Accepted for publication at the Conference on System Engineering Research – CSER 2007

Interpreting Distributed System Architectures using VDM++ – A Case Study

Marcel Verhoef
Chess, Nieuwe Gracht 78, 2011 NJ Haarlem, The Netherlands and
Radboud University Nijmegen, ICIS, The Netherlands
Marcel.Verhoef@chess.nl
Peter Gorm Larsen
Engineering College of Aarhus, Dalgas Avenue 2, DK-8000, Denmark
pgl@iha.dk

Abstract

The complexity of real-time embedded systems is increasing in particular due to the use of distributed system architectures for their implementation. Notations to describe these software intensive distributed computer systems at the system-level are at best still in their infancy. An extension to the Vienna Development Method (VDM) is proposed to address the problem of analyzing deployment of software on distributed hardware.

The extension enables the description of systems rather than just software. The language contains primitives for describing concurrent thread-based software components that are explicitly deployed on one or more processors which in turn are interconnected by one or more networks.

The value of this modelling approach is illustrated using a case study for a missile counter measures system. We describe different alternative distributed architectures and we explore the projected timing properties for given scenarios of missile attacks.

Introduction

The early stages of the development of a new system are typically extremely volatile because there are still many unknowns in the requirements and there are many proposed implementation strategies. Key decisions need to be taken by the system architect, while often working under extreme time pressure. Dealing with these different dimensions of uncertainty and at the same time reducing project and product risks while increasing the confidence in the design, makes system architecting a very challenging task. The system architect needs to *bridge the gap* between the disciplines and deal with the *design complexity* in a very *cost-effective* way.

The approach advocated in this paper aims to bridge at least a part of this gap, such that potential bottlenecks of a given system architecture can be discovered *very early* in the system design in a cost-effective fashion. Such bottlenecks might be networks with too small a capacity for the communication load in a given scenario or operations executed on processors that would be too slow compared to the given timing requirements. The overall idea is to create an abstract model of the system under development using an unambiguous formal notation that can be used for both describing the intended behaviour of the system as well as the deployment of software onto a hardware architecture. Tool support then enables simulation of abstract models of different scenarios and allows visualisation of the execution traces and examination of system level timing properties.

The paper first explains the VDM++ technology used for expressing the abstract

system models and the capability to exercise these models. This is followed by a case study inspired from the design of an aircraft missile counter measures system. We explore the design space by changing the abstract system model based on "what-if" design scenarios, while constantly evaluating the requirements. The feedback and experiences gained from this exercise allow us to support the architectural design of such a system. Finally a few concluding remarks and future work are provided.

The VDM++ Technology

VDM++ is an object-oriented and model-based specification language with a formally defined syntax, static and dynamic semantics. It is largely a superset of the ISO standardized notation VDM-SL (Andrews et al 1996). Different VDM dialects are supported by industry strength tools, called VDMTools, which are currently owned and further developed by CSK Systems[1]. In VDM++, a model consists of a collection of class specifications. If you are not familiar with VDM++ an overview of the language is provided on the VDM portal[2].

A timed extension to VDM++, simulating a multi-threaded single processor model of computation, was delivered as part of the VICE project: "VDM++ In a Constrained Environment" (Mukherjee et al 2000). Recently, a number of extensions have been proposed to VDM++ in order to make the language more suitable to describe and analyse real-time embedded and distributed systems (Verhoef et al 2006). This paper is an application of those new extensions. It is important to note that this enables a description of the computational burden of functionality using a **`duration`** and a **cycles** keyword.

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

Typically, techniques like VDM are used to formally verify important properties of an abstract model of a system. However, in this work, we have a totally different focus. Here, we use this technology to produce an abstract model at a system architectural level and apply a simulator to investigate the properties of the model, with the aim to evaluate design alternatives rather than prove correctness. For example, we want to play "what-if" scenarios that can give a system architect valuable feedback at the very early stages of development of a new system.

The Counter Measures System

The application used as a case study for this paper is the controller for a Counter Measures (CM) system on a military aircraft. Such a system takes information from several sensors concerning threats and sends commands to actuators which releases flares intended to distract the incoming missile. A potential system architecture can be seen in Figure 1.

