach to reducing the cost of testing executable specifications for a large data set using Hadoop. Testability of a specification helps us increase our confidence in the specification. We are trying to develop executable formal specifications while we assume documents used in the development may be unreliable and members familiar with the detail of the development may be unavailable. We rely on testing to check the consistency between the implementation and the specifications. In order to increase our confidence in the specification, we

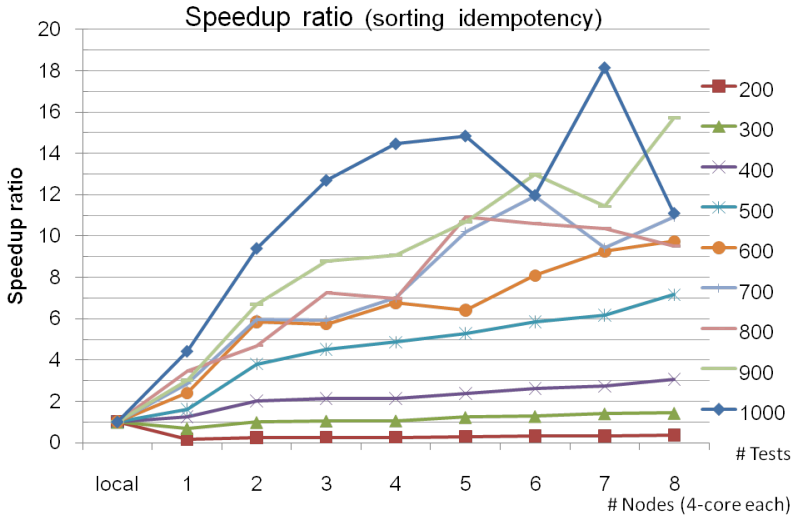

Fig. 6. Speedup in increasing the number of test data for idempotency on the various number of nodes.

can increase the number of test cases on elastic computing platforms at reasonable cost. We are able to automatically distribute the interpretation of the executable specification and the execution of its corresponding implementation code for each test data by using Hadoop. While straightforward sequential execution for large data set is expensive, we observed scalability in the performance in our approaches.

As one avenue of future works, we will investigate a more detailed performance breakdown to achieve more efficient environment. We will also try to extend usability of our framework. VDMTools and other programming language systems include test coverage statistics tools. We will extend our framework to exploit these tools in a parallel and distributed way to examine the impact of our approach on increasing test coverage.

References

1. Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical report, UCB/EECS-2009-28, Reliable Adaptive Distributed Systems Laboratory, February 2009.
2. Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *ACM SIGPLAN Notices*, 35(9):268–279, 2000.
3. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.
4. A. DUARTE, W. CIRNE FILHO, F. V. BRASILEIRO, and P. D. L. MACHADO. Gridunit: Software testing on the grid. In *Proceedings of the 28th ACM/IEEE International Conference on Software Engineering*, volume 28, pages 779 – 782. ACM, 2006.

5. Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Prioritizing test cases for regression testing. In *In Proceedings of the International Symposium on Software Testing and Analysis*, pages 102–112. ACM Press, 2000.
6. Hadoop. As of Jan.1, 2011. http://hadoop.apache.org/.
7. G. M. Kapfhammer. Automatically and transparently distributing the execution of regression test suites. In *Proceedings of the 18th International Conference on Testing Computer Software*, 2001.
8. Peter Gorm Larsen, Paul Mukherjee, Nico Plat, Marcel Verhoef, and John Fitzgerald. *Validated Designs For Object-oriented Systems*. Springer Verlag, 1998.

Counterpoint: Towards a Proof-Support Tool for VDM

Ken Pierce

School of Computing Science, Newcastle University,
Newcastle upon Tyne, NE1 7RU, United Kingdom.
K.G.Pierce@ncl.ac.uk

Abstract. This paper is a position paper that presents ideas for an extension to the Overture tool platform that will support the process of proof in the VDM family of formal languages. The intention of the paper is to garner interest in building this extension and to promote discussion of a development road map. While creation of formal specifications in VDM is currently supported by two robust tools —the commercial VDMTools and the open-source Overture tool— these tools focus on execution of specifications and automated testing. This new extension to Overture will focus on support for proof and act as both a counterpart and counterpoint to the existing tools. Hence the name of the extension will be *Counterpoint*. Counterpoint will extend the Overture tool, which is built using the Eclipse framework. It will support the management of discharging a set of proof obligations generated from VDM specifications. It will support both hand-crafted proofs in the natural deduction style and mechanization of proofs through external tools.

1 Introduction

VDM (Vienna Development Method) is a mature and widely-used formal method. It is model based, meaning that specifications in VDM are explicit models of the systems which they represent, with a central notion of state and operations. Operations can be defined both implicitly in terms of pre- and post-conditions and explicitly using constructs familiar to programmers.

VDM is supported by two robust tools: the commercial VDMTools[1] and the open-source Overture tool[2]. These tools can syntax check VDM specifications and include static type checkers. Both tools can also execute a subset of VDM specifications (those with explicit function definitions) and perform dynamic checking at run-time. Single runs can be initiated by the tool user, as well as automated execution of a large number of test cases.

The expressiveness of VDM means however that static type checking is in general undecidable [DHB91]. For example, the consistency of functions with pre-conditions and the satisfiability of implicit functions cannot be checked statically [Ber97]. Where static checks cannot be performed, proof obligations can be generated [Ber97]. These proof obligations must be discharged by hand in order to show that a model is consistent. The generation of these obligations is supported by both VDMTools and Overture, however further tool support for proof is minimal at best.

[1] http://www.vdmtools.jp/en/
[2] http://www.overturetool.org/

Central to the idea of the original VDM-SL[3] language is the notion of refinement [Jon90]. In refinement, abstract specifications are shown to be refined (or reified in VDM terminology) by more concrete specifications. The aim being to eventually reach a concrete specification that can be realised in a programming language, with (relative) correctness to the original abstract specification being preserved by the reification chain.

Typically, this process involves data reification (finding concrete representations for abstract data types such as sets) and operation decomposition (demonstrating that the behaviour of abstract operations is realised by one or more concrete operations) [Jon90]. The correctness of reification is demonstrated by proof.

The author of this paper was first introduced to VDM over seven years ago, during his undergraduate degree. This included a few lectures and a piece of coursework on proof. Later, the author's thesis [Pie09] addressed the correctness of Simpson's four-slot mechanism [Sim90] for asynchronous communication using VDM.

The author also collaborated with colleagues from Newcastle University in an effort to use VDM to verify the "Mondex" electronic purse challenge problem[4] of Grand Challenge 6 [Woo06,SCW00]. This work included the production and checking of a large number of proofs by hand. The team agreed that while it was an excellent learning exercise, undertaking the proof task by hand was a challenge they would not wish to repeat again soon!

The main point of this author's introduction is to assert that proof is (and always has been) central to world of VDM and that the push for industrial adoption and tool-support has drawn focus away from this key aspect. This paper is intended to be the next (albeit initially small) step on the road to redressing that balance. The end of this road will see many new features available in the Overture tool, which —importantly— should compliment the current feature set and be of great benefit to the VDM community. The author has chosen to name this extension to the Overture tool "*Counterpoint*".

While the choice of another musical term may seem a little tongue-in-cheek, the name should hopefully evoke the idea of two seemingly different parts in a musical piece working together to form a richer whole — as execution and testing can work together with proof in verification. A second (older) meaning to the term, that of a counter argument, evokes the notion of proof. Finally, the name is a simple English word that rolls off the tongue easily.

In the remainder of this paper, Section 2 proposes a set of features for Counterpoint, Section 3 reiterates the aims of the Overture initiative and finally Section 4 draws conclusions and provides a look at the way ahead.

2 Counterpoint: a Proof-Support Tool for VDM

Counterpoint will be an extension to the Overture tool. It will provide a *Proof* perspective to complement the current *VDM* (editing) and *Debug* (execution) perspectives. The main view (an element of a perspective in Eclipse terminology) provided by Counterpoint will be a proof obligation manager. The proof obligation manager will display a

[3] VDM++ and VDM-RT are object-oriented extensions of VDM-SL.

[4] A technical report describing this work is in preparation at the time of writing.

list of the proof obligations that must be discharged for the current project. Proof obligations that assert a model's consistency can already be generated from Overture and VDMTools [LLR+10,Ber97]. It is expected that the current underlying proof obligation generator in Overture will remain unchanged, with the graphical interface being superseded by Counterpoint's proof perspective.

For refinement proofs, it will be necessary to define a retrieve function (or designate a function from a specification) and to define the relationship between abstract and concrete operations. Counterpoint will generate proof obligations based on this information. This will likely require some changes to the underlying framework to allow management of multiple specifications and the relationships between them.

Each proof obligation can be discharged by associating it with a proof artifact. Counterpoint will support three types of proof artifact: *automated proof* results, generated by plug-ins and external tools; *natural deduction proofs*, constructed in the Counterpoint editor; and *other evidence*, for less formal proofs. These are explained in greater detail below.

Support for "as automated as possible" proofs (with fully automatic proving being the ultimate goal) is important if the aim is to have proof adopted in industry as a verification technique[5]. The author also believes that the act of crafting proofs and seeing the process is an excellent reason to undertake the task by hand. The author therefore believes that Counterpoint should be a platform in which automated proof and handcrafted proof can coexist, providing the user with the best choice of tools for *their* purpose.

The proof manager will give a visual mnemonic indicating the status of proof obligations. Red will indicate that a proof obligation is yet to be discharged and green indicates a proof obligation that has been discharged. A blue colour will be used for proof obligations that the user asserts to be true, but which cannot be checked automatically by the tool (i.e. other proof evidence). This approach is also taken by the Rodin tool for Event-B [Abr07], however Counterpoint will ensure that some form of evidence is associated with a 'blue' proof.

Full automation of proofs for VDM is unlikely (because of the need to find witness values for existential quantifications, for example), therefore some form of user-guided proof is likely to be necessary. It is key therefore that the various Counterpoint features give understandable and constructive feedback to the user (for both automated and hand crafted proofs).

The three types of proof artefact are now explained in greater detail.

Automated proof Counterpoint's support for automated proof will be based on plug-ins that link to external theorem provers. As noted in [LBF+10], there is currently no VDM-specific theorem prover. Off-the-shelf tools such as HOL [SN08] can however be used for VDM specifications, as seen in [Ver07,VHL10]. In the future of course, a VDM-specific theorem prover may be built and could be integrated into Counterpoint in the same way.

[5] It should be noted that industrial use of VDM has done very well using the existing tool support.

Automated proof artifacts in Counterpoint will contain information required to discharge the proof (e.g. version information, tactics used) as well as information produced by the external prover. Plug-ins should ensure that this feedback to the user is useful, particularly when an automated proof fails. Counterpoint will support the ability to perform a brute force attempt to discharge proof obligations and will automatically attempt to re-discharge proof obligations when specifications are changed.

On the issue of feedback from external proof tools, the author would recommend that feedback is mapped back into VDM syntax (as in PROPSER [AS99,DCN+00]), as opposed to being presented in prover-specific form (as in Vermolen's work [Ver07]). This will ensure that the user is not required to learn the syntax of external provers with which they may not be familiar. This should ensure a smoother, more integrated user experience, especially if (or hopefully, when) multiple prover plug-ins become available. Of course, this puts a greater burden on plug-in developers, especially during initial development or time-limited student projects. Therefore the author suggests a compromise is reached where mature plug-ins with VDM-syntax feedback are included in the official Overture releases, with experimental or early-development plug-ins available as optional extras for keen users.

The emphasis on the initial development of Counterpoint will be to provide extension points for plug-ins (and not on the creation of plug-ins themselves). Extension points will provide an interface between plug-ins and Counterpoint. This will in theory give plug-ins a consistent look and feel for feedback and prover configuration (such as supplying user-defined tactics) and allow plug-ins to access models through the Overture AST (Abstract Syntax Tree). Creation of these plug-ins will be an excellent source for student projects.

Natural deduction proofs The core of Counterpoint's natural deduction proof support will be an editor for crafting proofs. The editor will include automated line numbering to reduce the tedium of updating evolving proofs and will lay out proofs (including hypotheses, conclusions and justifications) in an intuitive way. An ASCII syntax will be defined for the editor, however a more complex file format than plain text may be needed for proofs (e.g. XML).

Counterpoint will support a number of useful tools based around this core proof editor. It will include a directory of theorems, initially taken from [BFL+94]. A view of this directory will be provided that allows users to easily browse and search for theorems, e.g. searching theorems by name, listing all theorems relating to sets, or finding a theorem with a specific conclusion.

In the course of producing proofs, users may create theorems that will be useful to others. During the Mondex work in VDM for example, much time was spent on a lemma which stated that the sum of the values in a set of natural numbers yields a natural number. This could be useful to others in future. To harness this process, Counterpoint will be linked to an online repository of theorems that will allow the 'theory base' [BFL+94] of VDM to expand through shared effort.

Counterpoint will also provide a proof checker, which will use the directory of theorems and the specification of the model in order to check the validity of proofs. The proof checker will check proofs as they are constructed, much like the syntax and type

checker of the VDM core of Overture. The proof checker will also be executed automatically if the specification changes.

Counterpoint will also provide pretty-printing support[6] through generation of LaTeX source using the VDM macros (which already support natural deduction proofs). Generation of a LaTeX sources / PDFs of the entire proof effort of all the proofs of a project will also be supported.

Finally, Counterpoint will offer the possibility to incorporate user-guided proof support, much like that the mural tool [JJLM91]. This support could in fact comprise a reimplementation of the mural engine; or integration of results from research projects such as AI4FM [GJ10]; or take on new ideas of keen students. Or better yet a combination of all of the above. The author believes this is a particularly rich vein for research.

Other evidence This category of proof artifact is designed for less formal proofs and recognizes that not all proofs will necessarily be taken to the fully formal level. These proofs could range from assertions of correctness based on inspection, through to semi-formal proofs and structured arguments such as those given in [Pie09]. Some proof obligations required for consistency are simple or trivial [Ber97]. This mode will support plain text, LaTeX source (including VDM macros), and PDF and image files (e.g. JPG, PNG) as proof artifacts. PDF files and images could be used where current proofs exist in other formats, such as early proofs from scanned manuscripts, for example.

As noted previously, Counterpoint could not automatically check this type of proof, so a blue colour will mark artifacts that the user asserts to be valid. The tool could however mark proofs that must be manually checked after the specification in a way that could affect the validity of those proofs. Because the proofs cannot be checked by Counterpoint, there is an onus on the user to be correct. Counterpoint will at least require that some information is provided as a proof artifact, which means that the user cannot simply dismiss a proof without providing some explanation. This will aid traceability and confidence in the proof effort.

Figure 1 shows a representation of the components of Counterpoint proposed above. There are three types of element in the diagram (based on line colour and fill colour). Shaded (green) elements are already complete, because they are part of the Overture tool already. The author suggests that initial development phase should be focused on the white elements with solid outlines, since these are achievable and form a foundation of the framework. The elements with dashed outlines require this basic framework to exist and likely require more significant research, therefore the author suggests that they be tackled in later phases of development.

A mockup of a possible *Proof* perspective for Overture is given in Figure 2. It shows a Proof explorer view (top-left) that lists the proof obligations of the current model. Each proof obligation has two icons, indicating its type (automated or hand-crafted) and status: red for unproved, green for proved and blue for marked as proved by the user. In the large editor window (right), a natural deduction proof is being constructed. In the workbench (bottom), a search of the 'theory base' is being performed.

[6] As an aside, the author would also be interested in a plug-in for pretty-printing and pretty-editing of VDM specifications (i.e. Math syntax) from inside the Overture tool.

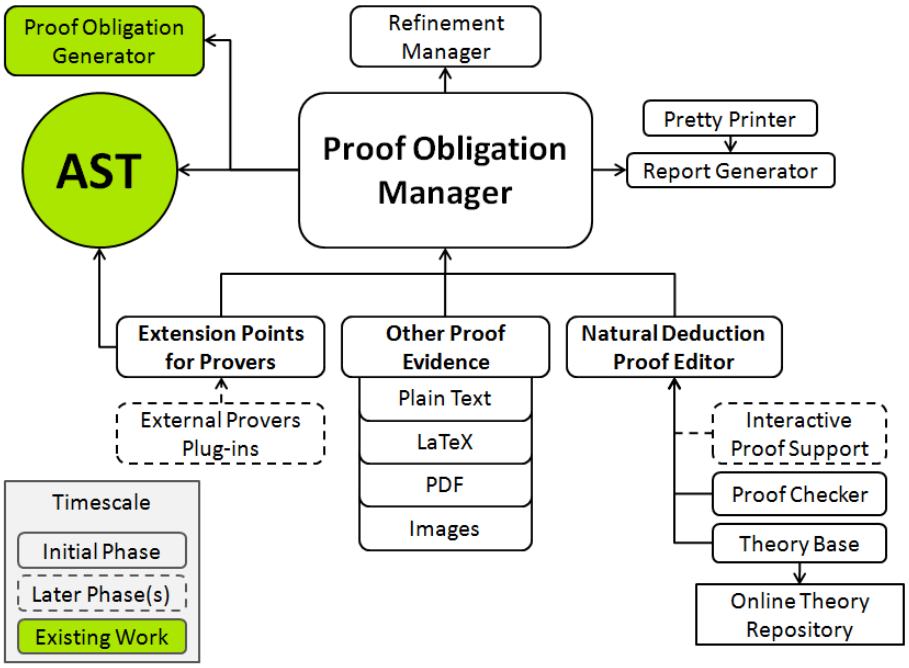

Fig. 1. Overview of proposed components for Counterpoint

3 Aims of the Overture Initiative

Because Counterpoint will be an extension to the Overture tool, the author wishes to reiterate the core values of the Overture initiative[7] (or now perhaps the Overture / Counterpoint initiative):

- to promote and enhance the use of VDM (and formal proof)
- to provide industrial strength tools
- to be open-source
- to be community-driven
- to provide opportunities for research
- to provide opportunities for teaching
- to be a fun project to be involved in!

4 Conclusions and The Way Forward

This position paper presented some ideas for an extension to the Overture tool platform that will support the process of proof in the VDM family of formal languages.

[7] `http://www.overturetool.org/?q=About`

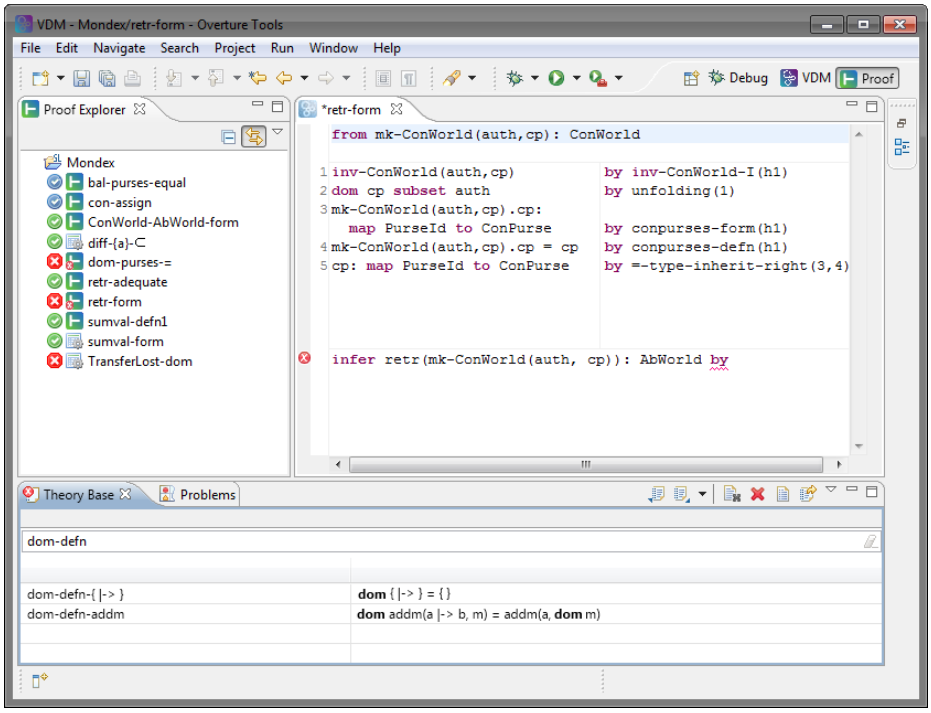

Fig. 2. A mockup of a Proof perspective for Overture

Tool-support for the creation, execution and testing of VDM specifications is currently strong, however tool-support for proof is lacking. The author of the paper hopes that together we can bring tool support for proof up to the current standard of Overture, to provide Overture and our users with an even richer set of features for the specification of systems with VDM.

Since this extension will not simply be a single plug-in, but a whole new perspective (both literally and figuratively) to the Overture tool, the author has chosen to name this extension Counterpoint. Counterpoint will provide support for managing proof obligations; for discharging proofs through external theorem provers; for hand crafting proofs in the natural deduction style; as well as a number of other features to aid the proof process.