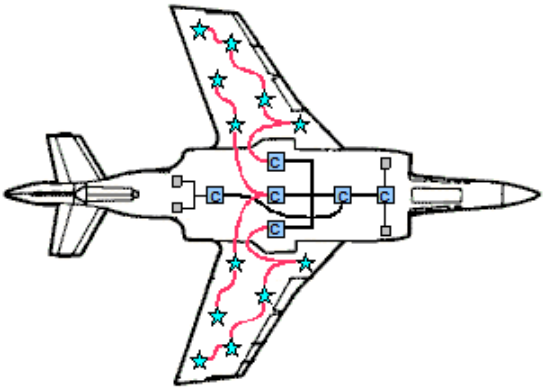

Figure 1: Physical layout CM system

Flares of different types are released in a timed sequence, the number of flares released and the delay between the releases depending on the type of threat and its angle of incidence with the missile. The threat sensors relay the ID of the threat to the controller. For each

different kind of threat and the angle of the missile the controller must then derive a plan of how to deal with the given threat by firing a sequence of flares with a given pattern with a given flare dispenser dealing with the given angle. Such a pattern contains the number of flares to be fired and the delay between each firing. The task communicates the stated number of firings to the flare release hardware with the specified delay between each flare launch. All around the aircraft, different flare dispensers are located dealing with threats arriving from different angles. The following requirements apply to this system:

Priority handling: If a new threat is sensed (in the same angle area where a threat is currently being dealt with), the system should check the priority of the more recent threat. In case the priority of the new threat is greater than the previous one, execution of the current firing sequence should be aborted. Computation of the new firing sequence shall then take place instead.

Parallel handling: If different threats are sensed with angles that are treated by different flare dispensers then the corresponding firing sequences shall be performed in parallel.

Reaction time: The controller shall send the first flare release command within 250 milliseconds of receiving threat information from the sensor.

Abort time: The controller shall be able to abort a firing sequence within 130 milli-seconds.

A guideline to a systematic process explaining the logical steps that can be followed to produce an overall system model for this Counter Measures system is presented in (CSK Systems 2006). The system can be composed out of a number of available hardware components typically delivered as subsystems (sensors and flare dispensers) that can be organised in some physical layout. In this paper, we will focus on one of the proposed layouts (as shown in Figure 1) and study some architectural alternatives.

We assume in a 2-dimentional setting (for simplification reasons):

- four sensor subsystems are available, each covering 90 degrees of viewing angle;
- one missile detector subsystem;
- three flare controller subsystem, each covering 120 degrees of release angle, each controlling four flare dispensers;
- twelve flare dispenser subsystems coping with 30 degrees of release angle each.

In the VDM++ model, this will be the initial configuration considered. We will illustrate how easy it is to change to a different number of components and their allocation to processors.

In practice, requirements changes occur constantly in the development of computer based systems. For example, there are typically alternative suppliers for each subsystem. It is possible to incorporate the assumptions with respect to the functionality and the performance of these subsystems, in the VDM++ model shown in this paper. This means in practice that alternative choices can be explored as well as predicting the consequences of requirement changes.

In the same way redundancy can be built into the system architecture such that the fault tolerance of the system architecture can be evaluated. This approach supports the gradual introduction of redundancy in the design based on dependability and safety analysis of the envisaged system. For example using the Scalable Redundancy Approach (Ahlström et al 2001) for moving from a non-redundant design to a redundant design. Naturally the flexibility of allocating tasks to processors may be limited by physical requirements to the layout of the system and that needs to be taken into account.