The author did not consider model checking in this paper, since he sees it as orthogonal (though complementary) to proof, and that perhaps it would exist in some future *Model Checking* perspective. It would clearly be another excellent string to the Overture tool's bow however, so the author would of course welcome contributions on the topic.

The author hopes that this paper will encourage discussion within the Overture and VDM community. As a next step, the author suggests that the above ideas be formed into a concrete set of requirements, taking community feedback into account. These requirements can then be used to prioritize, plan and measure development effort.

As a rough plan, the author suggests that the creation of a *Proof* perspective and proof obligation manager would be a good first step. Initially, simple proof artifacts (i.e. LaTeX source, PDF files) will be supported. This will create a platform upon which the more complex automated and natural deduction proof support can be built. For natural deduction proofs, basic support will require a definition of a file format and creation of an editor, which should be a relatively simple step in this direction. Creation of a proof checker and user-guided proof support will require further research.

For interfacing with automated theorem provers, the key will be to define extension points that allow plug-ins to interface with Counterpoint and access the necessary model information through the Overture AST. It will be necessary to think carefully about how external theorem prover plug-ins will need to interact with the tool and models. The author suggests that integrating the work of [Ver07,VHL10] into the emerging Counterpoint framework would be a useful pilot project, from which requirements for extension points could be generalised.

Acknowledgements

The author wishes to thank his colleagues Jeremy Bryans, Richard Payne, and Zoe Andrews, who participated in a focus group after the Mondex work at Newcastle. The author also wishes to thank John Fitzgerald and Cliff Jones for discussions on the topic of proof tools, as well as the two reviewers who helped improve this paper. The author's work is supported by the EU FP7 project DESTECS.

References

[Abr07] Jean-Raymond Abrial. Rodin Tutorial. `http://deploy-eprints.ecs.soton.ac.uk/10/1/tutorials-2007-10-26.pdf` (Unpublished), 2007.

[AS99] S. Agerholm and K. Sunesen. Formalizing a Subset of VDM-SL in HOL. Technical report, IFAD, April 1999. http://www.vdmportal.org/twiki/pub/Main/VDMpublications/FormailizingVDM-SL.pdf.

[Ber97] Bernhard K. Aichernig and Peter Gorm Larsen. A Proof Obligation Generator for VDM-SL. In John S. Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997)*, volume 1313 of *Lecture Notes in Computer Science*, pages 338–357. Springer-Verlag, September 1997. ISBN 3-540-63533-5.

[BFL+94] Juan Bicarregui, John Fitzgerald, Peter Lindsay, Richard Moore, and Brian Ritchie. *Proof in VDM: A Practitioner's Guide*. FACIT. Springer-Verlag, 1994. ISBN 3-540-19813-X.

[DCN+00] Louise A. Dennis, Graham Collins, Michael Norrish, Richard Boulton, Konrad Slind, Graham Robinson, Mike Gordon, and Tom Melham. The PROSPER Toolkit. In *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Berlin, Germany, March/April 2000. Springer-Verlag, Lecture Notes in Computer Science volume 1785.

[DHB91] Flemming M. Damm, Bo Stig Hansen, and Hans Bruun. On type checking in vdm and related consistency issues. In *Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development-Volume I: Conference Contributions - Volume I*, VDM '91, pages 45–62, London, UK, 1991. Springer-Verlag.

[GJ10] Gudmund Grov and Cliff B. Jones. Ai4fm: A new project seeking challenges! In Rajeev Joshi, Tiziana Margaria, Peter Mueller, David Naumann, and Hongseok Yang, editors, *VSTTE 2010*, August 2010.

[JJLM91] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *mural: A Formal Development Support System*. Springer-Verlag, 1991.

[Jon90] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990.

[LBF+10] Peter Gorm Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl, and Marcel Verhoef. The Overture Initiative – Integrating Tools for VDM. *ACM Software Engineering Notes*, 35(1), January 2010.

[LLR+10] Peter Gorm Larsen, Kenneth Lausdahl, Augusto Ribeiro, Sune Wolff, and Nick Battle. Overture VDM-10 Tool Support: User Guide. Technical Report TR-2010-02, The Overture Initiative, www.overturetool.org, May 2010.

[Pie09] Ken G. Pierce. *Enhancing the Usability of Rely-Guarantee Conditions for Atomicty Refinement*. PhD thesis, Newcastle University, 2009.

[SCW00] Susan Stepney, David Cooper, and Jim Woodcock. An electronic purse: Specification, refinement, and proof. Technical monograph PRG-126, Oxford University Computing Laboratory, July 2000.

[Sim90] H.R. Simpson. Four-slot fully asynchronous communication mechanism. *Computers and Digital Techniques, IEE Proceedings* -, 137(1):17–30, Jan 1990.

[SN08] Konrad Slind and Michael Norrish. A brief overview of hol4. In Otmane Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer Berlin / Heidelberg, 2008.

[Ver07] S. D. Vermolen. Automatically Discharging VDM Proof Obligations using HOL. Master's thesis, Radboud University, Nijmegen, 2007. Draft.

[VHL10] Sander Vermolen, Jozef Hooman, and Peter Gorm Larsen. Automating Consistency Proofs of VDM++ Models using HOL. In *Proceedings of the 25th Symposium On Applied Computing (SAC 2010)*, Sierre, Switzerland, March 2010. ACM.

[Woo06] Jim Woodcock. Verified software grand challenge. In *FM*, pages 617–617, 2006.

VDM++ as a Basis of Scalable Agile Formal Software Development

Hiroshi Mochio[1] and Keijiro Araki[2]

[1] Chikushi Jogakuen University, 2-12-1 Ishizaka, Dazaifu-shi, 818-0192 Fukuoka, Japan mochio@chikushi-u.ac.jp
[2] Kyushu University, 744 Motooka Nishi-ku, Fukuoka-shi, 819-0395 Fukuoka, Japan araki@csce.kyushu-u.ac.jp

Abstract. This paper presents the use of VDM++ formal specification language as a basis of scalable agile formal (SAF) software development method, which is an agile method for mission-critical or large-scale software development. Combination of agile method, of which usefulness has been recently recognized, and VDM++, a formal method, enables describing system architecture, verifying specification and generating source code all in an agile and formal way. The system architecture and the verification are indispensable for SAF software development.

1 Background

It has been about ten years since the Agile Manifesto was declared. [2] During this period, agile software development method has been more and more adopted in real industry field and realistic results have been obtained. The significance of the agile development method is already undoubted today. Agile development principles, such as iteration based on lean requirements, good communication among the stakeholders and regularly responding to changes, lead to satisfaction of developers and customers. Consequently, it brings about high productivity.

The method of lean requirements enables developers to catch an outline of the iteration that they are working on. Therefore, the developers can easily understand the whole requirements and always keep their motivations high. Good communication enables developers to keep products of high quality because they can recognize a gap between the status of the products and the requirements. Iteration of a short term development enables developers to cope with changes of the market and the customer flexibly.

Contrarily, there have been several comments about proper problems of the agile development method. Lack of ability to develop mission-critical software and lack of scalability are typical ones.

About the lack of ability for mission-critical software, Kent Beck, who promotes eXtreme Programming (XP), one of the most commonly used agile development methods, notes that more than XP may be needed in safety-critical systems. It means that in such situations additional process measures such as documented traceability, formal design review by outside experts and the like, may also be required. [9] In XP, reliability of software is built up by means of test

and review. Since XP is a test-driven development method, at first, tests must be described before design, and then verification by tests is performed at every stage in the iteration. Additionally, pair-programming, so to say, a kind of real-time peer review, is the basic coding style of XP. Therefore, software produced through XP always has high quality, but more reliable means of verification, for example, verification by proof, is needed for safety-critical domains. Documentation in agile development is a little peculiar because of the principle that agile method is based on iteration for lean requirements. In agile iteration, developers should select the functions to be realized from the ordered list of requirements by themselves, and implement them. In such situation working software over comprehensive documentation (Agile Manifesto) is the most important, and so, no rigorous document about requirements or specifications exists. This is because there is a tacit understanding that a developing team can get the iteration into perspective. However, when high safety is required, specification about safety must be shared through the whole development process. In other words some kind of means to describe the specification strictly is needed. On the other hand it has been pointed out in relation to scalability of agile software development that in case of a large-scale project, developers cannot get the whole development process into perspective. Namely agile development method is suitable only for not so large-scale a project. Kent Beck says as follows.

> For the first iteration, pick a set of simple basic stories that you expect will force you to create the whole architecture. Then narrow your horizon and implement the stories in the simplest way that can possibly work. At the end of the exercise you will have your architecture. [1]

This means that if a system is of modest scope such that a small number of stories and an iteration or two can lay out a reasonable architectural baseline, then this approach may be very effective, and architecture can emerge quite nicely in this model. [9] Conversely, if the system is not of modest scope, it may be difficult to lay out an architectural baseline. It is the point of criticism about the scalability of the agile method.

Formal method is the generic name for methods to describe and verify a system based on mathematics, which were originally studied in Europe in the 1970s. Formal methods are usually applied to the upper stage of development so as to exclude ambiguity or errors of specification by rigorous description and verification. These days the application cases to industry field are increasing. As a result, it is found that formal methods are effective in reinforcing safety of a mission-critical system or reducing regression processes in software development.

There are two major usages of formal methods. One is formal specification description and another is formal verification. For the former, formal specification languages, such as Z, B, VDM and OBJ are used. The typical method of the latter is model checking, and many kinds of model checking systems, such as SMV, SPIN and LTSA, are available. Usually the right formal method is applied to the right place. It is unusual to apply only one formal method to all over the development process rigidly.

Sometimes formal methods are criticized because of cost increase with its introduction or developers antipathy against it.

The reason of the cost increase is that at least one new process for formal description or verification must be added to the upper stage of development. Besides, in the initial introduction process of formal methods, cost of training developers is necessary. Nevertheless some report says that these costs are offset by the decrease of regression processes in lower stage of development, which leads to reduction of the total cost.

The antipathy of developers is more critical than the cost increase. Indeed many of developers hesitate to make use of formal methods in spite of admitting the effectiveness of them. In such cases people are likely to regard the difficulty of formal methods as the reason for the hesitation. However, if analyzed in detail, it becomes clear that there is another factor of the hesitation. The true reason developers often reject formal methods is that formal methods are usually introduced into traditional predictable development process such as the waterfall. Since the whole detailed specification of a system must be defined in the early stage of such process, developers are forced to take great pains in upper stage. The developers do not resist the difficulty of formal methods, but resist the difficulty to specify such a system as consists of not predictable factors all at once. [5] If the scope of the target system is modest and the perspective on it is easy to be detected, actually the difficulty of formal methods does not matter.

Agile method and formal method were originated in respective contexts, and have been developed separately. They are often taken as conflicting methodologies. But, the same as the two concepts, agile or formal, are not opposed to each other, the two methodologies are mutually compatible. Appropriate combination of them rather results in more efficient and higher quality development method, because each can get rid of the others problem. [3, 8]

In this paper, VDM++ is introduced as the core of a SAF development method, which is applicable to mission-critical or large-scale software. First, requirements for SAF method and those for the core formal method are showed. Second, after a brief overview of VDM++, it is presented that VDM++ can work as the core formal method of SAF development. Third, realizability of SAF method is discussed referring to a case of VDM++ application from industry and a software development environment with VDM++.

2 Scalable Agile Formal (SAF) Software Development

The requirements for SAF method are roughly divided into those to deal with mission-critical systems and those to deal with large-scale systems.

The former requirements are satisfied by formal specification description and verification, which are primary functions of formal methods. Rigorous description based on mathematics and detailed verification makes it possible to achieve the required high-level reliability or safety.

The latter are satisfied mainly by these three means. [9]

– Intentional Architecture

– Lean Requirements at Scale
– Managing Highly Distributed Teams

Leffingwell explains them as follows.

> An intentional architecture typically has two key characteristics: (1) it is component-based, each component of which can be developed as independently as possible and yet conform to a set of purposeful, systematically defined interfaces; (2) it aligns with the teams core competencies, physical locations, and distribution. Agile teams should organize around components, each of which can be defined/built/tested by the team that is accountable for delivering it. Moreover, because of the existence of a set of interfaces that define a components behavior, teams can be isolated from changes in the rest of the system. Sufficient architecture must be established prior to substantive development. In the first few iterations, a primary version of architecture is built and tested. The architecture should be implemented, not just modeled. This process is called architectural runway. [9]

> Requirements that define performance, reliability, and scalability of a system must be understood by all teams who contribute to the solution. To this end, requirements, which are naturally lean, should have three main elements: a vision, a roadmap, and just-in-time elaboration. The vision carries the common objectives for the teams and serves as a container for key nonfunctional requirements that must be imposed on the system as a whole. The roadmap illustrates how that vision is to be implemented over time in accordance with a prioritized backlog. Just-in-time requirements elaboration defers specific requirements expression until the last responsible moment, eliminating the waste and scrap of overly detailed. [9]

> At scale, all agile is distributed development. Tooling is also different for distributed teams. Larger teams require relatively more tooling, and at enterprise level, a more systematic approach is required. Enterprise-level communication environment needs shared, program-wide visibility into priorities, real-time status and signaling, and dependencies. Teams must have access to networks and Internet-enabled tools that enable shared repositories for agile project management, shared requirements, source code management, a common integrated build environment, change management, and automated testing. [9]

In addition to the above-mentioned requirements, the core formal method of SAF development must meet the requirements of ordinary agile method. Since one of the most important characteristics of agile methods is test-driven development based on the principle of working software over comprehensive documentation, high-quality working software is demanded at every the end of each iteration. Therefore, ability to describe tests, framework for automated tests,

function to animate specification, and an automated code generation tool are indispensable to the core formal method environment.

Taking all mentioned above into consideration, requirements for the core formal method for SAF software development are the following.

1. Rigorous description and verification
2. Test-driven development
3. Object-oriented description of architecture
4. Animation of specification
5. Just-in-time requirements elaboration
6. Automated code generation
7. Internet-enabled tool for communication

3 VDM++ as a Basis of SAF Software Development

VDM++ is an object-oriented extension of the formal specification description language for Vienna Development Method (VDM), which is a formal methodology originally developed at the IBM Vienna Laboratory in the 1970s. VDM++ is a product of the Aphrodite project in EU and its original, VDM-SL, was internationally standardized as ISO/IEC13817-1 in 1996. It can express many kinds of abstract data types based on mathematical equipment, such as propositional or predicative logic, set, mapping, and so on. Hence VDM++ can describe objects in a variety of abstraction levels, from an abstract model like system architecture to a concrete component. It rigorously defines functional behaviors of a system with explicit description of preconditions, postconditions, and invariants. Both implicit and explicit styles are available for definition of a function. Implicit style definition, which defines what to do without any description of concrete processing, declares relation between the input and the output as functional specification. It is usually used to describe highly abstract models. On the other hand, explicit ones, which define how to do the output from the input in terms of algorithm, can run on an interpreter, and so enables prototyping and verification by animation.

It is worth mentioning that VDM++, a formal specification language, resembles to ordinary programming languages in description style. The following is a part of an example in Fitzgerald 2005. [4] It describes the specification of an operation to return the schedule of experts who are called up by the alarm system of a chemical plant.

```
class Plant
...
public ExpertIsOnDuty: Expert ==> set of Period
ExpertIsOnDuty(ex) ==
    return {p | p in set dom schedule &
                ex in set schedule(p) }
end Plant
```

In Java, it would look something like:

```
import java.util.*;

class Plant {
  Map schedule;

  Set ExpertIsOnDuty(Integer ex) {
  TreeSet resset = new TreeSet();
  Set keys = schedule.keySet();
  Iterator iterator = keys.iterator();

  while(iterator.hasNext()) {
    Object p = iterator.next();
    if (((Set)
      schedule.get(p)).contains(ex))
      resset.add(p);
    }

    return resset;
  }
}
```

The VDM++ description looks like that of an ordinary programming language and is familiar to most programmers. Meanwhile it is, as is characteristic of formal specification languages, more abstract than the Java description, where it captured the essentials of the object simply.

Here the VDM++ adaptability to the SAF requirements mentioned above is considered step by step.

Rigorous description and verification: VDM++ can rigorously define behavior of a system with preconditions, postconditions, and invariants in function specification. Additionally it can verify the specification by satisfiability check for implicit definition or integrity check for explicit one. [4]

Test-driven development: VDMUnit is a test framework for VDM++, which is a transplant from JUnit developed for Java by Kent Beck and Eric Gamma. Fig. 1 is an overview of VDMUnit framework. [4]

Object-oriented description of architecture: VDM++ can describe system architecture that defines components and their interfaces of a system in variety of description styles, from the abstract one for concept models to the concrete one for detailed specification.

Animation of specification: One of the most important principles of agile development is constant and close communication with customers. Customers should usually check the products during iteration, so that tasks of the iteration will comply with their stories, which are, so to say, requirements of agile style. Thus moving objects should be demonstrated for the customers to grasp the development state easily. In VDM++ development environment, specification described in explicit style can be animated with interpreter.

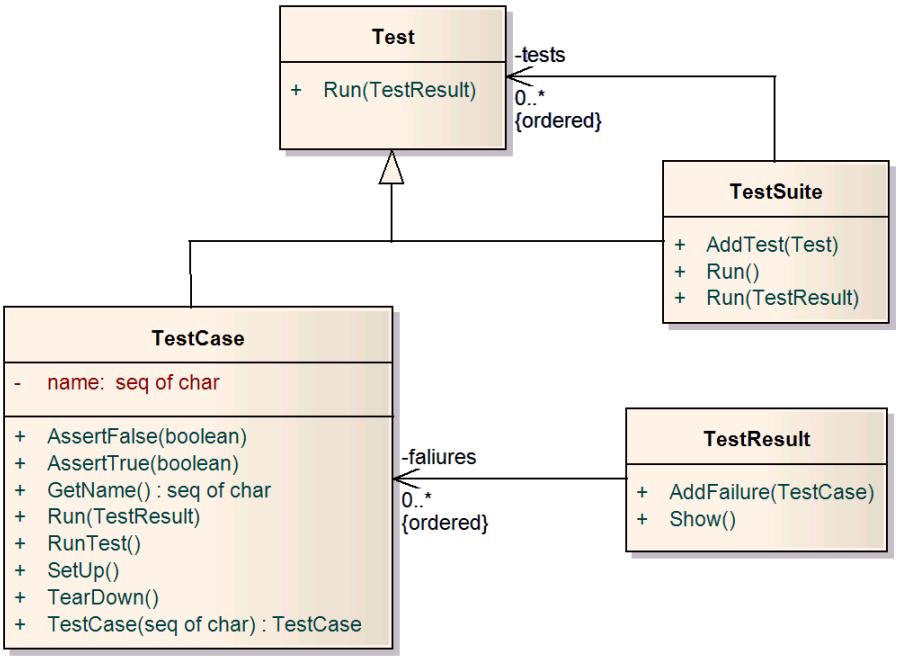

Fig. 1. An overview of VDMUnit test framework

Just-in-time requirements elaboration: VDM++ has an abstract data type, called token, with which VDM++ describes, without concrete definition, such objects as is not necessary to the modeling for the moment. Moreover, when function specification is described in implicit style, no more detailed definition is needed until concrete specification is demanded.

Automated code generation: There are two major kinds of description environments for VDM++, VDMTools of CSK and Overture of the Overture Project. Each of them has function to generate Java/C++ source code from VDM++ specification.

Internet-enabled tool for communication: Tooling, such as above-mentioned VDMTools or Overture, must be expanded and enriched.

4 Prospects

While a large-scale development case with VDM++ and support tooling for VDM++ are presented, realizability of SAF software development is considered.

FeliCa Networks Company developed the firmware of mobile FeliCaIC chip from 2004 until 2006. They used VDM++ to describe the external specification of components, which had about 100,000 lines, while the C++ source code of the firmware had about 160,000. [6, 7] The whole development was based on traditional waterfall process, but some kinds of formal methods were partly

introduced to improve the process. According to Kurita, there were four major purposes in applying formal methods: (1) rigorous specification description all through the development, (2) achievement of high quality in upper stage of development through high-precision description and tests, (3) thorough testing based on specification, and (4) activation of communication within the development team and with customers. As a result they had corrected many faults in the upper stage, and therefore no bug has been detected since the product was released.

Though the project followed the traditional waterfall development process, it implies possibility of VDM++ architecture description in SAF development, because the external specification of the components was described with VDM++. Besides, the fact that the tests was based on the formal specification with VDM++ and the fact that VDM++, a formal method, contributed to activation of communication, both of them are hints of realizability of SAF development with VDM++.

The Overture project is an open-source project to develop new generation tooling for VDM. The mission of the Overture project is twofold: (1) to provide an industrial-strength tool that supports the use of precise abstract models in any VDM dialect for software development. (2) to foster an environment that allows researchers and other interested parties to experiment with modifications and extensions to the tool and different VDM dialects. [10]

Overture is an integrated development environment (IDE) built on top of the Eclipse platform. The core element of Overture, called VDMJ, consists of parser, abstract syntax tree, type checker, interpreter with test coverage function, and integrity examiner. Overture is the integration of VDMJ, editor, file navigator, debugger, and formatting tool, which can be evolved by expansion plug-ins.