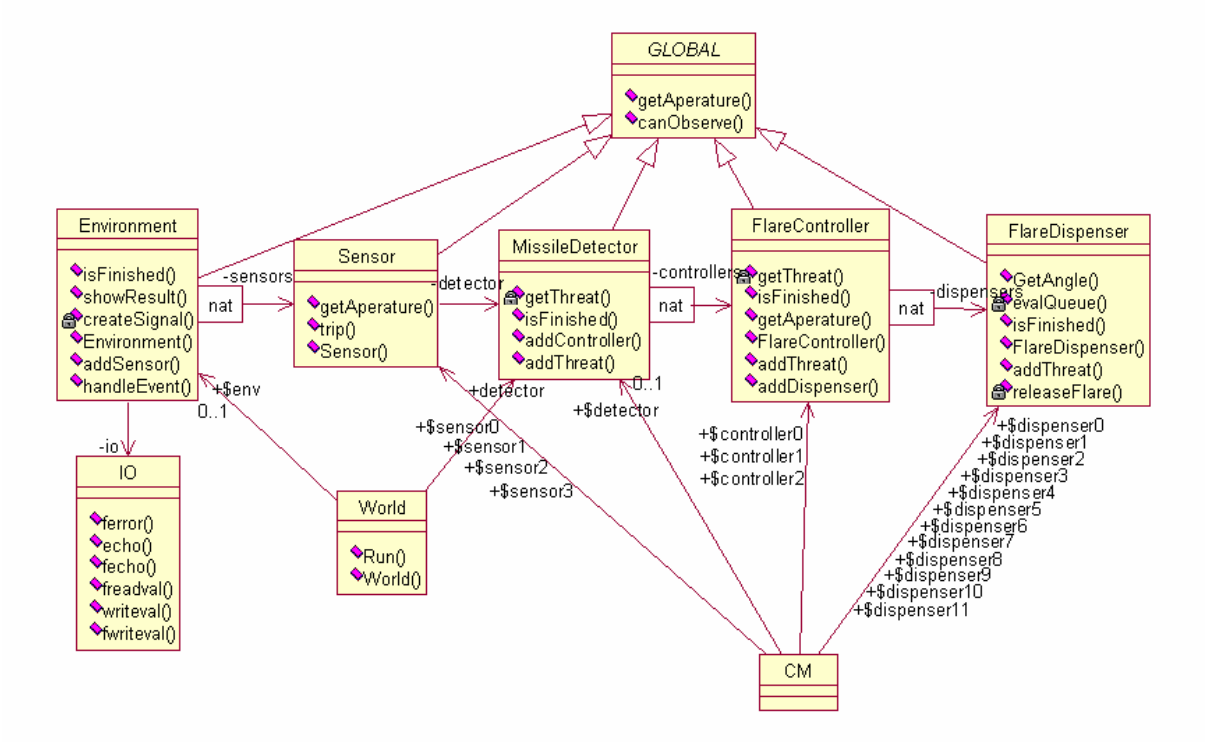

Figure 2: UML class diagram of the Counter Measure case study

The CM System

The overall VDM++ model for the CM case study is presented as a UML class diagram in Figure 2. This figure shows the world, the environment and the application model which all will be explained below in turn. For space limitations, we will limit the description of the model to those parts that introduce the VDM constructs that are used to describe the system architecture.

Initially, an attempt was made to analyse the system performance if all functionality was carried out on a single **`CPU`**. As was to be expected, this simple system architecture did not live up to the timing requirements.

Instead we describe a more suitable distributed architecture here, consisting of six processing units and three networks. The connection between these components can be seen in Figure 3, which reflects the physical lay-out shown in Figure 1. The Sx's and the FD's in Figure 3 are meant to indicate how functionality is deployed to the different processors (see about deployment below).

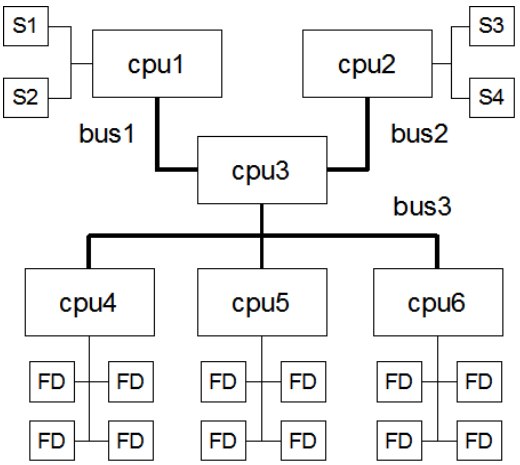

Figure 3: Connection between the CPUs

Objects in the VDM++ model are created inside the so-called **`system`** class, as shown on the next page. Six **`CPU`** and three **`BUS`** instances are declared. Two parameters are used to specify the scheduling policy and the

capacity respectively. In this example, the policies are "First Come First Served" (FCFS) and "Fixed Priority" (FP).

```
system CM

instance variables
  cpu1 :CPU := new CPU(<FCFS>,1E6);
  cpu2 :CPU := new CPU(<FCFS>,1E6);
  cpu3 :CPU := new CPU(<FP>,1E9);
  cpu4 :CPU := new CPU(<FCFS>,1E6);
  cpu5 :CPU := new CPU(<FCFS>,1E6);
  cpu6 :CPU := new CPU(<FCFS>,1E6);
  bus1 :BUS := new BUS(<FCFS>,1E6,
               {cpu1,cpu3});
  bus2 :BUS := new BUS(<FCFS>,1E6,
               {cpu2,cpu3});
  bus3 :BUS := new BUS(<FCFS>,1E6,
               {cpu3,cpu4,cpu5,cpu6});
```

The **BUS** class has similar parameters but they also have a parameter indicating the topology of the network, by listing the set of **CPU**s it is connecting. Each of the system components is declared as follows:

```
instance variables
  public static detector :
    MissileDetector :=
      new MissileDetector();

  public static sensor0 :
    Sensor := new Sensor(0);
   … 3 more sensors are created

  public static controller0 :
    FlareController :=
      new FlareController(0);
   … 2 more flare controllers are created

  public static dispenser0 :
    FlareDispenser :=
      new FlareDispenser(0);
   … 11 more flare dispensers are created
```