Today, large-scale development is almost inevitably done in distributed circumstances. Therefore network-enabled powerful IDE is indispensable to SAF development. Overture is one of the likeliest candidates for such an IDE.

5 Concluding Remarks

With VDM++ and its supporting tool, system architecture can be described, components can be specified and verified, and the source code can be generated, consistently. Thus SAF software development, an agile development method for mission-critical or large-scale systems, is possible with VDM++.

Future works are as follows.

Overture IDE needs to be expanded for scalable agile development. The first thing to do is to equip online video chat and whiteboard-like system as circumstances for real-time communication among all concerned. Secondly test-driven function should be improved so that tests can be generated and run automatically. Additionally, specification animation with enriched interface is desirable because such function is useful especially for communication between a developer and a customer. Then reinforcement of prototyping and code generating function is needed, for working software is the most important thing in agile

development process. Finally a team management tool is necessary for making development teams correspond to the components defined as elements of system architecture.

References

1. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley (2000)
2. Beck, K., et al.: Manifesto for agile software development. Online: http://www.agilemanifesto. org/ (2001)
3. Black, S., Boca, P., Bowen, J., Gorman, J., Hinchey, M.: Formal versus agile: Survival of the fittest? Computer 42(9), 37–45 (2009)
4. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object-oriented Systems. Springer-Verlag London (2005)
5. Fowler, M.: The new methodology. Online: http://www.martinfowler.com/articles /newMethodology.html (overhauled in 2005) (2000)
6. Kurita, T., Chiba, M., Nakatsugawa, Y.: Application of a formal specification language in the development of the Mobile FeliCa IC chip firmware for embedding in mobile phone. FM 2008: Formal Methods pp. 425–429 (2008)
7. Kurita, T., Nakatsugawa, Y.: The application of VDM++ to the industrial development of firmware for a smart card IC chip. Intl. Journal of Software and Informatics 3(2-3), 323–355 (2009)
8. Larsen, P., Fitzgerald, J., Wolff, S.: Are formal methods ready for agility? A reality check. FM+AM 2010, 2nd Intl. Workshop on Formal Methods and Agile Methods pp. 13–25 (2010)
9. Leffingwell, D.: Scaling Software Agility: Best Practices for Large Enterprises. Pearson Education, Inc. (2007)
10. Overture-Community: Overture: Formal modelling in VDM. Online: http://www.overturetool.org/

Toward Customizable and Bi-directionally Traceable Transformation between VDM++ and Java

Fuyuki Ishikawa[1]

GRACE Center, National Institute of Informatics,
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan,
f-ishikawa@nii.ac.jp

Abstract. VDM allows for formalization, verification and validation of software specifications, typically focusing on only abstract essences of the target system. Therefore, it is necessary to derive programs from VDM specifications in an efficient and reliable way, while incorporating details and reflecting implementation strategies. This paper discusses a method and tool to support this process through customizable and bi-directionally traceable transformation. Specifically, transformation rules from VDM++ to Java are specified for explicitly defining the implementation strategies and customizing the code generation process, e.g., introduction of implementation-specific variables. These rules can then be used for bidirectional transformation between VDM++ specifications and Java code. Changes at either of the two sides can be reflected to the other side even with the existence of implementation strategies that essentially lead to gaps between VDM++ and Java. This paper reports initial attempts that started with definitions of variables and method signatures, or structure definitions as in class diagrams.

Keywords: VDM, Bidirectional Model Transformation, Code Generation, Traceability

1 Introduction

VDM is a method for formal specification of software systems [5, 6]. Currently, VDM is said to be lightweight, because the languages and current supporting tools for VDM can be used in a similar way to those for programs. Specifically, modules or classes are defined with variables and methods (functions/operations), and typically tested by using the interpreter [1, 4].

Because structures of the languages for VDM (VDM-SL and VDM++) are similar to those of common programming languages such as Java, it is somewhat easy to syntactically map VDM specifications to implementation code, for many common notations often used. Actually it is necessary to do so in an efficient and reliable way, to reflect what are defined and examined in VDM into implementation code.

On the other hand, abstraction is the key in VDM (or specifications in general) [5,6]. Only abstract essences of the target system are modeled while implementation details are abstracted away. For example, the languages use abstract data types that do not mention how they are allocated with the memory space and manipulated. In addition,

developers may choose to use declarative notations or to omit details unnecessary for the intended analysis of the specification. This way, there are gaps between VDM specification and implementation code. It is therefore necessary to derive programs from VDM specifications in an efficient and reliable way, while incorporating details and reflecting implementation strategies.

Existing VDM tools have not investigated this aspect. For example, code generators in VDM Toolbox only provide a few options and do not allow for incorporation of implementation strategies [4]. As a result, developers often need to modify the generated code to reflect the strategies. In addition, in that case it is necessary for developers to exclude the modified parts to avoid override by code generation and manually manage the changes. As another example, transformation tools between UML class diagrams and VDM specifications have been recently focused on [4, 11]. However, this leads to a situation where developers have two class diagrams: one written in VDM vocabularies with abstraction, and the other written in C++ or Java vocabularies with implementation details. Although the latter is popularly used in common development processes, it is not clear how to locate the former when introducing VDM. In order to effectively leverage VDM in the development process, the essential gaps in such a situation should be discussed and handled.

In response to the problem discussed above, this paper discusses a method and tool to manage the gaps between VDM specification and implementation code through customizable and bi-directionally traceable transformation. The approach is to use transformation rules from VDM++ to Java for explicitly defining the implementation strategies and customizing the code generation process. The rule language allows for local overriding so that default rules are defined to generate valid Java code while developers can add rules to customize the transformation process. In addition, a solid transformation theory and its implementation are used to provide the basis of bi-directionally traceable transformation [3, 7]. This allows for reflecting changes at either of the VDM and Java sides to the other side, even with the existence of the gaps.

Realization of the method and tool requires much effort for coverage of various syntax elements as well as ideally sophisticated user interface. This paper reports initial attempts for proof-of-concept implementation, which deal with very basic parts of VDM++ and Java.

2 Motivation

2.1 Abstraction Gaps between VDM++ and Java

Figure 1 shows a simple example of variable definitions in a VDM++ specification and its transformation into Java code. Besides the necessary syntax translation, this example includes the following transformation.

1. The variable *a* of the *real* type in VDM++ is implemented as the *double* type in Java.
2. The variable *b* of the *real* type in VDM++ is implemented as the *float* type in Java.
3. The variable *s* of the *seq of nat* type in VDM++ is implemented as the *LinkedList<Integer>* type in Java.

```
  instance variables
    private a : real;
    private b : real;
    private s : seq of nat;
    private state : State;
       -- only in model
end TestClass
```

```
    private double a;
    private float b;
    private int x;
    private LinkedList<Integer> s;
    private Logger log;
       // only in impl
}
```

Fig. 1. Transformation Example

4. The variable *state* is unnecessary in the implementation code. This situation happens, for example, when a variable is necessary only to define invariants, or to define a mock to let the model run (replaced by libraries in the implementation code).
5. The variable *log* is necessary only in the implementation code. This situation happens, for example, when details unnecessary for the targeted analysis or value-added functionality such as logging are abstracted away in VDM++.

The transformations #1, #2 and #3 illustrate how abstract types are converted to concrete types that define how to allocate data with the memory space and manipulate. Among them, #1 and #2 illustrate customization (definition of different conversion for the same type). The transformations #4 and #5 illustrate model-specific or implementation-specific variables that exist at only one side of VDM++ and Java. Although the example only includes variable definitions, equivalent discussion stands also for method definitions.

This way, VDM, or early (formal) modeling methods in general, essentially leads to abstraction gaps. Specifically, to clarify the links between abstract model and the implementation, it is necessary to explicitly distinguish and manage different aspects included or excluded in the abstract model.

- What aspects in formal specification (VDM) are essential decisions, inherited to the implementation (Java code)
  - Inherited as they are (with syntax translation)
  - Inherited as with additional decisions (e.g., how to allocate and manipulate data on memory)
- What aspects in formal specification (VDM) are tentative and unnecessary in implementation (e.g., assertions, tentative mock to let it run without concrete implementation)
- What additional aspects are introduced only into the implementation (e.g., loggers)

This paper focuses on these gaps between VDM++ and Java. The gaps essentially come from the fact generally design decisions or implementation strategies are made and introduced when deriving implementation code from specifications. The approach in this paper thus does not consider VDM-to-Java ”translation” but ”transformation” (not generating code with equivalent structures and behaviors).

2.2 Expected Usage of VDM

VDM does not define one specific usage on how to incorporate it into the development process. This paper focuses on usages to rigorously model and validate design, used as input to implementation teams (rather than to model early requirements only for understanding the domains and problems there). VDM is suitable for the usages, compared with other methods for abstract and formal modeling, as its languages include more concrete and design-aware syntax such as object-orientation.

As illustrated in the example in Section 2.1, each class is modeled with some abstraction, but the basic class structures are discussed defined concretely. With this expectation, this paper does not consider integrating variables and methods from multiple VDM++ classes to one Java class, or decomposing variables and methods in one VDM++ class into multiple Java classes. The latter is especially useful to gradually introduce complexity, but is covered with other methods such as Event-B [2] or similar refinement methods for VDM [10].

2.3 Goal Setting

For the usages to be cost-effective and attractive, it is necessary to reflect what are modeled and validated in VDM into the implementation, in an efficient and reliable way. This paper proposes an approach to explicitly describe the gaps between the formal specification and the implementation as transformation rules. Below describes the goals this approach is intended to achieve, as well as detailed approaches given characteristics of VDM and the expected usages.

First, customized code generation is supported. In the approach, developers can customize the code generation to incorporate their own abstraction strategies or implementation strategies by specifying transformation rules. Assuming similar structures in the formal specification and in the implementation, the transformation rules denote strategies such as making data type concrete. The rules themselves just denote syntax transformation to accept wide range of customization, e.g., use of database connectivity and comment insertion. On the other hand, it is costly and often unnecessary to specify all the transformation rules. Therefore default rules are provided, and the proposed language for transformation rules allows for customization through overriding the defaults. This approach also facilitates to leverage hierarchical definition and reuse of rule sets, e.g., to reflect common implementation strategies in the domain, to generate comments in specific styles required in the team, or to generate annotations for further processing and analysis.

Second, verification through common test cases is supported. As VDM itself, if referred to as a lightweight method, does not define a specific formal way to obtain concrete code that satisfies what are validated in the specification. The approach in this paper does not consider to formally obtain code, either, as it allows for very wide range of syntax transformation. This point is different from fully formal methods that consider stepwise refinements where each refinement step is semantically (mathematically) describable and provable. Although stepwise refinements would be possible also in VDM, current tools do not support and there will be limitations when dealing with

object-orientation in VDM++. Instead, test cases used for checking the VDM specification should be used for checking the implementation code. To support this test case inheritance, it is possible to automatically generate test code for the VDM specification and the implementation from one configuration, by understanding the abstraction gaps explicitly specified within the transformation rules. This aspect was discussed in the author's previous paper [8] and is omitted in this paper (though trivial changes are necessary).

Finally, traceability between formal specification and its implementation is supported. Transformation rules explicitly keep the relationships between the VDM specification and the implementation code, or how the latter is derived from the former. The relationships are essential to understand and make modifications in an existing set of specification and implementation. This paper constructs the transformation rules on the basis of a solid underpinning for bidirectional graph transformation. It allows for tracking what part in the VDM specification correspond to what part in the implementation code. In addition, it also allows for automatically reflecting changes in the implementation code to the VDM specification. Specifically, given the limitations of the current code generator, discussed in Section 2.1, this paper aims at supporting the following properties,

1. Suppose Java code J is generated from VDM++ specification V, and J is modified into J' only by introducing implementation-specific elements. Generation of VDM++ specification from J' then leads to V. The same holds for the case where $J' = J$.
2. Suppose Java code J is generated from VDM++ specification V, and V is modified into V' only by introducing model-specific elements. Generation of Java code from V' then leads to J.
3. Suppose Java code J is generated from VDM++ specification V, J is modified into J', and VDM++ specification V' is generated from J'. Generation of Java code from V' then lead to J'.

The current scope of this paper is to investigate benefits and limitations of the proposed framework when the transformation rules are explicitly given by developers. Further challenges are discussed as future work, i.e., to deal with vagueness that appears when explicit knowledge is not given by developers.

2.4 Expected Features

Figure 2.4 illustrates expected features and usages of the proposed framework. As in common usages of VDM, formal specification models are built in the VDM++ language with some abstraction strategies. Class diagrams can be used to first clarify the structural aspects (variables and method signatures) of the design before the rigorous specification in VDM++. Behavioral aspects can also be modeled too, in an abstract but executable way. The VDM++ specification is analyzed through type check, review and other techniques as well as check with given test cases. These tasks can be supported by existing tools, i.e., VDM++ editor, interpreter, debugger and test frameworks as well as UML editor and UML-VDM translators.

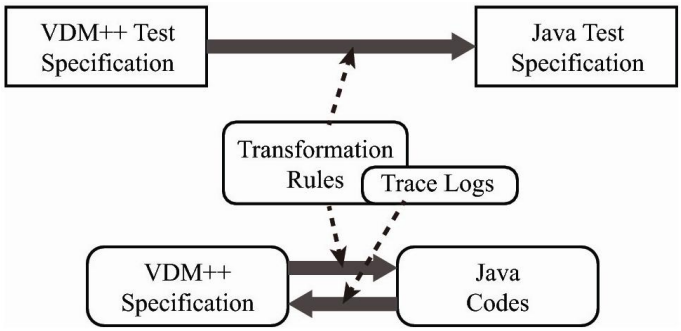

Fig. 2. Expected Usages of Proposed Framework

Transformation rules are then specified, defining differences to be introduced between the VDM specification and the Java code, besides the syntax differences. Java code are then generated using the rules. The transformation procedure internally processes parse trees of the VDM specification and the Java code. Transformation itself can be done with Abstract Syntax Trees (ASTs), but locations of syntax elements in texts are also kept for implementing visual support for traceability. This paper uses the term ”parse tree” instead of common ”AST”. In addition to transformation of the VDM specification (of the target system), the VDM test specification can be also transformed into Java test code, by using the transformation rules. Thus common test cases can be checked both for the VDM specification and the Java code.

When modifications are made in the VDM specification, Java code are generated again. Depending on contents of the modifications, transformation rules may be changed as well. It is possible to extract what parts of the VDM specification are affected by each rule, to help understand whether each rule is affected by the modifications or not. When modifications are made in the Java code, basically they can be reflected back to the VDM specification by using the logs kept in the previous transformation (VDM++ to Java). However, completeness of this functionality depends on the user interface to manage changes at the Java code, which is out of the scope of this paper.

Features involved in the above description may be used in different ways. For example, developers may choose to only rigorously model and validate the interface, and use generated Java code as skeleton. Developers may define rules incrementally and iteratively, to check the result of transformation, find points to customize by additional rules, and run transformation again.

This way, the framework provides support for connecting various deliverables in the VDM++ (or abstract modeling) world and ones in the Java (or implementation) world.

3 Overview of Approach

3.1 Transformation Rules

Transformation rules specify how to syntactically transform specific parts of a VDM++ specification into Java code. First of all, pure syntax translation, without introducing any implementation decisions, is at the base of transformation. Suppose VDM++ specification includes the following variable definition.

```
private x : real;
```

This fragment is translated into the following Java fragment.

```
private real x;
```

Syntax differences (the order and the delimiter in this case) are handled, without any transformation rules.

The above example is only for illustration, as *real* is not a valid Java type. There is no really equivalent type in Java, as *real* in VDM++ refers to a mathematical notion and does not define any specific format put on the memory [1]. Therefore a transformation rule is mandatory in this case to declare how the *real* type is implemented in Java, as *double*, as *float* or possibly as a user-defined class.

Suppose *double* is chosen to implement all the references to the *real* type. The following rule indicates this decision.

```
type-implement: real by double
```

This rule changes the transformation result as follows.

```
private double x;
```

The first part (*type-implement* in the rule denotes a pattern of implementation decisions. Default rules are defined so that valid Java code can be obtained even if developers define no rule. The current framework follows an existing code generator, and choose *double* as the default for *real*. It is actually unnecessary for developers to specify the above rule by themselves.

Suppose only for the variable *x*, exceptionally the *float* type is used. Another rule, reflecting an implementation decision, is then added by developers. This rule clarifies to which part the rule is applied, and locally overrides the above rule.

```
class: TestClass{
  type-implement: real by float in variable x
}
```

This way, the language for transformation rules allows developers to customize code generation behaviors when the default is not acceptable. The remainder of this section describes patterns embedded in the rule language, which are extracted from existing literatures on VDM, primarily books [5, 6].

Other rules include introduction or removal of a new variable, a new argument of a method, a new method, and a sentence inserted within the behavioral description of a method.

[1] VDM tools, especially interpreters may define specific formats to implement the real type

3.2 Foundations in Transformation

A theory for bidirectional graph transformation is applied to process the transformation rules [7]. It defines a set of graph transformation functions and their semantics so that changes in the result graph can be reflected to the source graph in a well-behaved way (in a certain sense).

Graph transformation can be defined by using the languages UnCAL or UnQL+. UnQL+ provides a high-level notation for four types of manipulation, select, replace, delete and extend. On the other hand, UnCAL is a foundational algebra with full expressivity, working as the background of UnQL+. In this paper, the high-level language UnQL+ is sufficient to illustrate the essences, though implementation of the proposed framework may also use UnCAL for detailed control of transformation.

UnQL+ allows for definition of transformation as a query, similar to a SQL query, extracting specific parts from the source graph and constructing a graph possibly adding new parts. For example, below reviews the rule in Section 3.1, that implements the *real* type by *double*.

```
type-implement: real by double
```

This rule is converted to UnQL+ queries including the following one, which replaces the term *real* in the type declaration in each variable definition.

```
replace
  varblock.vardef.vartype -> $a
  by {double:{}}
  in $db
  where {real:{}} in $a
```

Figure 3.2 illustrates an expected VDM++ syntax tree and how this query works. The first line denotes the type of query: replace some parts of the source graph with a given graph. The second and fifth lines define subtrees to be replaced. The second line refers to child subgraphs of a node reached by tracing the path *varblock.vardef.vartype* from the root. The fifth line defines conditions to declare each of the extracted subgraphs is replaced only if it contains a leaf node with the label *real*. The third line defines each of the subgraphs is replaced with a leaf node with the label *double*. The fourth line just refers to the source graph as the input to the processor (*$db*).

The above query is one of the queries generated from the transformation rule to implement *real* by *double*. For example, another rule is necessary to use *Double* inside the complex type (e.g, *HashSet<Double>*). This rule is converted to a similar query, but a regular expression are used for the path description to match occurrences of *real* nested in complex types.

A result graph of a query can be an input to another query. UnQL+ ensures composablity, i.e., it is possible to define a complex transformation by defining and applying small transformations one by one. In the proposed framework, each transformation rule is converted to one or a few queries in UnQL+, and then processed in order. When a query is composed from multiple rules, specific rules are evaluated before default rules. In the example described in Section 3.1, first the specific occurrence of the label *real* are replaced with *float*, and then remaining occurrences of *real* are replaced with *double*.

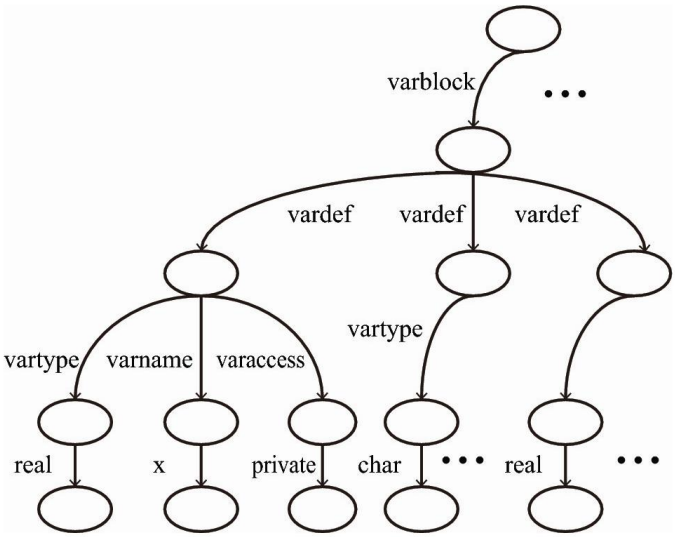

Fig. 3. Manipulation on Syntax Tree

The other types of queries are also available, selecting or deleting designated subgraphs as well as extending (inserting) a graph to be a child of the designated path. With these types of UnQL+ queries, the transformation rules can be implemented on the basis of a solid graph transformation theory, though further examples are omitted.

3.3 Understanding and Tracing

The underlying theory and tool make logs about from which node in the VDM++ syntax tree each node in the Java syntax tree is derived from (see [7] for details). This allows for extracting correspondences between VDM++ fragments and Java fragments. When VDM++ fragments are removed by transformation rules, there is no corresponding Java fragments. The same stands for the case when Java fragments are inserted.

As a UnQL+ query includes description of target subgraphs to be replaced, deleted or extended, it is possible to construct a select query to extract the subgraphs. Thus it is possible to identify VDM++ fragments that are replaced or removed by each transformation rule. In addition, it is possible to identify Java fragments that are inserted by each transformation rule by identifying and logging nodes newly introduced by each rule application.

With these mechanisms, it is possible to identify correspondences among VDM++ fragments, Java fragments and transformation rules.

3.4 Reflecting Changes Backward

This paper discusses what support is feasible when the Java program is modified, and then the changes are reflected back to the VDM++ specification.

When a syntax element is replaced, added or removed in the Java program, it can mean either an implementation-specific decision or an essential change that should be reflected to the VDM++ specification. Therefore it is necessary for supporting tools to ask developers to make some input to identify the intention.

When a modification in the Java program means an additional implementation-specific decision, a transformation rule should be defined accordingly (e.g., changing the way a concrete type is implemented, removing or introducing a variable or method). The rule is necessary to keep the modification even if a Java program is regenerated from the VDM++ specification (possibly with further modification). Thus this case can be dealt with the presented framework. Practically, automatically deriving a rule from the edited Java program would be attractive and feasible, rather than explicitly inputing the rule, but it is out of the scope of this paper and will be discussed as future work.

When a modification in the Java program means an essential change, it depends on the kind of the modification how it should be reflected to the VDM++ specification. Below discusses this point.

Suppose an element that equally exists in both VDM++ and Java is replaced. An example of this case is renaming of a variable or method, which requires the same renaming in the VDM++ specification (note that any transformation rule does not change a name of a variable or method). In this case, it is possible to reflect the change in the Java syntax tree back to the VDM++ syntax tree. The underlying transformation theory originally supports this kind of backward transformation, though the tracing mechanism presented in Section 3.3 can do as well. On the other hand, it is necessary for the user interface to understand the renaming change occurred. This will be realized, for example, by forcing developers to use a provided command for renaming (common in Eclipse-based editors), or by detecting text edit by the developer. The same discussion stands for removing an element that equally exists in both VDM++ and Java.

On the other hand, careful consideration is required for reflection of insertion in the Java program to the VDM++ specification. Generally in the underlying transformation theory, there can be multiple source graphs (VDM++ parse trees) that are transformed into the identical target graph (Java parse tree). In the case of insertion, logs kept in the previous forward transformation do not provide any information for identifying how a unique source graph is chosen among the possible ones, specifically for the inserted nodes at the target graph. This general discussion also stands for the transformation rules proposed in this paper. For example, a new *int* variable in Java may be reflected back as a new *int* variable in VDM++, but there is no reason to exclude the possibility to have a new *nat* variable in VDM++. Use of custom rules makes it difficult to deal with this problem. Because of this essential difficulty, currently insertion is recommended to be made in the VDM++ specification, though accumulation of practical use will lead to definition of default unique inverse transformation rules for insertion.

4 Prototype Implementation of GUI-based Tool

This section describes a prototype implementation of GUI-based tool for the framework.

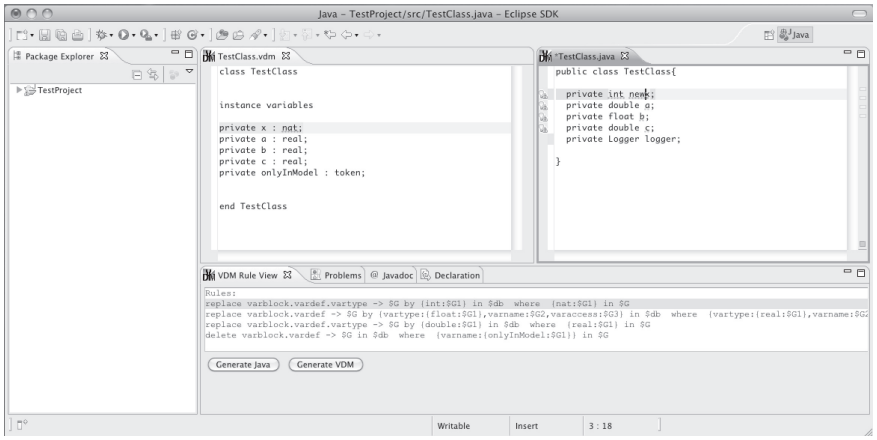

Fig. 4. Screenshot of GUI-based Tool

The tool internally uses the implementation of the transformation theory with UnQL+/UnCAL, called GRoundTram [3]. GRoundTram provides the functionalities for transformation from a VDM++ syntax tree to a Java one, leaving traces for understanding from which VDM++ syntax elements each Java syntax element is derived. GRoundTram also provides the functionalities for backward transformation using the logs in the forward transformation, involving change detection in the Java syntax tree.

The tool implements the features described in Section 3, which are integrated with the features of GRoundTram. The features include conversion of transformation rules into UnQL+ queries as well as construction of select queries to identify the VDM++ fragments to which each rule affect.

Currently, the primary feature in terms of the user interface is a three-window interface to help understand and trace relationships between VDM++ specification, transformation rules and Java code (Figure 4). When a text fragment is selected in one of the three texts (i.e., VDM++, Java, and transformation rules), highlight related text fragments in the other two texts. For example, when some text fragment is selected in the VDM++ specification, then transformation rules are highlighted that make changes on the fragment. At the same time, Java code that correspond to the VDM++ fragments are also highlighted. This user interface can be implemented with the mechanisms described in Section 3.3.

The current implementation is based on partial syntax definitions of VDM++ and Java. Future work for practical use includes full coverage of the syntax or integration with existing parsers.

5 Concluding Remarks

This paper has discusses a method and tool to support the process to derive implementation from VDM specification, in an efficient and reliable way, through customizable and bi-directionally traceable transformation.

The approach allows developers to choose any point between two extremes: fully automatic code generation without customizability and fully manual coding. Examples include cases where only some classes are implemented to accept concurrent access, and cases where a platform-specific library is used to replace a few mock classes in VDM++. On the other hand, the approach is also suitable for iterated and derivative development where both of the specification and the implementation need to be updated consistently. The transformation rules work as explicit documentation of relationships between the specification and the code, which is essential especially when responsible developers change.

Future work includes enhancement of practical implementation, such as coverage of default transformation rules, dedicated user interface and evaluation with large specifications and various implementation strategies (e.g., using databases). Future work also includes semantical support on the top of the current syntactical layer, in order to ensure transformation results are valid, at least with the default rules and preferably certain types of custom rules by developers. However the author believes the approach presented in this paper provides a solid foundation for traceability between formal specification in VDM and its implementation with programming languages.

Acknowledgments

The author would like to thank participants in Top SE [9], an education program for the industry, for questions and opinions on the problem discussed in this paper. The author would like to thank for the BiG project [3] team for quick support in the GRoundTram tool as well as Florian Wagner for his help on implementation of the prototype.

References

1. Overture - Open-source Tools for Formal Modelling. `http://www.overturetool.org/`
2. RODIN - rigorous open development environment for complex systems. `http://rodin.cs.ncl.ac.uk/`
3. The BiG Project. `http://www.biglab.org/`
4. VDM information web site. `http://www.vdmtools.jp/`
5. Fitzgerald, J., Larsen, P.G.: Modelling Systems: Practical Tools and Techniques in Software Development. Cambridge University Press (1998)
6. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs For Object-oriented Systems. Springer (2005)
7. Hidaka, S., Hu, Z., Inaba, K., Kato, H., Nakano, K., Matsuda, K.: Bidirectionalizing graph transformations. In: The 15th ACM SIGPLAN International Conference on Functional Programming (ICFP 2010) (September 2010)

8. Ishikawa, F., Murakami, Y.: Challenges in inheriting test cases configurations from vdm to implementation. In: The 7th VDM-Overture Workshop (2009)
9. Ishikawa, F., Taguchi, K., Yoshioka, N., Honiden, S.: What Top-Level Software Engineers Tackles after Learning Formal Methods - Experiences from the Top SE Project. In: The 2nd International FME Conference on Teaching Formal Methods (TFM 2009). pp. 57–71 (November 2009)
10. Kawamata, Y., Sommer, C., Ishikawa, F., Honiden, S.: Specifying and checking refinement relationships in vdm++. In: The 7th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2009) (2009)
11. Lausdahl, K., Lintrup, H.K., Larsen, P.G.: Connecting UML and VDM++ with open tool support. In: The 16th International Symposium on Formal Methods (FM 2009). pp. 563–578 (2009)

Utilizing VDM Models in Process Management Tool Development: an Industrial Case

Claus Ballegård Nielsen

Aarhus School of Engineering, Denmark, clausbn@iha.dk

Abstract. Around the world companies are striving to become more effective and productive by improving their processes and workflows, within their organisation. Software tools exist, which aid and support process management and improvement methods. These tools are complex and difficult to build because they must ensure the integrity and consistency of the processes, while still being accessible, comprehensible and easy to use. This paper presents the work effort and the initial results of an academia-industry collaborative research project, which takes its foundation in the further development of an existing software suite, from a company developing advanced process management tools. An executable formal model has been created, via the Vienna Development Method (VDM), in order to analyse both the existing tool as well as an expansion, aimed at making the process management tools much more dynamic. The formal model was used to; (1) assist in the exploration of the system domain, (2) to aid the communication leading to more informed design decisions, and (3) to establish insight into complex dependency relations while still ensuring the consistency of the processes. The project involved a development scenario where the industry partner had no knowledge of formal modelling, and the academic partner had nearly no domain knowledge of the business field, therefore graphical representations have been linked to the model to aid the communication between stakeholders.

1 Introduction

In the ever-expanding commercial business market there is a constant push towards maximizing throughput and minimizing costs and development time in order to become more competitive, and preferably improve customer quality in the process. A key factor is increased efficiency, which can be accomplished by improving the processes and workflows within an organization. Software tools exist which are aimed at aiding and supporting process management and process improvement methods. The key focus of these tools is on establishing, maintaining and presenting process descriptions in an effective manner while still ensuring the integrity and consistency of the entire process.

This paper presents the work-effort and experiences gained through a research project which seeks to improve organization processes through the use of advanced tool support. The tool focuses on the knowledge that the end users of the process descriptions have and on the integration of concrete project data directly into the process.

These tools are complex and their inner behaviour is difficult to grasp, because they consist of many different data types which have a high number of relations in cyclic patterns, and there are imperative demands for versioning and traceability.

An existing tool, supplied by the industry partner Callis[1], is used as the baseline on to which the research effort is applied and a tool extension is to be developed. To clarify the complexities found in these tools and comprehend the effects on the tool that the intended extension will have, formal modelling has been utilized in the project. A formal model is used to analyse and describe the tool; as opposed to describing the business process descriptions themselves. The use of formal techniques for analyzing and validating software specifications and designs has been widely encouraged and extensively researched [5,15]. An executable model containing the key elements of the process management tool and the intended extension has been created, using the Vienna Development Method (VDM) [4,10,9]. By having an executable model the unclear parts and areas of concern can be analysed in a lightweight manner by modelling the functional behaviour, thereby adding more precision and confidence to the development process. VDM has been utilized in a wide variety of areas in the specification of software systems [3,13,14,2], however this is the first time VDM has been applied to a process management tool.

It has been anticipated that one of most important effects of having a formal model will be an increase in the quality of information exchange and communication between the project partners. This is especially important in the given project, because it consists of an industry partner with no knowledge of formal modelling, and an academic partner with nearly no domain knowledge of the business field.

Formal modelling has been used in numerous process management and modelling approaches [7]. Mishra et al [12] use the formal specification language CSP-CASL to integrate product and process quality. The applicability of formal specifications for realizing the objectives of process areas in CMMI is investigated, with the aim of contributing to the prospect of automation in process compliance. Van der Aalst [1] discusses the use of Petri nets for optimizing and supporting business processes in the context of workflow management. The graphical language and formally defined semantics of Petri nets ensures a clear and precise definition, that allows for the use of Petri nets as a design language for the specification of complex workflows. Despite the use of formal methods in these approaches in relation to processes, they are not directly related to the use of formal methods in this project. Most of the existing literature on the subject focuses on the formal method itself as a tool for describing and verifying the processes and not on specifying a software tool for supporting the process.

The remainder of this paper is set out as follows. Initially, Section 2 presents the notion of process improvement in the context of this paper. An outline of the research project is given in Section 3, while Section 4 supplies an overview of Callis' tool suite, in relation to which the work is performed. Section 5 contains the reasoning and expectations of using formal modelling as well as an outline of the performed work. The results and experience obtained are presented in Section 6, followed by concluding remarks in Section 7.

[1] Callis `http://www.callis.dk/`

2 Process Improvement

In the context of this paper a process is a well-established chain of procedures or subtasks which are to be performed to handle a certain task or operation. Companies and organizations use processes and process descriptions to aid the business in resolving tasks and fulfilling certain business objectives, through the means of a unified, structured and optimized workflow. Organizations are always searching for ways of improving their performance, and a well-established approach is to optimize the organizations existing processes, through process improvement.

In supporting the efforts in process improvement, key mechanisms include process methods, process standards and process management tools [6]. These tools aid organizations in defining and modelling their processes as well as publishing, broadening and maintaining existing business processes.

To the customers of Callis, the main responsibilities and goals required by these tools are:

Definition Aiding in the establishment and maintenance of the business process descriptions, as well as enabling interrelations between the organizations custom processes and standardized process descriptions such as ISO9001[2], Prince2[3] and CMMI[4].

Control Controlling the process description with regards to configuration, versioning and progress, in order to ensure traceability, evidence, integrity and metrics.

Communication Communicating the process to the users in an efficient, consistent and easy way, to promote and encourage the use of the process descriptions.

The establishment and management of a process generally involves multiple levels, which each have a role in relation to requirements and expectations of the process.

The overall process is defined and specified at a high level, which is detached from the individual projects which are actually the future users of the process, normally by management and process improvement agents. Subsequently the process description is passed down the chain in an organization and may be affected by process agents and managers at national, regional and divisional levels before finally reaching the project level. These levels entail that data in the process management tool may be affected at various levels, meaning that the tool must manage multiple versions, releases and users being related to data.

3 The Research Project

The research project is aimed at improving organizations business processes through the use of advanced tool support. The tool is to be improved by partly focusing on the knowledge and hands-on experience that the end-users of the processes descriptions

[2] ISO9001 `http://www.iso.org/iso/catalogue_detail?csnumber=21823`

[3] Prince2 `http://www.ogc.gov.uk/methods_prince_2.asp`

[4] CMMI `http://www.sei.cmu.edu/cmmi/`
(Accessed: May, 2011)

possesses, and partly by integrating concrete project data and artefacts directly into the process descriptions.

The *first* aspect of the project is to create an extension of the existing tool which supports and strengthens the end user's ability to interact with the tool, such that it can be used actively for establishing workflow overviews and concrete observations about process descriptions. Currently the process management tool is of a static natured, expect for a minor corner of the tool. Static means that the process descriptions are fixed and inactive, they appear as read-only descriptions of how a process should work. In one perspective this is essentially a desired behaviour which is required from a process manager and auditor standpoint. For reasons of traceability and appraisals the process description should be carved in stone at a specific version. However in another perspective this presents a challenge, given that the end-users who are actually using the process have no way of interacting with the tool. The knowledge of these end-users is extremely valuable to process improvement efforts because of their in-depth experience with the use of the processes, combined with their proficiencies and expertise in their field. This gives them the capability to identify both irregularities and ambiguities in a process description and to pinpoint elements of the process workflow that can be optimized.

Having inflexible and static processes creates a very rigid chain through which this information needs to pass before it gets worked into the projects process description. This has the consequence that feedback on the process description is lost, which eventually results in process improvement initiatives never occurring. Therefore the future is to expand the tool from being a tool for statically describing processes, into to a dynamic tool which allows for end-user involvement in improvements of processes and dynamic adjustments of a process. This will allow the end-user to become an active part in both the use and future-development of the process. The idea is to move the capabilities of making adjustments to a process, out to the projects and out to the people who are actually using the process. This is achieved by adding *tailorings* to the process description. A tailoring is the procedure of making minor changes, which has a limited project scope, to existing process descriptions through simple text descriptions. Through tailorings adjustments and comments to the process can then be looped back in to the general process or be used as optional examples or best practices *tailoring set* (a managed collection of tailorings) that can be added to other projects.

The *second* aspect of the tool's future is to supply an overview of the project progress based on the defined process and the artefacts specified in the process, with relation to a specific project. By probing the data in the actual files related to the artefact, the status can be read back into the tool and displayed in relation to the process in order to supply an overview of a projects progress. To enable the probing of project artefacts the tool needs to interface with various types of versioning document handling systems and document types. The goal is to create a full loop leading from process modelling and definition, through process usage, to process feedback, and back to including the feedback in the process modelling to complete the loop.

4 The Industrial Case - A Process Management Tool

The process management and modelling tool which is used as case in the project, is an industrial solution which is used by CMMI5 certified companies. The tool has support for the modelling and management of the process architecture, process integrity, and statistics.

The process descriptions are constructed by using a predefined set of standard description elements, into which organization specific data can be entered. The processes are defined by creating a structure of relationships between the description elements. In order to ensure the integrity of the process and to conform to process standards there are a large number of restrictions and rules as to how the description elements can be interrelated. There are a large number of different element and subelement types, which each have different properties; some types can reference themselves, some cannot, some types can reference other types but cannot be referenced by these types, some can only relate to certain types and some can only be referenced by certain types, and some types can only be used for grouping other types. This is illustrated in Fig. 1, which shows an abstract view of the elements in the tool. On the figure some of these relationships can be seen, for example DescriptionElementA is capable of referencing itself, DescriptionElementB can reference all subelement types and DescriptionElementC can reference all other types of DescriptionElements but only reference one type of subelement. To complicate things further each element can have multiple versions and processes can have multiple instances. This high number of relations, cyclic references, requirements and restrictions lead to a complex tool which is difficult to comprehend and get an overview of.

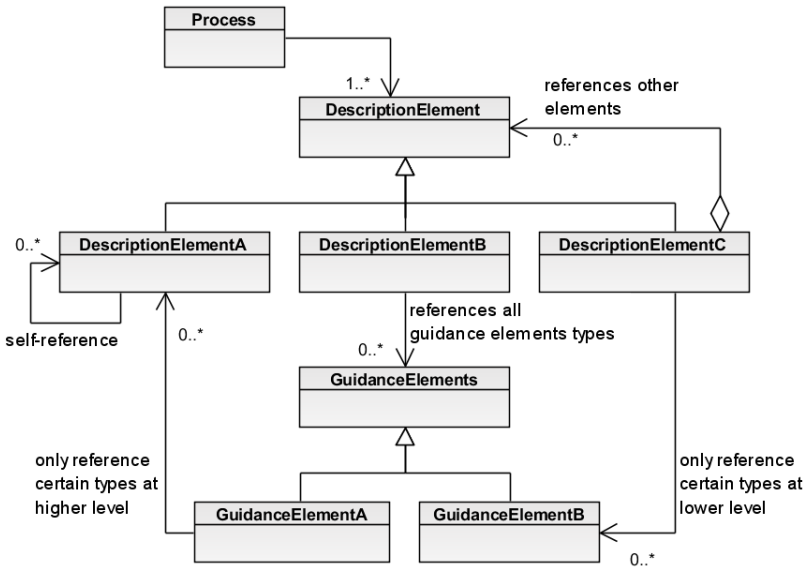

Fig. 1. Abstract class diagram of relations between elements.

5 Use of Formal Modelling in the Project

In order to manage and analyse the complexities of the existing tool and explore the possible solutions in extending the tool, in relation to the research aim, formal modelling has been used from the start of the project. The focus of the modelling has been on abstraction, understanding and communication and not on code verification. The model was developed by one person over a period of two months, with a weekly meeting between the project partners. The final model consisted of approximately 2500 lines of code distributed across 16 classes.

The creation of the model enabled the construction of different scenarios, through which the structures and relations of the elements in a process could be defined. This part of the model reflects the functionality of the existing tool and had three purposes:

- to gain insight into the existing tool and to better comprehend the methodology used in business process descriptions,
- to check the consistency of the existing tool and attempt to reveal unidentified problems,
- and to enable the modelling of the intended extension of the tool, in order to analyse the effects of incorporating the research goal in the existing tool implementation.

5.1 Exploration of the Domain and the Existing Tool

An executable model for a subset of the existing tool has been created in VDM++, with a focus on the structures used in the tool and the relationship between the process elements. Prior to starting the modeling, a smaller process was described using the tool, with the purpose of getting acquainted with both the methodology of business processes and to the concept and features of the tool. The existing source code of the tool was not used during the modeling, however the use of the tool and the central principles of process descriptions were discussed during weekly meetings with a process expert from Callis.

The ways in which the different element types can be combined, as mentioned in Section 4, are strictly controlled by the tool. As a process is built, it is type checked against an internal model of the allowed relationships between element types. Consequently the large matrix of constraints on references which exists between the different types was not included in the modelled subset. Instead the focus was on certain potential issues such as the risk of non-terminating recursion and on the structures which were relevant for the extension.

The classes were structured in a hierarchy which reflects the structure of the tool, but with a more abstract representation of the element types, as depicted in the class diagram in Fig. 2. This class hierarchy enabled the modelling of key elements of both the existing and future tool, and fairly elaborate processes could be defined through the use of the model. In the class diagram it can be seen that the element type is capable of referencing itself and other elements of the same type, thus enabling recursion.

The *ProcessDecorator* and *GuidanceDecorator* classes are part of the Decorator pattern [8] and are used to model future aspects of the tool's future. The *GuidanceDecorator* is used to add tailorings to elements at the lowest level of the tool hierarchy. The

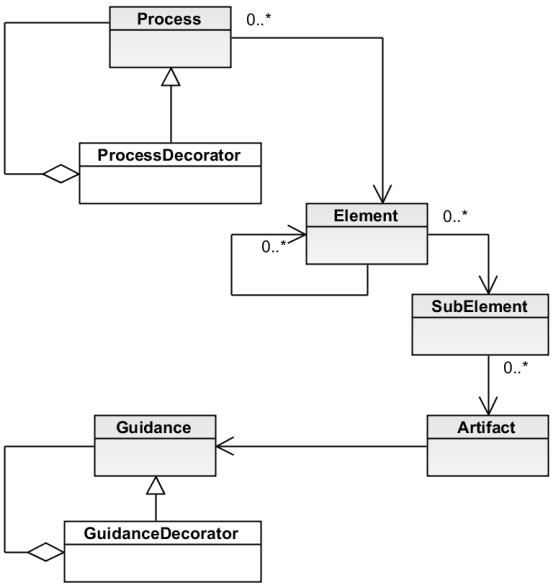

Fig. 2. Class diagram of the models class hierarchy for describing processes

use of the decorator allows for tailoring to be added multiple times to the same element, without directly changing the existing element; this is in accordance with the intended functionality of the future tool.

With regards to tailoring of the process, a tailoring is actually detached from process versions in the future tool. Here the *ProcessDecorator* can be used for altering a process at the top most level, by removing existing elements or by adding new. This was used for creating new versions of an existing process, in order to examine how tailoring were to be handled between different versions the process; for instance what happens to a tailoring that has a relation to a process element, which is then removed in a newer version of the process.

The model allows for different scenarios to be defined and run in order to study diverse process constructs, both in relation to complexity and size. Instances of the class structure in Fig. 2 can be created in a scenario, such that different object structures can be characterized. The capability of running different scenarios is implemented through the use of the Strategy pattern [8], were each concrete strategy represents a scenario. The use of the strategy pattern allowed the scenario to be selected at runtime by changing a configuration file, thus allowing for different scenarios to run without having to change the model itself.

To get an overview of the structures for at given scenario, the Visitor Pattern [8] was used to traverse the object hierarchy and print the object data. Different implementations of the Visitor class were used to focus on certain aspects of model, for instance one was used to present structures relating to tailoring and one was used for construct-

ing a graphical representation of the running scenario. The graphical representation is described in Section 5.2.

An excerpt of a scenario output is shown in listing 1.1, where the structure of the scenario can be seen by each element name being printed at a certain depth. The digit following the # denotes the elements unique ID.

```
###-- Displaying process overview

-Process Set-
----#3 The Super Process 1
--------Elements
----------#4 Element 1
------------SubElements
--------------#5 SubElement 1.1
--------------#8 SubElement 1.2
--------------#10 SubElement 1.3
----------#11 Element 2
------------SubElements
--------------#12 SubElement 2.1
```

Listing 1.1. Excerpt of Scenario Output

By applying a special Visitor, made specifically for tailoring, an output can be created which displays the path to tailored elements, as depicted in Listing 1.2. The name in the parenthesis indicates the source of the tailoring. This output displays a new version of the process set previously shown in Listing 1.1, in which certain elements have been tailored.

```
###-- Displaying tailored elements

-Process Set-
----#18 The Super Process 1 version 2
--------Elements
----------#4 Element 1
------------SubElements
--------------#5 SubElement 1.1
                Tailoring test text (HealthCare Project)
--------------#8 SubElement 1.2
                Check X before that (The SuperProject)
```

Listing 1.2. Excerpt of Scenario Output from Tailoring Visitor

5.2 The Model as a Means of Communication

Having the VDM model enabled different scenarios of distinctive process setups to be defined and analysed by executing the model. However during the development process we found that it was still difficult to discuss the details of the model and to get a quick

overview of the effects that different development suggestions may have on the tool. This was especially true for the industrial partner with no prior knowledge of formal modelling or VDM. Initially some difficulties were caused by the difference in expertise with regards to formal modelling and process descriptions between the industrial and the academic partner respectively. The limited cross-knowledge made it challenging to determine what the modelling could and should be used for. Several times areas of the tool extension, which were considered candidates for analysis in the model, were eventually discarded as they were revealed to mainly concern how data should be selected and managed from a user-interface and usability viewpoint. A subject for which formal modelling is of limited use. This may have been caused by the industry partner's limited understanding of the capabilities of VDM modelling. Likewise the academic partner had a challenge in understanding the process descriptions, because of the narrow knowledge of the business domain. Since the tool is built to support multiple business areas and be compliant with multiple standards it was difficult to get an overview of the constraints in the process descriptions, especially because no formal definition of the processes exists, except partially what can be derived from the entire ISO9001 specification or CMMI models. Instead the construction of the tool is based on de facto standards and experience in how the process description elements are used in practice by process agents.

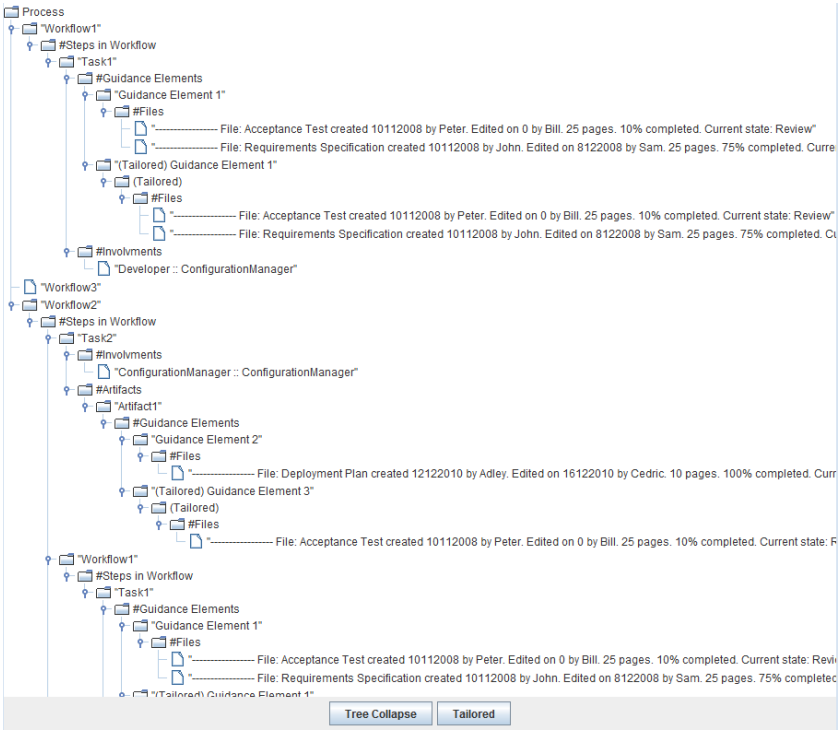

Fig. 3. Screenshot of the graphical presentation of the model

To aid the communication and momentum in the weekly development meetings, a graphical representation was made for a portion of the process structure. This is possible owing to a functionality in Overture [11] which enables the developer to utilize functionality defined in an external Java jar file. It is possible to add a graphical representation on top of the model, which can then be completely controlled from within the VDM model. An example of such a representation is depicted in Fig. 3, where the structure of a process is displayed in a Java JTree structure. Buttons at bottom of the screen enable a switch from a view which focuses on the entire process to a view that focuses on tailored process elements.

6 Results

A key goal of having the model was to use it in relation to the two research aspects and the extension of the tool. Tailoring was designed to strengthen the end-users possibility of interacting with the process, and the model made it possible to (1) examine how different sources of tailoring could be added to a single process and (2) to determine the behaviour of tailoring when applied to different versions of the same process. With regards to the aspect of relating actual project specific data to a process, the model was used for identifying the required data and to examine if the tool would be capable of retrieving the required data. This lead to some further investigation, outside the scope of the modeling, as to how this specific data could be gathered and described such that the tool could process it.

The model did not disclose as many issues as were initially expected, neither in the existing tool nor in the scenarios run for the extension. The only real defect discovered in the existing tool was a risk of recursion in certain scenarios. This is however an issue which is fairly simple to predict, considering the design of the tool. The existing tool has a check for self-referential recursion, but the recursion issue found, was not considered in the existing tool. Nonetheless as it turned out, it was not a risk either as the tool has no automatic traversal of references, meaning the user of the tool would have to traverse the recursive path manually to encounter the problem, and additionally the tool automatically limits its depth to a certain level.

In relation to the extension, the model did not reveal any faults or inconsistencies that could be a cause of concern in a later implementation of the tool. Instead the model confirmed the behaviours and effects that had been anticipated during the continuous planning of the extension. In this sense the model provided assurance in the development process, as it made it possible to test different scenarios, which could then be examined and the expectations could be confirmed.

There were parts of the tool that it was not necessary to include into the model, because they were clarified sufficiently. This has to be attributed to the weekly meetings with the domain experts from Callis, which provided precise and valuable insight into the tool and process modelling in general. Without this knowledge the model would have been substantially larger, because of the many additional unknowns that had to be included and examined. Thus good communication is extremely important and with the

project taken as a whole, the model was mainly used as mean for improving communication and driving development forward. The addition of a graphical representation of modelled scenarios improved and simplified communication between the project members, because it made the modelled scenarios independent from the formal notation used. The visual examples made it easier for Callis to provide feedback on questions raised from the concrete work with the model, leading to a minimization in the risk of requirement misinterpretations.

There were however limits to the areas in which the model was applicable. A lot of the subjects that the industry partner had an interest in modeling, had to do with human-computer interaction and data presentation, e.g. how does one define tailoring for an element or how is data from an artefact presented. These are questions and topics, to which the VDM model could not be used, and more traditional development techniques had to be brought in to play.

7 Concluding Remarks

Specifying and developing new functionality for an existing tool is a challenging and difficult task. Not only must the challenges faced in the development of completely new tools be handled, but the impact on, and the influence of, the existing tool has to be included as well. Anticipating how new additions will affect existing software requires knowledge and insight in to both the business area and the software tool itself. This paper has presented the experiences gained in the use of VDM modelling in the development of a process management tool supplied by an industrial partner. A model has been created which includes a subset of the existing tool, as well as elements of an extension, planed for a future versions of the tool.

The use of the model did not disclose as many issues as were initially expected, and with the exception of a non-terminating recursion risk related to circular references, only minor issues were found. The creation of the model did however provide an insight into the tool and the business domain, which was advantageous given the academic partners limited domain knowledge of the business field. Similarly a feature of VDM was used to create a graphical representation of the model, which made it possible to interact with the model without having any previous knowledge of formal modeling. This made it easier to communicate the modelled scenario between the project partners. In this project the principal value of the model has been the increased communication in the project. The success of a project is very dependent on the communication between the domain expert and the developers. Here the VDM model was successfully used as a means of discussing specification of functionality and in achieving consensus in order to avoid misunderstandings and ambiguities.

The future plans involve a full implementation of the extension into the existing tool, and the deployment of the tool in industry workshops, in order to evaluate the effects on process improvements.

Acknowledgements The author would like to thank the industrial partner Callis for taking part in this research project and in this publication on the project experiences and findings. In particular the development meetings and discussions with Peter Voldby

Petersen have been highly valuable in the comprehension of the Process Management Tools domain and the end-users application of these types of tools. This research project is partially funded by the iKraft Project, as part of Innovation pool in the Central Denmark Region.

References

1. Aalst, W.M.P.V.D.: The Application of Petri Nets to Workflow Management. Journal of Circuits, Systems, and Computers 8(1), 21–66 (1998)
2. Agerholm, S., Schafer, W.: Analyzing SAFER using UML and VDM++. In: Fitzgerald, J., Larsen, P.G. (eds.) VDM in Practice. pp. 139–141 (September 1999)
3. 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. pp. 85–93 (September 1999)
4. Bjørner, D.: The Vienna Development Method: Software Abstraction and Program Synthesis, Lecture Notes in Computer Science, vol. 75: Math. Studies of Information Processing. Springer-Verlag (1979)
5. Clarke, E.M., Wing, J.M.: Formal Methods: State of the Art and Future Directions. ACM Computing Surveys 28(4), 626–643 (1996)
6. Conradi, R., Fuggetta, A.: Improving Software Process Improvement. IEEE Software 19(4), 92–99 (July 2002)
7. Curtis, B., I.Kellner, M., Over, J.: Process modeling. Commun. ACM 35(9), 75–90 (September 1992)
8. E.Gamma, R.Helm, R., J.Vlissides: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series, Addison-Wesley Publishing Company (1995)
9. Fitzgerald, J.S., Larsen, P.G., Verhoef, M.: Vienna Development Method. Wiley Encyclopedia of Computer Science and Engineering (2008), edited by Benjamin Wah, John Wiley & Sons, Inc
10. Jones, C.B.: Systematic Software Development Using VDM. Prentice-Hall International, Englewood Cliffs, New Jersey, second edn. (1990), iSBN 0-13-880733-7
11. Larsen, P.G., Lausdahl, K., Ribeiro, A., Wolff, S., Battle, N.: Overture VDM-10 Tool Support: User Guide. Tech. Rep. TR-2010-02, The Overture Initiative, www.overturetool.org (May 2010)
12. Mishra, S., Schlingloff, B.H.: Compliance of cmmi process area with specification based development. In: ACIS International Conference on Software Engineering Research, Management and Applications. pp. 77–84. IEEE Computer Society, Los Alamitos, CA, USA (2008)
13. Schlatte, R., Aichernig, B.: Database Development of a Work-Flow Planning and Tracking System Using VDM-SL. In: Fitzgerald, J., Larsen, P.G. (eds.) VDM in Practice. pp. 109–125 (September 1999)
14. Smith, P.R., Larsen, P.G.: Applications of VDM in Banknote Processing. In: Fitzgerald, J.S., Larsen, P.G. (eds.) VDM in Practice: Proc. First VDM Workshop 1999 (September 1999), available at www.vdmportal.org
15. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.: Formal Methods: Practice and Experience. ACM Computing Surveys 41(4), 1–36 (October 2009)

Formal Modelling and Safety Analysis of an Embedded Control System for Construction Equipment: an Industrial Case Study using VDM

Takayuki Mori[1,2]

[1] School of Computing Science, Newcastle University, UK
[2] Komatsu Ltd., Japan

Abstract. This paper reports on an industrial application of formal methods to develop an embedded control system for construction equipment. Informal specifications and safety requirements of the system are formalised using a formal modelling language VDM, and the derived model is used for safety analysis of the system. In our approach, we introduce a kind of modelling pattern: a fault framework, which abstracts the notion of faults and can be widely exploited for analysis and design of control systems dealing with faults. The results of validation and safety analysis of the model are presented, and it is revealed that our modelling approach is effectively applied to the analysis of a practical embedded control system in industry.

1 Introduction

A number of functions of modern construction equipment are realised by embedded control systems in order to achieve desired performance, e.g. low fuel consumption and emission as well as high productivity and comfort. The scale and complexity of control software are rapidly increasing. This makes it difficult to ensure the correctness of the software by conventional approaches such as testing and human review. Formal techniques are expected to be promising approaches to make the control software more reliable.

Safety is a critical factor in the control systems of construction equipment. In order to ensure safety, the Failure Mode and Effects Analysis (FMEA) method [5] has been used for decades. In FMEA, we identify all potential faults of the system to be developed and assess the effect of each fault. If the effect is not negligible for the system from the viewpoint of safety or functionality, a way of detecting the fault and a measure to be taken in case of the fault occurrence should be determined to guarantee a certain level of safety or functionality.

The FMEA process is usually carried out manually by system experts. However, the growth in the scale and complexity of the control system makes the task itself more complicated and difficult. For example, it could be possible that a measure against some fault would cause a side effect to another portion of the control system and lead to an unpredictable behaviour. For the above reason, we aim to describe formally the specifications of fault detection and associated measures of the system using a formal modelling notation VDM [1, 2], and check if the system satisfies certain safety properties.

This paper reports on a case study of applying VDM to safety analysis of a transmission controller for a wheel loader (a digging and loading vehicle). The controller is responsible for gear change (including forward-reverse change) of the transmission, which transfers engine power to the wheels. The transmission consists of a number of gears and clutches hydraulically controlled by the system. By engaging the proper combination of clutches, the rotating direction of an axle (that is, the moving direction of the vehicle) and the gear ratio of the transmission are determined.[3] The moving direction of the vehicle is specified by a direction lever which is mounted on a steering column and manipulated by an operator of the vehicle. The proper gear is selected by a gear change algorithm implemented in the controller according to the vehicle speed and the engine revolution etc. In our current research, however, we simply focus on a part of the control system, a specification for detecting the direction lever position, and investigate if its safety properties are guaranteed when some fault occurs in the system. This is mainly because the wheel loader has characteristics that its moving direction is frequently switched by the operator for digging and loading work, and detecting the direction lever position is a crucial factor in the system. Moreover, the scale of the system seems to be moderate for our initial trial.

The rest of the paper is organised as follows. The next section describes informal specifications and safety requirements of the control system considered in the case study. In Section 3, we present a formal model of the system along with a kind of modelling pattern: a fault framework. Section 4 describes the results of validation and safety analysis of the system. Finally, Section 5 concludes the paper. The full VDM++ model for the case study is provided in Appendix A.

2 Informal Description of the System

In this section, we informally describe the specifications and the safety requirements of the control system under consideration.[4]

2.1 Control Specifications

The control system consists of the direction lever and the transmission controller. Figure 1 shows the system diagram. Each component is described as follows:

Direction Lever: It is an input device to the transmission controller, mounted on the steering column of the vehicle and manipulated by the operator. The lever has three positions, namely forward (F), neutral (N) and reverse (R), specifying the direction to go. It generates three digital and one analogue input signals. For redundancy, the former are used as primary signals and the latter is used as a backup. The electrical characteristics of the digital and the analogue input signals are illustrated in Figs. 2 and 3 respectively.

[3] The transmission is 4-speed in both forward and reverse directions.

[4] The example in this paper is simplified to some extent from the original control specifications. Furthermore, tangible data values, such as voltage etc., are not shown explicitly and denoted by symbols for confidentiality reasons.

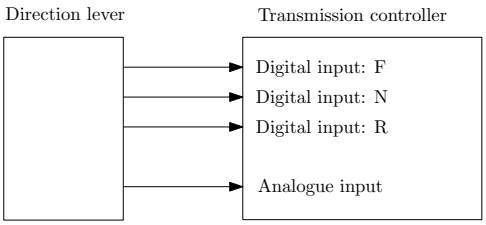

Fig. 1. System Diagram

Digital Input Signals: Only one of the three signals F, N or R is "on" depending on the lever position. The signals do not overlap one another. That is, there need to exist areas in which no digital input signals are "on" between the lever positions F and N, and between N and R. The lever can be intentionally held in the middle of the lever positions. This means that we cannot distinguish open-circuit of the digital input from the case in which the lever is being held in the middle position.

Analogue Input Signal: It is a voltage signal which ranges depending on the lever position, indicating a value from v_{R1} to v_{R2} at the position R, v_{N1} to v_{N2} at the position N and v_{F1} to v_{F2} at the position F. It has some tolerance at each position.

The possible combination of the digital and the analogue input signals is specified in Table 1. For instance, when the lever is set to the position R, only the digital input signal R should be "on" and the analogue input signal should be between v_{R1} and v_{R2}. In case the lever is held in the middle of the positions R and N, the following are possible:

1. The digital input signal R is "on" and the analogue input signal is between v_{R1} and v_{N2}.
2. No digital input signals are "on" and the analogue input signal is between v_{R1} and v_{N2}.
3. The digital input signal N is "on" and the analogue input signal is between v_{R2} and v_{N2}.

This means it is possible that the lever positions detected by the digital and the analogue input are different from each other (though the possibility is low in reality) and this makes the control specifications complicated.

Notes. In older types of control system, the direction lever consisted of only digital input signals. In case a fault has occurred in the system, the lever position is simply regarded as N, which is allowed from a safety perspective. But from the viewpoint of functionality, it is desirable that the vehicle can move even if some fault occurs in the system. For the above reason, we have worked on adding an analogue input

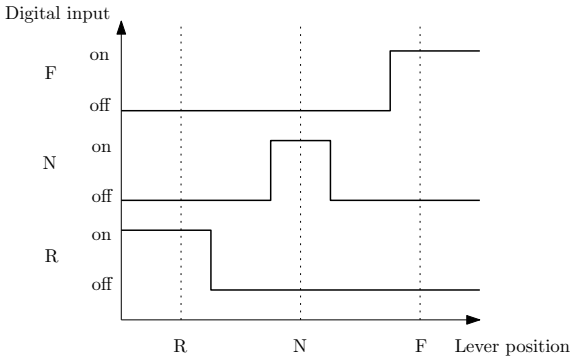

Fig. 2. Electrical Characteristics of the Digital Input Signals

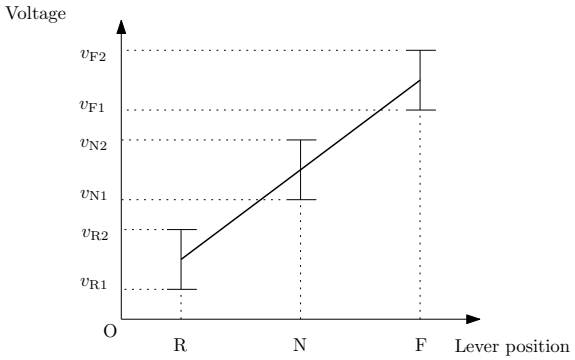

Fig. 3. Electrical Characteristics of the Analogue Input Signal

Table 1. Possible Combination of the Digital Input and the Analogue Input

Direction lever position	Digital input signal	Analogue input signal				
		R	Mid(RN)	N	Mid(FN)	F
		v_{R1}–v_{R2}	v_{R2}–v_{N1}	v_{N1}–v_{N2}	v_{N2}–v_{F1}	v_{F1}–v_{F2}
R	R	◯	–	–	–	–
Mid(RN)	R	×	×	×	–	–
	None	×	×	×	–	–
	N	–	×	×	–	–
N	N	–	–	◯	–	–
Mid(FN)	N	–	–	×	×	–
	None	–	–	×	×	×
	F	–	–	×	×	×
F	F	–	–	–	–	◯

◯ : normal position, × : possible in the middle position, – : impossible

signal to the system to improve redundancy. This not only makes the control specifications complicated, but also makes safety analysis of the system difficult. This motivates us to apply formal methods to our safety analysis process.

Transmission Controller: It detects the position of the direction lever using the digital and the analogue input signals. In detection, it also diagnoses each input signal and takes the proper measures if some fault has occurred. The specifications for detecting the direction lever position are described as follows. Table 2 shows how to detect the lever position by the digital input signals. If only one signal is "on", the signal indicates the lever position. If no or multiple signals are "on", the lever position is determined by a fault measure.
Table 3 specifies the lever position detection by the analogue input signal. In case the signal indicates a voltage of a middle position of the lever, the position is recognised as N. If it is out of range (too low or high), the lever position is determined by a fault measure.

We identify the following six fault modes possible in the system.

F1: Digital input: open-circuit or short-circuit to ground (minor fault)
F2: Digital input: open-circuit or short-circuit to ground (severe fault)
F3: Digital input: short-circuit to power
F4: Analogue input: open-circuit or short-circuit to ground
F5: Analogue input: short-circuit to power
F6: Analogue input: internal circuit fault

A way of detecting each fault and a measure which should be taken in case the fault is detected are specified. As an example, we show the specification of F1 in Table 4. The table instructs how to detect the fault and what to do in case of the fault. Specifically, if "error state" holds, the system starts to detect the fault and takes a "measure before fault confirmation". If the error state has continued for "fault detecting time", it is confirmed that the fault has occurred, and a "measure after fault confirmation" is taken. After that, if "recovery state" holds continuously for "recovery detecting time", it is confirmed that the fault has recovered, and a "measure after fault recovery" is taken.
Some fault modes deserve comment. Both F1 and F2 indicate open-circuit or short-circuit to ground of the digital input signals. For the reason that we cannot distinguish the fault from the case in which the lever is being held in the middle position (as mentioned above), the fault is detected in two-stage manner. The fault detecting time of F1 (minor fault) is set to a value less than that of F2 (severe fault) so that F1 is detected earlier than F2. In case the occurrence of F1 has been confirmed, the system gives an alarm. If the operator puts the lever back to the normal position (if possible) and the error state no longer holds, the fault F1 recovers and the alarm stops. If the operator does not put the lever back or open-circuit has actually occurred, the fault F2 is confirmed eventually.[5]

[5] Strictly speaking, F1 is not regarded as a fault in the system, though this is not directly relevant to our case study.

Table 2. Detection of the Direction Lever Position by the Digital Input Signals

No.	Digital input signals			Detected lever position
	R	N	F	
1	◯			R
2		◯		N
3			◯	F
4				Undefined. Obey fault detection and measure.
5	◯	◯		Undefined. Obey fault detection and measure.
6	◯		◯	
7		◯	◯	
8	◯	◯	◯	

◯ : on, blank: off

Table 3. Detection of the Direction Lever Position by the Analogue Input Signal

No.	Analogue input voltage (A_{in})	Detected lever position
1	$A_{\mathrm{in}} < v_{\mathrm{R1}}$	Undefined. Obey fault detection and measure.
2	$v_{\mathrm{R1}} \leq A_{\mathrm{in}} \leq v_{\mathrm{R2}}$	R
3	$v_{\mathrm{R2}} < A_{\mathrm{in}} < v_{\mathrm{N1}}$	N (middle position between R and N)
4	$v_{\mathrm{N1}} \leq A_{\mathrm{in}} \leq v_{\mathrm{N2}}$	N (normal position)
5	$v_{\mathrm{N2}} < A_{\mathrm{in}} < v_{\mathrm{F1}}$	N (middle position between F and N)
6	$v_{\mathrm{F1}} \leq A_{\mathrm{in}} \leq v_{\mathrm{F2}}$	F
7	$v_{\mathrm{F2}} < A_{\mathrm{in}}$	Undefined. Obey fault detection and measure.

Table 4. An Example of Fault Detection and Measure

Fault mode	Digital input: open-circuit or short-circuit to ground
Error state	All digital input signals F, N and R are “off”.
Fault detecting time	$t_{1\mathrm{f}}$ seconds
Measure before fault confirmation	Keep the detected lever position before the error state.
Measure after fault confirmation	Obey the detected lever position by the analogue input.
Recovery state	Only one digital input signal F, N or R is “on”.
Recovery detecting time	$t_{1\mathrm{r}}$ seconds
Measure after fault recovery	Keep obeying the detected lever position by the analogue input until it becomes consistent with that by the digital input. After the consistency, obey the detected lever position by the digital input.

The fault F6 indicates impossible combination of the digital and the analogue input signals specified in Table 1. The system detects the situation as a fault of an internal circuit.

In short, the specifications are summarised as follows:

1. If the digital input signals are normal, the position detected by the digital input signals is valid.
2. If the digital input signals have a fault, the position detected by the analogue input signal is valid.
3. Even if the digital input signals have recovered from the fault, the position detected by the analogue input signal is still valid until the detected positions by the digital and the analogue input signals are consistent with each other. Once the consistency is reached, the position detected by the digital input signals becomes valid.
4. If both the digital and the analogue input signals respectively have a fault, the lever position is recognised as N.

2.2 Safety Requirements

We informally describe the safety requirements to be satisfied by the control system as follows:

R1: If any fault occurs in the system, the detected position of the direction lever must be consistent with the actual lever position or recognised as neutral (N), i.e. if the actual position is F, the detected position must be F or N; if the actual position is N, the detected position must be N; and if the actual position is R, the detected position must be R or N.

R2: If any fault occurs in the system, the detected position of the direction lever must not change to F or R without lever manipulation by the operator of the vehicle.

The above requirement R1 inhibits the vehicle from moving in the opposite direction of the lever position or moving while the lever is set to N. It is allowable from the safety viewpoint that the lever position is recognised as N by a fault measure while the actual position is not N. The requirement R2 inhibits the vehicle from moving suddenly as opposed to the operator's intention.

3 Formal Modelling of the System

We model the control system described informally in the previous section using an object-oriented formal modelling notation VDM++ [2], because the notions of the object-oriented method, e.g. inheritance, encapsulation etc., seem to be useful also in formal specification description. The overview of the model (class diagram) is illustrated in Fig. 4. In the following subsections, we explain the characteristics of the model and describe each class in detail. The full VDM++ model is given in Appendix A.

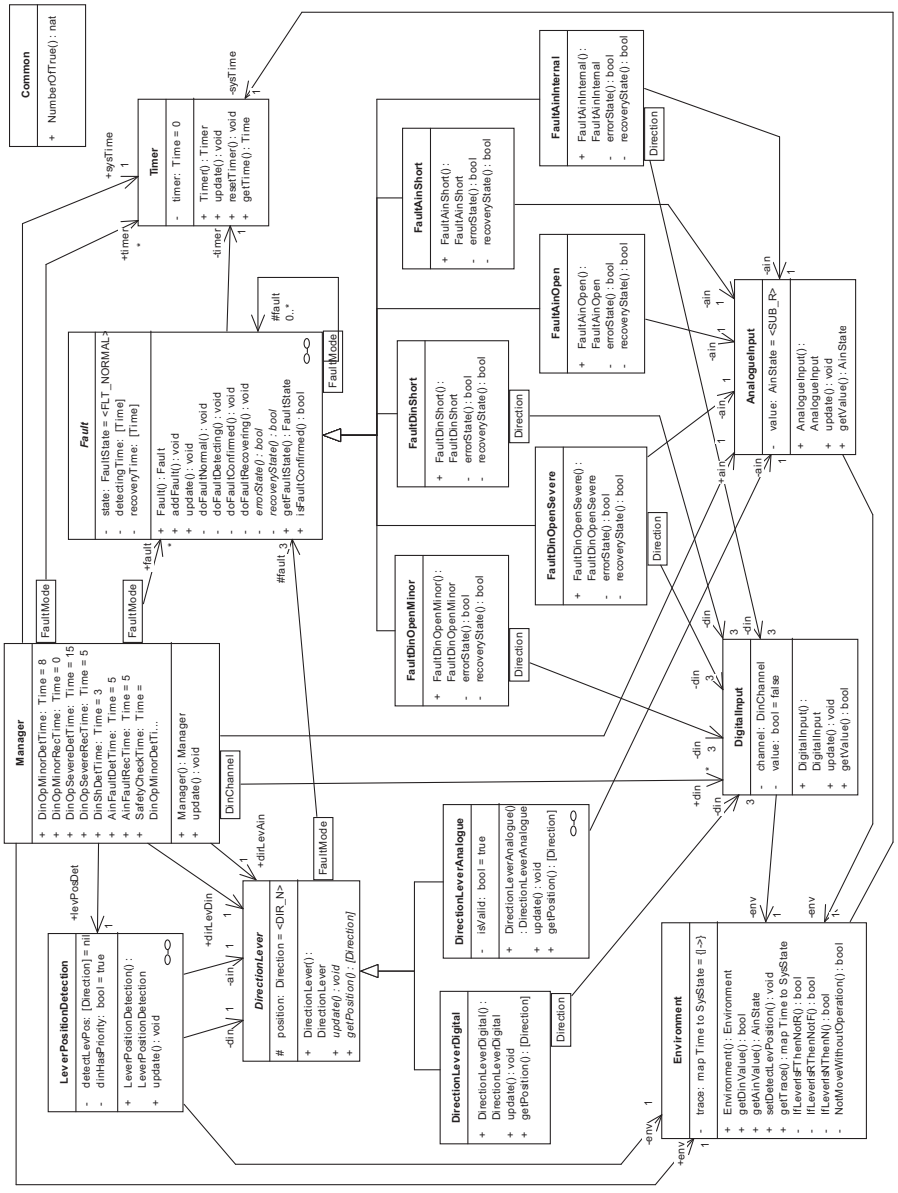

Fig. 4. Overview of the Model

3.1 Periodic Execution Architecture

The actual transmission controller is a periodic real-time system with a certain period, that is, a specific program is executed repeatedly every time unit. In order to reflect such a mechanism into the model, we introduce a periodic execution architecture, referred to as "time-triggered object-oriented model" in [6]. In this architecture, each class has a method `update`, in which its attributes are updated.[6] The `Manager` class, which controls the model execution, calls the `update` method of each class in a specified order as it increments the system timer by one time unit. That is, each class is updated once per time unit.

3.2 Fault Framework

In the model, we abstract the notion of faults (e.g. open- or short-circuit etc.) as a class containing a state represented by the state transition diagram of Fig. 5. The figure says:

1. As long as the device is normally working, the state stays in `NORMAL`.
2. If the error state holds, the state goes into `DETECTING`.
3. If the error state continues for a specified time (`detectingTime`), the fault is confirmed (`CONFIRMED`). If the error state no longer holds while in `DETECTING`, the state goes back to `NORMAL`.
4. If the recovery state holds while in `CONFIRMED`, the state goes into `RECOVERING`.
5. If the recovery state continues for a specified time (`recoveryTime`), the fault recovers (`NORMAL`). If the recovery state no longer holds while in `RECOVERING`, the state goes back to `CONFIRMED`.

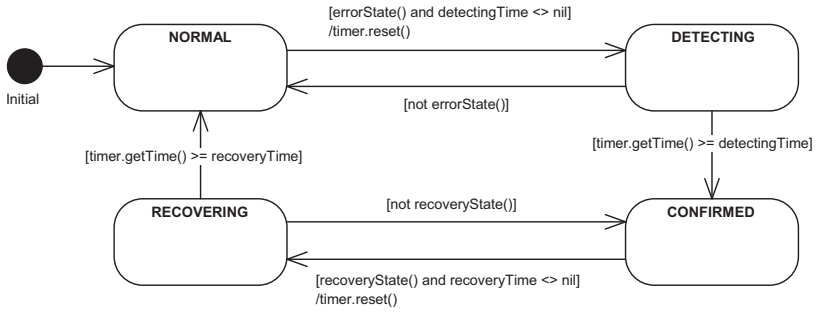

Fig. 5. State Transition Diagram of Fault

[6] The method corresponds to a `Step` in a VDM-RT context [3, 4]. But we use the name `update` in this paper according to [6] and our convention.

3.3 Detailed Description of Each Class

Common: Types and a function commonly used by various classes are defined in this class. It is inherited by all the other classes to make the model description simple, though in Fig. 4, inheritance arrows are hidden for legibility. Some of principal types are illustrated here.

```
types
public Time = nat;

public Direction = <DIR_F> | <DIR_N> | <DIR_R>;

public AinState = Direction | <SUB_R>  | <MID_RN>
                            | <MID_FN> | <SUPER_F>;
```

`Time` is defined as natural number (`nat`), denoting discrete time steps in the control system. The notion of direction (F, N and R) is defined as a union type of three quote types. We abstract the analogue input signal using a union type `AinState` instead of expressing it in voltage as follows:

<SUB_R>: $A_{\mathrm{in}} < v_{\mathrm{R1}}$
<DIR_R>: $v_{\mathrm{R1}} \leq A_{\mathrm{in}} \leq v_{\mathrm{R2}}$
<MID_RN>: $v_{\mathrm{R2}} < A_{\mathrm{in}} < v_{\mathrm{N1}}$
<DIR_N>: $v_{\mathrm{N1}} \leq A_{\mathrm{in}} \leq v_{\mathrm{N2}}$
<MID_FN>: $v_{\mathrm{N2}} < A_{\mathrm{in}} < v_{\mathrm{F1}}$
<DIR_F>: $v_{\mathrm{F1}} \leq A_{\mathrm{in}} \leq v_{\mathrm{F2}}$
<SUPER_F>: $v_{\mathrm{F2}} < A_{\mathrm{in}}$

Manager: This class controls the overall model. It is responsible for constructing all instances of the model and executing them. All instances are created in the constructor `Manager`, and the operation `update` calls the same operations of all the other classes in a specified order (usually from lower to upper level classes). Specific values of fault and recovery detecting times, which are used to instantiate each fault class, are defined in this class. Note that they do not express the actual values of the control system. They are properly chosen to make the model feasible in practicable time and to simplify creation of test data. In our case, magnitude relation between the values is significant, and the value itself is not.

Timer: A generic timer class containing one instance variable `timer`, which is incremented by one step time in the operation `update`. The operation `resetTimer` sets the `timer` to zero. Instances of the class are used as a system timer, which denotes time evolution in the model, and a timer for each fault class, which is used to detect the fault.

Environment: As advocated in [3, 4], the components outside the controller are modelled as a class `Environment`. It provides input to the controller, i.e. the digital input signals F, N, R and the analogue input signal, and receives the detected lever

position from the controller as output of the system. It also has the actual lever position used for safety requirement check. This information is put into one record type `SysState`, and `trace`, a mapping from `Time` to `SysState`, is defined as an instance variable of the class, indicating time series of input and output data of the system.

```
types
public SysState :: dinF : bool
                   dinN : bool
                   dinR : bool
                   ain : AinState
                   levPos : LeverPosition
                   detectLevPos : [Direction];

instance variables
private trace : map Time to SysState := {|->};
```

The operations `getDinValue` and `getAinValue` respectively return the digital and the analogue input value at the current system time, and another operation `setDetectLevPosition` is called to set the detected lever position to `trace`.

DigitalInput and AnalogueInput: These two classes represent input channels of the controller, serving as interface to `Environment`. In the operation `update`, each class gets its current value from `Environment` and stores it in its instance variable `value`, which is used by the other upper level classes, that is, `Fault` and `DirectionLever` (described below). Three instances of the `DigitalInput` class, namely the digital input F, N and R, are created by the `Manager` class.

Fault: A superclass of the following six subclasses. It implements the state transition of fault described in Sect. 3.2, and provides the other classes with the state information. Fault detecting time and recovery time are respectively declared as an instance variable of optional type `[Time]`. The value `nil` means that the fault is undetectable or unrecoverable, respectively. The operations `errorState` and `recoveryState` are implemented in each subclass representing each fault mode (F1 to F6 in Sect. 2.1), because they are different depending on the fault modes. Note that these operations are declared as abstract methods.

- **FaultDinOpenMinor:**
  Digital input: open-circuit or short-circuit to ground (F1)
- **FaultDinOpenSevere:**
  Digital input: open-circuit or short-circuit to ground (F2)
- **FaultDinShort:**
  Digital input: short-circuit to power (F3)
- **FaultAinOpen:**
  Analogue input: open-circuit or short-circuit to ground (F4)

- **FaultAinShort:**
  Analogue input: short-circuit to power (F5)
- **FaultAinInternal:**
  Analogue input: internal circuit fault (F6)

DirectionLever: A superclass of the following two subclasses. In the `update` operation (implemented in the subclass), it updates its instance variable `position`, which denotes the position of the direction lever detected by the digital or the analogue input depending on its subclass. Users of this class do not need to take into account by which input the position is detected.

- **DirectionLeverDigital:**
  This class detects the direction lever position using the digital input and its fault information according to the control specifications described in Sect. 2.1. The operation `getPosition` returns `nil` if at least one digital input fault is confirmed, otherwise it returns the detected lever position.
- **DirectionLeverAnalogue:**
  This class detects the direction lever position using the analogue input and its fault information according to the control specifications described in Sect. 2.1. It has an instance variable `isValid`, which is used to realise the measures after fault recovery of F4 and F5. Briefly speaking, the variable is set to `false` if an analogue input fault is confirmed, and held `false` unless the normal N position is detected after fault recovery. As long as the variable is `false`, the lever position is regarded as N.

LeverPositionDetection: In the operation `update` in this class, the conclusive direction lever position is determined using the positions detected by the digital and the analogue input respectively, and the result is set to `Environment`. This class has an instance variable `dinHasPriority`, which is used to realise the measures after fault recovery of F1 and F2. The variable is set to `false` if a digital input fault is confirmed, and held `false` unless the detected lever positions by the digital and the analogue input are consistent with each other after fault recovery. As long as the variable is `false`, the lever position detected by the analogue input is valid.

3.4 Safety Requirements

Safety requirements are described in the `Environment` class because they require to access the lever position information enclosed in the class. Specifically, the requirements are formalised as postconditions of the operation `setDetectLevPosition`, which the `LeverPositionDetection` class calls to set the conclusive detected lever position to the `Environment` class. The postconditions are divided into four sub-operations (see Appendix A.4). We comment on the first and the last ones for illustration:

```
private IfLeverIsFThenNotR: () ==> bool
IfLeverIsFThenNotR() ==
```

```
  let curTime = sysTime.getTime()
  in
    return
    (((curTime >= Manager`SafetyCheckTime) and
      (forall t in set
          {curTime - Manager`SafetyCheckTime,..., curTime} &
           trace(t).levPos = <DIR_F>))
    => trace(curTime).detectLevPos <> <DIR_R>)
post RESULT;
...
private NotMoveWithoutOperation: () ==> bool
NotMoveWithoutOperation() ==
  let curTime = sysTime.getTime()
  in
    return
    (((curTime >= Manager`SafetyCheckTime) and
      (forall t in set
          {curTime - Manager`SafetyCheckTime,..., curTime-1} &
          (trace(t).levPos = trace(curTime).levPos and
           trace(t).detectLevPos = <DIR_N>)))
    => trace(curTime).detectLevPos = <DIR_N>)
post RESULT;
```

The operation `IfLeverIsFThenNotR` insists: if the lever has been set in the position F for a specified time, the detected lever position at the current time should not be R, which corresponds to the safety requirement R1 in Sect. 2.2. The operation `NotMoveWithoutOperation` insists: if the lever has not been manipulated and the detected lever position has been N for a specified time (until one step time before), the detected lever position at the current time should also be N, corresponding to the safety requirement R2.

4 Validation and Safety Analysis

In our approach, validation process is composed of two phases: unit testing and system testing. The former tests each class of the model, while the latter deals with the whole system. A testing framework VDMUnit [2, Chap. 9] is used for both testing phases. Various time series of input data for the `Environment` class (so called test scenarios) are elaborated. Using assert functions of VDMUnit, we check if a return value of each method of each class (in the case of unit testing) or the detected lever position (in the case of system testing) is consistent with the expected value at the time as we execute the model periodically. An example of the test cases is given below:

```
class SystemTest1 is subclass of TestCase, Environment

operations
...
```

```
public runTest : () ==> ()
runTest() ==
(
    let testInData = {t |-> testData(t).inData |
                       t in set dom testData}
    in (
        dcl mgr : Manager := new Manager(testInData);
        for t = 0 to (card dom testData - 1)
        do (
            mgr.update();
            assertTrue("t=" ^ VDMUtil`val2seq_of_char[nat](t) ^
                       ", failed.",
                       mgr.env.getTrace()(t).detectLevPos =
                       testData(t).expectVal)
        )
    )
);

types
private TestData :: inData : SysState
                    expectVal : [Direction];
values
-- time to (input data, expected value of trace(t).detectLevPos)
private testData: map Time to TestData =
{
   0 |-> mk_TestData(mk_SysState(false, true, false,
                     <DIR_N>, <DIR_N>, nil), <DIR_N>),
   1 |-> mk_TestData(mk_SysState(false, false, false,
                     <MID_FN>, <MID_FN_>, nil), <DIR_N>),
   2 |-> mk_TestData(mk_SysState(true, false, false,
                     <DIR_F>, <DIR_F>, nil), <DIR_F>),
...
};

end SystemTest1
```

The value `testData` denotes the time series of input data and expected values of the test scenario. We considered various scenarios: for example, normal lever operation without faults, a case in which open-circuit of the digital input signal F occurs and then it recovers, and so on. As for the system testing, we executed 14 test scenarios in total.

As a result, we have confirmed that the model behaved as expected for all the input data series elaborated. The test coverage information generated by Overture indicates that almost all statements of the model are tested, except for a part of the following two operations: `DirectionLeverDigital`update` (coverage is 98.6%) and `Fault`doFaultNormal` (89.4%). But these statements can never been executed under the current specifications of fault detection and the data settings. Therefore we conclude that virtually every part of the model is tested.

In the validation process, however, we realised that one of the safety requirements was not satisfied (a postcondition was violated) for certain input data series. This occurs in the following manner:

1. The direction lever is in the middle of the positions F and N, and no digital input signals are "on".
2. The analogue input signal indicates the position F (this meets the specifications of Table 1).
3. The digital input signal N periodically short-circuits to power with a period less than the fault detecting time, that is, the signal N alternates between "on" and "off" in a short period of time. The controller is not able to detect the fault (because the time in which the signal N remains "on" or "off" respectively is too short for the controller to detect the fault) and recognises the lever position as N.
4. In these situations, if the short-circuit of the digital input signal N recovers, that is, the signal N settles down to "off", the analogue input signal (recognised as F) becomes valid by a fault measure. This indicates that the detected lever position changes from N to F without manipulation by the operator, which violates the safety requirement R2 in Sect. 2.2.

However, the above case could never happen in reality because it is caused by nothing but a coincidence of several rare accidents. Nevertheless, it seems to be one of the advantages of formal modelling that the above phenomenon which could hardly be predicted in a manual fashion has been discovered.

5 Conclusion

In this paper, we have reported on a case study of applying a formal modelling technique to safety analysis of an embedded control system for construction equipment, and we have also presented a fault framework, which makes it possible to encapsulate a fault detection mechanism into the `Fault` class and separate it from the other control logics.

The validation of the model revealed that, under particular conditions, the exemplified system failed to satisfy certain safety requirement which had been considered to be satisfied, though it could rarely happen in reality. This demonstrates the advantage of the formal modelling and validation techniques.

The control system treated in this paper is only a part of the entire system. In future work, we will apply the technique described above to a larger scale system. On the other hand, in our test scenario based approach, the result considerably depends on the quality of the scenarios. It might be sheer luck that we discovered the violation of the safety requirements. We will challenge formal verification of the model with the help of another verification tool, e.g. UPPAAL, in order to investigate if there exists another case which violates the safety requirements.

Acknowledgements: The author would like to thank John Fitzgerald and Ken Pierce for fruitful discussions. The work has been supported by Komatsu Ltd. Especially, the author is grateful to Shuuki Akushichi and Yasunori Ohkura for their valuable comments on a draft.

References

1. Fitzgerald, J., Larsen, P.G.: Modelling Systems: Practical Tools and Techniques in Software Development, Second Edition, Cambridge University Press (2009)
2. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object-Oriented Systems, Springer, London (2005)
3. Larsen, P.G., Fitzgerald, J., Wolff, S.: Methods for the Development of Distributed Real-Time Embedded Systems using VDM, Int J Software Informatics, vol. 3, no. 2-3, pp. 305–341 (2009)
4. Larsen, P.G., Wolff, S., Battle, N., Fitzgerald, J., Pierce, K.: Development Process of Distributed Embedded Systems using VDM, Overture Technical Report Series, no. TR-006 (2010)
5. McDermott, R.E., Mikulak, R.J., Beauregard, M.R.: The Basics of FMEA, Productivity Press (1996)
6. Yokoyama, T., Naya, H., Narisawa, F., Kuragaki, S., Nagaura, W., Imai, T., Suzuki, S.: A Development Method of Time-Triggered Object-Oriented Software for Embedded Control Systems (in Japanese), IEICE Trans. D-I, vol. J84-D-I, no. 4, pp. 338–349 (2001)

A VDM++ Model for the Control System

A.1 The Common class

```
class Common

types
public Time = nat;

-- Digital input channel
public DinChannel = <CH_LEV_F>  -- Direction lever F
                  | <CH_LEV_N>  -- Direction lever N
                  | <CH_LEV_R>; -- Direction lever R

public Direction = <DIR_F> | <DIR_N> | <DIR_R>;

-- Analogue input state
public AinState = Direction
                | <SUB_R>    -- Lower than minimum
                | <MID_RN>   -- Middle between R & N
                | <MID_FN>   -- Middle between F & N
                | <SUPER_F>; -- Higher than maximum

-- Physical position of the direction lever
public LeverPosition =
                  Direction
                | <MID_RNR>  -- Middle between R & N with Din R
                | <MID_RN_>  -- Middle between R & N without Din
                | <MID_RNN>  -- Middle between R & N with Din N
                | <MID_FNN>  -- Middle between F & N with Din N
```

```
                | <MID_FN_>  -- Middle between F & N without Din
                | <MID_FNF>; -- Middle between F & N with Din F

public FaultMode = <DIN_OPEN_MINOR>
                 | <DIN_OPEN_SEVERE>
                 | <DIN_SHORT>
                 | <AIN_OPEN>
                 | <AIN_SHORT>
                 | <AIN_INTERNAL>;

functions
-- Count the number of 'true' in a sequence of boolean values
public NumberOfTrue : seq of bool -> nat
NumberOfTrue(args) ==
    len [args(i) | i in set inds args & args(i)];

end Common
```

A.2 The Manager class

```
class Manager is subclass of Common

instance variables
-- Environment is declared as public
-- because it is referred to by test cases
public  env : Environment;
private sysTime : Timer;
private timer : map FaultMode to Timer;
private din : map DinChannel to DigitalInput;
private ain : AnalogueInput;
private fault : map FaultMode to Fault;
private dirLevDin : DirectionLever;
private dirLevAin : DirectionLever;
private levPosDet : LeverPositionDetection;

values
-- Detecting or recovery time of faults
-- Declared as public because they are referred to by test cases
public DinOpMinorDetTime:  Time = 8;
public DinOpMinorRecTime:  Time = 0;
public DinOpSevereDetTime: Time = 15;
public DinOpSevereRecTime: Time = 5;
public DinShDetTime:       Time = 3;
public AinFaultDetTime:    Time = 5;
public AinFaultRecTime:    Time = 5;
```

```
-- Time for safety requirements
public SafetyCheckTime: Time = (DinOpMinorDetTime +
                                AinFaultDetTime);

operations
public Manager : map Time to Environment`SysState ==> Manager
Manager(mTrace) ==
(
    -- Instantiate all objects
    sysTime := new Timer();
    timer := {fMode |-> new Timer() |
              fMode in set {<DIN_OPEN_MINOR>, <DIN_OPEN_SEVERE>,
                            <DIN_SHORT>, <AIN_OPEN>,
                            <AIN_SHORT>, <AIN_INTERNAL>}};
    env := new Environment(mTrace, sysTime);
    din := {ch |-> new DigitalInput(ch, env) |
            ch in set {<CH_LEV_F>, <CH_LEV_N>, <CH_LEV_R>}};
    ain := new AnalogueInput(env);
    let mapDin = {<DIR_F> |-> din(<CH_LEV_F>),
                  <DIR_N> |-> din(<CH_LEV_N>),
                  <DIR_R> |-> din(<CH_LEV_R>)}
    in
    (
        fault := {<DIN_OPEN_MINOR> |->
                        new FaultDinOpenMinor(
                                DinOpMinorDetTime,
                                DinOpMinorRecTime,
                                timer(<DIN_OPEN_MINOR>),
                                mapDin),
                  <DIN_OPEN_SEVERE> |->
                        new FaultDinOpenSevere(
                                DinOpSevereDetTime,
                                DinOpSevereRecTime,
                                timer(<DIN_OPEN_SEVERE>),
                                mapDin, ain),
                  <DIN_SHORT> |->
                        new FaultDinShort(
                                DinShDetTime,
                                nil,
                                timer(<DIN_SHORT>),
                                mapDin),
                  <AIN_OPEN> |->
                        new FaultAinOpen(
                                AinFaultDetTime,
                                AinFaultRecTime,
                                timer(<AIN_OPEN>),
                                ain),
                  <AIN_SHORT> |->
                        new FaultAinShort(
                                AinFaultDetTime,
```

```
                                AinFaultRecTime,
                                timer(<AIN_SHORT>),
                                ain),
                    <AIN_INTERNAL> |->
                         new FaultAinInternal(
                                AinFaultDetTime,
                                nil,
                                timer(<AIN_INTERNAL>),
                                mapDin, ain)};

        -- Add association from <AIN_INTERNAL> to <DIN_SHORT>
        fault(<AIN_INTERNAL>).addFault(
               {<DIN_SHORT> |-> fault(<DIN_SHORT>)});

        dirLevDin := new DirectionLeverDigital(
                           {fMode |-> fault(fMode) |
                            fMode in set {<DIN_OPEN_MINOR>,
                                          <DIN_OPEN_SEVERE>,
                                          <DIN_SHORT>}},
                           mapDin)
    );
    dirLevAin := new DirectionLeverAnalogue(
                           {fMode |-> fault(fMode) |
                            fMode in set {<AIN_OPEN>,
                                          <AIN_SHORT>,
                                          <AIN_INTERNAL>}},
                           ain);
    levPosDet := new LeverPositionDetection(
                           dirLevDin, dirLevAin, env);
);

public update : () ==> ()
update() ==
(
    for all x in set dom din do din(x).update();
    ain.update();
    for all x in set dom fault do fault(x).update();
    dirLevDin.update();
    dirLevAin.update();
    levPosDet.update();
    sysTime.update();
    for all x in set dom timer do timer(x).update();
);

end Manager
```

A.3 The Timer class

```
class Timer is subclass of Common

instance variables
private timer : Time := 0;

operations
public Timer : () ==> Timer
Timer() ==
    skip;

public update : () ==> ()
update() ==
    timer := timer + 1;

public resetTimer : () ==> ()
resetTimer() ==
    timer := 0;

public getTime : () ==> Time
getTime() ==
    return timer;

end Timer
```

A.4 The Environment class

```
class Environment is subclass of Common

types
public SysState :: dinF : bool     -- Digital input F
                   dinN : bool     -- Digital input N
                   dinR : bool     -- Digital input R
                   ain : AinState -- Analogue input
                   levPos : LeverPosition
                   -- Physical lever position
                   detectLevPos : [Direction];
                   -- Detected lever position

instance variables
-- Time series of input/output data
private trace : map Time to SysState := {|->};
private sysTime : Timer;
```

```
operations
public Environment : map Time to SysState * Timer
                     ==> Environment
Environment(inData, pTimer) ==
(
    trace := inData;
    sysTime := pTimer
);

public getDinValue : DinChannel ==> bool
getDinValue(ch) ==
(
    let currentTime = sysTime.getTime()
    in
        cases ch:
            <CH_LEV_F> -> return trace(currentTime).dinF,
            <CH_LEV_N> -> return trace(currentTime).dinN,
            <CH_LEV_R> -> return trace(currentTime).dinR
        end
)
pre
    sysTime.getTime() in set dom trace;

public getAinValue : () ==> AinState
getAinValue() ==
    return trace(sysTime.getTime()).ain
pre
    sysTime.getTime() in set dom trace;

public setDetectLevPosition : Direction ==> ()
setDetectLevPosition(dir) ==
    trace(sysTime.getTime()).detectLevPos := dir
pre
    sysTime.getTime() in set dom trace
post
    -- Safety requirements
    IfLeverIsFThenNotR() and
    IfLeverIsRThenNotF() and
    IfLeverIsNThenN() and
    NotMoveWithoutOperation();

public getTrace : () ==> map Time to SysState
getTrace() ==
    return trace;

-- Safety requirements
private IfLeverIsFThenNotR: () ==> bool
IfLeverIsFThenNotR() ==
    let curTime = sysTime.getTime()
    in
```

```
        return
        (((curTime >= Manager`SafetyCheckTime) and
          (forall t in set
             {curTime - Manager`SafetyCheckTime,..., curTime} &
              trace(t).levPos = <DIR_F>))
         => trace(curTime).detectLevPos <> <DIR_R>)
post RESULT;

private IfLeverIsRThenNotF: () ==> bool
IfLeverIsRThenNotF() ==
    let curTime = sysTime.getTime()
    in
        return
        (((curTime >= Manager`SafetyCheckTime) and
          (forall t in set
             {curTime - Manager`SafetyCheckTime,..., curTime} &
              trace(t).levPos = <DIR_R>))
         => trace(curTime).detectLevPos <> <DIR_F>)
post RESULT;

private IfLeverIsNThenN: () ==> bool
IfLeverIsNThenN() ==
    let curTime = sysTime.getTime()
    in
        return
        (((curTime >= Manager`SafetyCheckTime) and
          (forall t in set
             {curTime - Manager`SafetyCheckTime,..., curTime} &
              trace(t).levPos = <DIR_N>))
         => trace(curTime).detectLevPos = <DIR_N>)
post RESULT;

private NotMoveWithoutOperation: () ==> bool
NotMoveWithoutOperation() ==
    let curTime = sysTime.getTime()
    in
        return
        (((curTime >= Manager`SafetyCheckTime) and
          (forall t in set
            {curTime - Manager`SafetyCheckTime,..., curTime-1} &
            (trace(t).levPos = trace(curTime).levPos and
             trace(t).detectLevPos = <DIR_N>)))
         => trace(curTime).detectLevPos = <DIR_N>)
post RESULT;

end Environment
```

A.5 The DigitalInput class

```
class DigitalInput is subclass of Common

instance variables
private channel : DinChannel;
private value : bool := false;
private env : Environment;

operations
public DigitalInput : DinChannel * Environment ==> DigitalInput
DigitalInput(ch, pEnv) ==
(
    channel := ch;
    env := pEnv
);

public update : () ==> ()
update() ==
    value := env.getDinValue(channel);

public getValue : () ==> bool
getValue() ==
    return value;

end DigitalInput
```

A.6 The AnalogueInput class

```
class AnalogueInput is subclass of Common

instance variables
private value : AinState := <SUB_R>;
private env : Environment;

operations
public AnalogueInput : Environment ==> AnalogueInput
AnalogueInput(pEnv) ==
    env := pEnv;

public update : () ==> ()
update() ==
    value := env.getAinValue();

public getValue : () ==> AinState
```

```
getValue() ==
    return value;

end AnalogueInput
```

A.7 The Fault class

```
class Fault is subclass of Common

types
public FaultState = <FLT_NORMAL>
                  | <FLT_DETECTING>
                  | <FLT_CONFIRMED>
                  | <FLT_RECOVERING>;

instance variables
private state : FaultState := <FLT_NORMAL>;
private detectingTime : [Time];
        -- nil means the fault is undetectable
private recoveryTime : [Time];
        -- nil means the fault is unrecoverable
private timer : Timer;
protected fault : map FaultMode to Fault := {|->};

operations
public Fault : [Time] * [Time] * Timer ==> Fault
Fault(detT, recT, pTimer) ==
(
    detectingTime := detT;
    recoveryTime := recT;
    timer := pTimer
);

-- Add association to another Fault to watch
public addFault : map FaultMode to Fault ==> ()
addFault(mFault) ==
    fault := fault ++ mFault;

public update : () ==> ()
update() ==
    cases state:
        <FLT_NORMAL> -> doFaultNormal(),
        <FLT_DETECTING> -> doFaultDetecting(),
        <FLT_CONFIRMED> -> doFaultConfirmed(),
        <FLT_RECOVERING> -> doFaultRecovering()
    end;
```

```
private doFaultNormal : () ==> ()
doFaultNormal() ==
(
    if errorState() and detectingTime <> nil
    then
    (
        timer.resetTimer();
        if detectingTime = 0
        then
            state := <FLT_CONFIRMED>
        else
            state := <FLT_DETECTING>
    )
    else
        skip
)
pre
    state = <FLT_NORMAL>;

private doFaultDetecting : () ==> ()
doFaultDetecting() ==
(
    if not errorState()
    then
        state := <FLT_NORMAL>
    else if timer.getTime() >= detectingTime
    then
        state := <FLT_CONFIRMED>
    else
        skip
)
pre
    state = <FLT_DETECTING>;

private doFaultConfirmed : () ==> ()
doFaultConfirmed() ==
(
    if recoveryState() and recoveryTime <> nil
    then
    (
        timer.resetTimer();
        if recoveryTime = 0
        then
            state := <FLT_NORMAL>
        else
            state := <FLT_RECOVERING>
    )
    else
        skip
```

```
)
pre
    state = <FLT_CONFIRMED>;

private doFaultRecovering : () ==> ()
doFaultRecovering() ==
(
    if not recoveryState()
    then
        state := <FLT_CONFIRMED>
    else if timer.getTime() >= recoveryTime
    then
        state := <FLT_NORMAL>
    else
        skip
)
pre
    state = <FLT_RECOVERING>;

private errorState : () ==> bool
errorState() == is subclass responsibility;

private recoveryState : () ==> bool
recoveryState() == is subclass responsibility;

public getFaultState : () ==> FaultState
getFaultState() ==
    return state;

public isFaultConfirmed : () ==> bool
isFaultConfirmed() ==
    return ((state = <FLT_CONFIRMED>) or
            (state = <FLT_RECOVERING>));

end Fault
```

A.8 The FaultDinOpenMinor class

```
class FaultDinOpenMinor is subclass of Fault

instance variables
private din : map Direction to DigitalInput;

operations
public FaultDinOpenMinor : [Time] * [Time] * Timer *
                           map Direction to DigitalInput
```

```
                            ==> FaultDinOpenMinor
FaultDinOpenMinor(detT, recT, pTimer, mDin) ==
(
    din := mDin;
    Fault(detT, recT, pTimer)
)
pre
    dom mDin = {<DIR_F>, <DIR_N>, <DIR_R>};

private errorState : () ==> bool
errorState() ==
    return NumberOfTrue([din(<DIR_F>).getValue(),
                         din(<DIR_N>).getValue(),
                         din(<DIR_R>).getValue()]) = 0;

private recoveryState : () ==> bool
recoveryState() ==
    return NumberOfTrue([din(<DIR_F>).getValue(),
                         din(<DIR_N>).getValue(),
                         din(<DIR_R>).getValue()]) = 1;

end FaultDinOpenMinor
```

A.9 The FaultDinOpenSevere class

```
class FaultDinOpenSevere is subclass of Fault

instance variables

private din : map Direction to DigitalInput;
private ain : AnalogueInput;

operations

public FaultDinOpenSevere : [Time] * [Time] * Timer *
                            map Direction to DigitalInput *
                            AnalogueInput
                            ==> FaultDinOpenSevere
FaultDinOpenSevere(detT, recT, pTimer, mDin, pAin) ==
(
    din := mDin;
    ain := pAin;
    Fault(detT, recT, pTimer)
)
pre
    dom mDin = {<DIR_F>, <DIR_N>, <DIR_R>};
```

```
private errorState : () ==> bool
errorState() ==
    let ainValue = ain.getValue()
    in
    (
        return (NumberOfTrue([din(<DIR_F>).getValue(),
                              din(<DIR_N>).getValue(),
                              din(<DIR_R>).getValue()]) = 0
                and
                 (ainValue = <DIR_F> or
                  ainValue = <DIR_N> or
                  ainValue = <DIR_R>))
    );

private recoveryState : () ==> bool
recoveryState() ==
    return NumberOfTrue([din(<DIR_F>).getValue(),
                         din(<DIR_N>).getValue(),
                         din(<DIR_R>).getValue()]) = 1;

end FaultDinOpenSevere
```

A.10 The FaultDinShort class

```
class FaultDinShort is subclass of Fault

instance variables
private din : map Direction to DigitalInput;

operations
public FaultDinShort : [Time] * [Time] * Timer *
                       map Direction to DigitalInput
                       ==> FaultDinShort
FaultDinShort(detT, recT, pTimer, mDin) ==
(
    din := mDin;
    Fault(detT, recT, pTimer)
)
pre
    dom mDin = {<DIR_F>, <DIR_N>, <DIR_R>};

private errorState : () ==> bool
errorState() ==
    return NumberOfTrue([din(<DIR_F>).getValue(),
                         din(<DIR_N>).getValue(),
```

```
                        din(<DIR_R>).getValue()]) > 1;

private recoveryState : () ==> bool
recoveryState() ==
    return false;

end FaultDinShort
```

A.11 The FaultAinOpen class

```
class FaultAinOpen is subclass of Fault

instance variables
private ain : AnalogueInput;

operations
public FaultAinOpen : [Time] * [Time] * Timer * AnalogueInput
                      ==> FaultAinOpen
FaultAinOpen(detT, recT, pTimer, pAin) ==
(
    ain := pAin;
    Fault(detT, recT, pTimer)
);

private errorState : () ==> bool
errorState() ==
    return ain.getValue() = <SUB_R>;

private recoveryState : () ==> bool
recoveryState() ==
    return ain.getValue() <> <SUB_R>;

end FaultAinOpen
```

A.12 The FaultAinShort class

```
class FaultAinShort is subclass of Fault

instance variables
private ain : AnalogueInput;

operations
public FaultAinShort : [Time] * [Time] * Timer * AnalogueInput
```

```
                        ==> FaultAinShort
FaultAinShort(detT, recT, pTimer, pAin) ==
(
    ain := pAin;
    Fault(detT, recT, pTimer)
);

private errorState : () ==> bool
errorState() ==
    return ain.getValue() = <SUPER_F>;

private recoveryState : () ==> bool
recoveryState() ==
    return ain.getValue() <> <SUPER_F>;

end FaultAinShort
```

A.13 The FaultAinInternal class

```
class FaultAinInternal is subclass of Fault

instance variables
private din : map Direction to DigitalInput;
private ain : AnalogueInput;

operations
public FaultAinInternal : [Time] * [Time] * Timer *
                          map Direction to DigitalInput *
                          AnalogueInput
                          ==> FaultAinInternal
FaultAinInternal(detT, recT, pTimer, mDin, pAin) ==
(
    din := mDin;
    ain := pAin;
    Fault(detT, recT, pTimer)
)
pre
    dom mDin = {<DIR_F>, <DIR_N>, <DIR_R>};

private errorState : () ==> bool
errorState() ==
    let dinValueF = din(<DIR_F>).getValue(),
        dinValueN = din(<DIR_N>).getValue(),
        dinValueR = din(<DIR_R>).getValue(),
        ainValue = ain.getValue()
    in
```

```
    (
        return ((dinValueF and not dinValueN and
                 not dinValueR and
                 (ainValue = <DIR_R> or ainValue = <MID_RN>))
                or
                (not dinValueF and not dinValueN and
                 dinValueR and
                 (ainValue = <DIR_F> or ainValue = <MID_FN>))
                or
                (not dinValueF and dinValueN and
                 not dinValueR and
                 (ainValue = <DIR_F> or ainValue = <DIR_R>)))
               and not fault(<DIN_SHORT>).isFaultConfirmed()
    )
pre
    <DIN_SHORT> in set dom fault;

private recoveryState : () ==> bool
recoveryState() ==
    return false;

end FaultAinInternal
```

A.14 The DirectionLever class

```
class DirectionLever is subclass of Common

instance variables
protected position : Direction := <DIR_N>;
protected fault : map FaultMode to Fault;

operations
public DirectionLever : map FaultMode to Fault
                        ==> DirectionLever
DirectionLever(mFault) ==
    fault := mFault;

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

public getPosition : () ==> [Direction]
getPosition() == is subclass responsibility;

end DirectionLever
```

A.15 The DirectionLeverDigital class

```
class DirectionLeverDigital is subclass of DirectionLever

instance variables
private din : map Direction to DigitalInput;

operations
public DirectionLeverDigital : map FaultMode to Fault *
                               map Direction to DigitalInput
                               ==> DirectionLeverDigital
DirectionLeverDigital(mFault, mDin) ==
(
    din := mDin;
    DirectionLever(mFault)
)
pre
    dom mFault = {<DIN_OPEN_MINOR>, <DIN_OPEN_SEVERE>,
                  <DIN_SHORT>} and
    dom mDin = {<DIR_F>, <DIR_N>, <DIR_R>};

public update : () ==> ()
update() ==
(
    -- Detect the lever position by the digital input.
    -- In case the input has a fault,
    -- the position is not updated.
    if fault(<DIN_OPEN_MINOR>).getFaultState() = <FLT_NORMAL>
       and
       fault(<DIN_OPEN_SEVERE>).getFaultState() = <FLT_NORMAL>
       and
       fault(<DIN_SHORT>).getFaultState() = <FLT_NORMAL>
    then
    (
        let dinValueF = din(<DIR_F>).getValue(),
            dinValueN = din(<DIR_N>).getValue(),
            dinValueR = din(<DIR_R>).getValue()
        in
        (
            if dinValueF and
               not dinValueN and
               not dinValueR
            then
                position := <DIR_F>
            else if not dinValueF and
                    dinValueN and
                    not dinValueR
            then
                position := <DIR_N>
```

```
            else if not dinValueF and
                    not dinValueN and
                    dinValueR
            then
                position := <DIR_R>
            else
                skip
        )
    )
    else
        skip;
);

-- Return nil if at least one digital input fault is confirmed,
-- otherwise return detected lever position
public getPosition : () ==> [Direction]
getPosition() ==
    if fault(<DIN_OPEN_MINOR>).isFaultConfirmed() or
       fault(<DIN_OPEN_SEVERE>).isFaultConfirmed() or
       fault(<DIN_SHORT>).isFaultConfirmed()
    then
        return nil
    else
        return position;

end DirectionLeverDigital
```

A.16 The DirectionLeverAnalogue class

```
class DirectionLeverAnalogue is subclass of DirectionLever

instance variables
private isValid : bool := true;
private ain : AnalogueInput;

operations
public DirectionLeverAnalogue : map FaultMode to Fault *
                                AnalogueInput
                                ==> DirectionLeverAnalogue
DirectionLeverAnalogue(mFault, pAin) ==
(
    ain := pAin;
    DirectionLever(mFault)
)
pre
    dom mFault = {<AIN_OPEN>, <AIN_SHORT>, <AIN_INTERNAL>};
```

```
public update : () ==> ()
update() ==
(
    let ainValue = ain.getValue()
    in
    (
        -- Check if the analogue input is valid or not.
        -- If an analogue input fault is confirmed,
        -- then set to invalid.
        -- If the normal N position is detected after
        -- fault recovery, then set to valid.
        if fault(<AIN_OPEN>).isFaultConfirmed() or
           fault(<AIN_SHORT>).isFaultConfirmed() or
           fault(<AIN_INTERNAL>).isFaultConfirmed()
        then
            isValid := false
        else if ainValue = <DIR_N>
        then
            isValid := true
        else
            skip;

        -- Detect the lever position by the analogue input
        -- including fault measures
        if fault(<AIN_OPEN>).getFaultState() = <FLT_NORMAL>
           and
           fault(<AIN_SHORT>).getFaultState() = <FLT_NORMAL>
           and
           fault(<AIN_INTERNAL>).getFaultState() = <FLT_NORMAL>
           and isValid
        then
        (
            if ainValue = <DIR_F> or ainValue = <DIR_R>
            then
                position := ainValue
            else
                position := <DIR_N>
        )
        else
            position := <DIR_N>
    )
);

public getPosition : () ==> [Direction]
getPosition() ==
    return position;

end DirectionLeverAnalogue
```

A.17 The LeverPositionDetection class

```
class LeverPositionDetection is subclass of Common

instance variables
private detectLevPos : [Direction] := nil;
private dinHasPriority : bool := true;
private din : DirectionLever;
private ain : DirectionLever;
private env : Environment;

operations
public LeverPositionDetection : DirectionLever *
                                DirectionLever *
                                Environment
                                ==> LeverPositionDetection
LeverPositionDetection(pDin, pAin, pEnv) ==
(
    din := pDin;
    ain := pAin;
    env := pEnv
);

public update : () ==> ()
update() ==
(
    let dinPosition = din.getPosition(),
        ainPosition = ain.getPosition()
    in
    (
        -- Check which input has priority.
        -- If a digital input fault is confirmed,
        -- then analogue is valid.
        -- After fault recovery, if the lever positions by
        -- digital and analogue are consistent,
        -- then digital is valid.
        if dinPosition = nil
        then
            dinHasPriority := false
        else if dinPosition = ainPosition
        then
            dinHasPriority := true
        else
            skip;

        -- Get the lever position from the prior input
        if dinHasPriority
        then
            detectLevPos := dinPosition
```

```
        else
            detectLevPos := ainPosition
    );

    -- Set the detected lever position to Environment
    env.setDetectLevPosition(detectLevPos)
);

end LeverPositionDetection
```

Requests for Modification of periodic thread definitions and duration and cycles statements

Ken Pierce[1] and Kenneth Lausdahl[2]

[1] School of Computing Science, Newcastle University,
Newcastle upon Tyne, NE1 7RU, United Kingdom
K.G.Pierce@ncl.ac.uk
[2] Aarhus School of Engineering, Dalgas Avenue 2
DK-8000 Aarhus C, Denmark
kel@iha.dk

1 Overview

This note describes two RMs (Requests for Modification to the Language Board) relating to VDM-RT that were submitted by Ken Pierce and Kenneth Lausdahl in March 2011. Both RMs are related. They originated while the authors were building real-time controller models for the DESTECS[3] project. The issue is that two key timing-related constructs in VDM-RT (periodic thread definitions and duration / cycles statements) *only* permit numeric literals to be used to define timing behaviour. This means that "magic numbers" must be hard coded into specifications. We suggest that this is overly restrictive and represents poor coding practice forced by the language, meaning that specifications are harder to read and maintain.

The proposals therefore suggest that these constructs should allow a wider range of expressions to be used instead, in order to increase flexibility and usability. We suggest that at the very least *values* (i.e. constants) should be permitted in addition to numeric literals. This is only a small change to the syntax and has no effect on the semantics. As a further step, these constructs could allow references to instance variables, which would allow more object-level flexibility. For example, this would allow instances of the same class to have different periodic thread behaviour. This would introduce semantic questions however, such as when references are read and if they can be modified during execution.

Clearly, depending on the choices made, the semantics may or may not be affected by the proposed changes. The authors therefore suggest that this choice be discussed with the community, however we are keen to see (at least) values and instance variables permitted in periodic thread definitions.

In the remainder of this note, Section 2 attempts to motivate the need for changes with a view from the DESTECS project, followed by more details of the RMs in Section 3 and 4.

[3] http://www.destecs.org/

2 Motivation for requests

Our main reason for requesting these changes comes from our attempts (within the DESTECS project) to build real-time controller models for use in co-simulation with continuous-time plant (or environment) models.

Parts of the controllers that we wish to model in DESTECS —such as low-level PID controllers— need to know the sample time (i.e. the thread period) in order to calculate the control output. In a model where other time-related calculations are required (such as calculation of discrete integrals), this information needs to be available in multiple places. Typically, one would follow good programming practice and define a constant (**`value`**) in order to ensure that the value used is correct in all places.

The necessity to hard code the thread period as a numeric literal permits the possibility that the actual thread period and the value that the controller uses to calculate control actions can differ. This typically results in (often wildly) incorrect simulation results. This situation can arise if the modeller changes one of the values and forgets to change the other. This is particularly easy if the model is being altered by someone new who didn't originally write it. Therefore we believe that allowing constants to be used in periodic thread definitions is entirely justified.

In the DESTECS project, we are also interested in design space exploration (DSE) using co-simulation. This is where a set of candidate designs are evaluated by comparing the results of co-simulation. The best design is chosen according to some parameters, for example, the design that meets the requirements at the cheapest projected cost of components.

To increase the ability to evaluate designs, we wish to introduce automation where possible. Essentially, we would like the ability to alter certain parameters of controller models, run simulations to gather results, then compare them. This is somewhat similar to the combinatorial testing offered by the Overture tool already.

Parameters should include the number of controllers, their loop period and their deployment architecture. In one case study (the ChessWay personal transporter [FLP+10]) for example, a small safety monitor runs on it's own CPU at a speed much higher than the main controller, in order to ensure the safety of the rider. We might wish to run multiple simulations with differing main controller and safety controller speeds, in order to find the most effective combination.

The hard-coded nature of periodic threads and duration/cycle statements makes this difficult, in addition to the class-level nature of periodic threads. Currently, the user must edit the model in between each simulation run (ensuring that they update the relevant values in all places in which they appear). This is far from convenient and can lead to errors. Permitting instance variables to be referenced by periodic thread definitions would allow threading behaviour to be specified through object constructors. This fits with the object-oriented nature of VDM-RT and more easily permits automation.

Object-level periodic threads would also help in another aim of the DESTECS project, which is to provide libraries of common components for building real-time controllers in VDM (to help users unfamiliar with VDM to begin building working examples quickly). We would like to do this by providing classes that can be instantiated as objects by users. It would be preferable to allow periodic behaviour to be config-

urable via a constructor, rather than requiring new users to begin using concepts such as inheritance straight away.

3 Expressions in periodic thread definitions (ID: 3220182)

The DESTECS project [BLV+10] focuses on co-simulation of discrete-event controllers written in VDM-RT with continuous-time models described in 20-sim [Bro97,Kle06]. The controller models that we wish to produce are typically real-time controllers that must perform an action at a regular interval. For example, reading sensors and producing control actions at a frequency of 1000Hz (or a period of 1 millisecond).

The best way to achieve this in VDM-RT is with a periodic thread definition:

```
thread
periodic(period,delay,jitter,offset)
```

The values of *period*, *delay*, *jitter* and *offset* can only be hard coded as numeric literals. Therefore to run a controller thread at 1000Hz (one thousand times per second, or 1 millisecond per cycle), the following definition could be used:

```
thread
periodic(1,0,0,0)
```

3.1 Extension 1: Using `values` (and simple calculations)

The following definition currently couldn't be used, however it is perhaps a more intuitive way to define the behaviour and better coding practice. Here a constant called `FREQUENCY` is used to set the frequency (in Hz), with the conversion to period (in milliseconds) handled in the periodic definition:

```
values
  FREQUENCY: nat1 = 1000

thread
  periodic(1000/FREQUENCY, 0, 0, 0)
```

3.2 Extension 2: Using `instance variables`

Note that threads also are class-level (as opposed to object-level). This means that each object instance of a periodic class must have the same periodic behaviour. In order to model two copies of a controller running at the same time but at *different* speeds (or perhaps more likely different jitter or delay), it is necessary to define a controller class without a thread and then create two subclasses that only describe the periodic behaviour, for example:

```
-- this object will run normally
class MyControllerA is subclass of MyController

thread
  periodic(1, 0, 0, 0)

end MyControllerA

-- this object is more jittery
class MyControllerB is subclass of MyController

thread
  periodic(1, 10, 0, 0)

end MyControllerB
```

By permitting instance variable expressions to appear in periodic definitions, periodic behaviour can become object-level and permit instances of the same class do have different thread behaviour. Consider this example:

```
class MyController

instance variables
  private frequency: nat1 := 1000

thread
  periodic(1000/frequency, 0, 0, 0)

end MyController
```

Here, the value of frequency can be set in the constructor. This makes practices such as automated testing possible, including automated exploration of alternative designs and deployments which is one of the aims of the DESTECS approach. This modification would also permit objects instantiated from libraries of classes to have their periodic behaviour configured through a constructor:

```
class LibController

instance variables
  -- default frequency
  private frequency: nat1 := 1000

operations
  public LibController: nat1 ==> LibController
```

```
  LibController(freq) ==
    -- user-configurable frequency
    frequency := freq

thread
  periodic(1000/frequency, 0, 0, 0)

end LibController
```

A clear issue is when the value of frequency is evaluated and whether or not it can be changed. In order to ensure that a periodic thread's behaviour is unchanging over the course of an execution (i.e. by not allowing frequency to be assigned during run-time), we suggest that something like a **final** keyword (or equivalent) is introduced. In the Java language [GJSB05], a final variable can be assigned to *at most once* during construction of an object and must be assigned to during object construction. If this concept were adopted into VDM-RT, then periodic definitions could be restricted only to permit instance variables that are declared final:

```
instance variables
  private final frequency;
```

4 Values in duration / cycles statements (ID: 3220223)

The issue with duration and cycles is very similar to that of periodic threads described above. These two statements delay the internal clock of VDM-RT to simulate actions taking time. This delay can be based on the speed of the (simulated) CPU, using **cycles**, or based on (simulated) time, using **duration**. The following assignment statements would therefore take 10 (simulated) clock cycles to complete:

```
cycles(10) ( x := 1; y := 2; z := true )
```

Or the statement could be modified to take 2 (simulated) milliseconds, regardless of the (simulated) CPU speed:

```
duration(2) ( x := 1; y := 2; z := true )
```

As with periodic threads however, only numeric literals are permitted for describing the number of cycles or duration. Therefore the following is not valid:

```
values
  CYCLES: nat = 7
```

```
operations
  public op1: () ==> ()
  op1() ==
    cycles(CYCLES) (
      x := 1; y := 2; z := true
    )
```

So again magic numbers must be hard coded into specifications and durations / cycles statements cannot be altered through object-construction (they are static by class). We therefore request that expressions of time in duration and cycles statements not be restricted to numeric literals. Again there is a question of how flexible we want the language to be (values, instance variables, or functions, etc.) and when and how often the expressions should be evaluated (i.e. once during object construction or every time the statement is executed). We therefore suggest that the community should discuss these issues together and modify the RMs accordingly.

Note there are also related RMs to the two described here, namely 3220437: Extend duration and cycles (allow intervals + probabilities) and 3220324: Sporadic thread definitions.

Acknowledgements

The authors' work is supported by the EU FP7 project DESTECS.

References

[BLV+10] J. F. Broenink, P. G. Larsen, M. Verhoef, C. Kleijn, D. Jovanovic, K. Pierce, and Wouters F. Design support and tooling for dependable embedded control software. In *Proceedings of Serene 2010 International Workshop on Software Engineering for Resilient Systems*. ACM, April 2010.

[Bro97] Jan F. Broenink. Modelling, Simulation and Analysis with 20-Sim. *Journal A Special Issue CACSD*, 38(3):22–25, 1997.

[FLP+10] John Fitzgerald, Peter Gorm Larsen, Ken Pierce, Marcel Verhoef, and Sune Wolff. Collaborative Modelling and Co-simulation in the Development of Dependable Embedded Systems. In D. Méry and S. Merz, editors, *IFM 2010, Integrated Formal Methods*, volume 6396 of *Lecture Notes in Computer Science*, pages 12–26. Springer-Verlag, October 2010.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, Amsterdam, 3 edition, June 2005.

[Kle06] Christian Kleijn. Modelling and Simulation of Fluid Power Systems with 20-sim. *International Journal of Fluid Power*, 7(3), November 2006.

Sune Wolff & John Fitzgerald: Proceedings of The 9th Overture Workshop, 2012

Department of Engineering
Aarhus University
Edison, Finlandsgade 22

8200 Aarhus N
Denmark

Tel.: +45 4189 3000