Instances of these system components are then deployed to different CPUs as follows:

```
public CM: () ==> CM
CM () ==
 ( cpu1.deploy(sensor0);
   cpu1.deploy(sensor1);
   cpu2.deploy(sensor2);
   cpu2.deploy(sensor3);
   cpu3.deploy(detector);
   cpu3.deploy(controller0);
   … 2 additional deployments to cpu3
   cpu4.deploy(dispenser0);
   … 3 additional deployments to cpu4
   cpu5.deploy(dispenser4);
   … 3 additional deployments to cpu5
   cpu6.deploy(dispenser8);
   … 3 additional deployments to cpu6 )

end CM
```

The allocation of two of the sensors is made to the first two CPUs (as shown in Figure 3), whereas both the missile detectors as well as all three controllers are allocated to processor **cpu3**. Four dispensers are allocated to each of the remaining **CPU**s. For the **CPU**s that use fixed priority scheduling, the priority of the operations executed on that **CPU** is defined in a similar fashion. Note that if we want to experiment with an alternative system architecture or an alternative deployment of task to processors the only changes necessary are here.

The remaining part of the model describes the functionality of the environment and the system components and that is independent of the architectural aspects. This improves the maintainability of the model.

The `World` Class. This top-level class is by convention used to set up the overall model consisting of both the system components and the environment surrounding the system. For example to determine which missile generated by the environment model is recognized by which sensor from the system model etc. In addition, it provides a top-level **`Run`** operation that starts up the system model and the environment model, initiates the scenario, finds out when the scenario is finished and finally shows the results (the output).

The `Environment` Class. This class reads in the input the user wishes to simulate and the times at which input shall appear. It has a periodic thread that creates stimuli to the system using the **`createSignal`** operation as shown in Figure 4.

```
private createSignal: () ==> ()
createSignal () ==
  duration (10)
  (if len inlines > 0
   then (dcl curtime : nat := time, done : bool := false;
         while not done do
           def mk_ (eventid, pmt, pa, pt) = hd inlines in
             if pt <= curtime
             then (for all id in set dom ranges do
                     def mk_(papplhs,pappsize) = ranges(id) in
                       if canObserve(pa,papplhs,pappsize)
                       then sensors(id).trip(eventid,pmt,pa);
                   inlines := tl inlines;
                   done := len inlines = 0;
                   evid := eventid )
             else (done := true;
                   evid := nil))
   else (busy := false;
         evid := nil));
```

Figure 4: The `createSignal` opration from the `Environment` class

The environment model awaits the system reaction using the operation **`handleEvent`**. Timing requirements can easily be stated as a post-condition to the **`handleEvent`** operation by calculating the time delay between each stimulus and response pair.

The application model. The system is composed essentially of four different classes with a small common superclass **`GLOBAL`** that contains a number of common definitions. The classes are **`Sensor`**, **`MissileDetector`**, **`FlareController`** and **`FlareDispenser`** respectively. These classes correspond to the four kinds of system components that are included in the system description above. The sensors simply pass on any information that has been sensed immediately to the missile detector. The missile detector has a periodic thread that inspects whether any new threats have been sensed, and if so relays the relevant information to the flare controller responsible for the sensed area. In turn, the flare controller has a periodic thread that inspects whether any new threats have arrived from the detector and if so relays the information to the right flare dispenser, if the current handling of flares dispensed needs to be affected because of this new arrival. Finally the flare dispensers have a periodic thread that inspects whether any new threats have arrived and if so interrupts the current sequence of flares being dispensed. When flares are released this is signalled to the **`Environment`** using the **`handleEvent`** operation. There, we check whether or not the first flare was released on time, for example.

Overall. The size of the full model of the CM system is less than 600 lines (including blank lines and comment lines) and about 100 of these lines are dedicated to the **`CM`** class of which extracts have been shown above.

Evaluating the CM Architecture

Since the model of the CM system as a whole can be executed by the interpreter from VDMTools, it is possible to experiment with different scenarios (or system-level test cases) at this point, to evaluate the architecture the model represents. For the given model this is done by creating a file with the time stamped events that constitute stimuli that we would like to simulate from the environment, and then interpreting the model by executing the

top-level `Run` operation in the `World` class. An input file could for example look like this:

```
[ mk_(1,<MissileA>,45,10000),
  mk_(2,<MissileB>,270,30000)]
```

which would simulate the appearance of two missiles at angle 45 and 270 degrees respectively arriving at 10000 and 30000 time units respectively. Any number of such scenarios can be executed. In case potential bottlenecks are found in such a scenario, the user needs to decide whether it is acceptable or the architecture needs to be adjusted. In the original model, all of the **`CPU`**s were given a capacity of 10E6 cycles per time unit. It turned out that `cpu3` was a bottleneck in reaching the timing requirements and thus a CPU with a higher capacity was chosen and as a consequence the timing requirements could now be met. This is an example of the kind of adjustments that can be made at the system level, without changing the application model. This can be done by adjusting the capacity of the **`CPU`** or the **`BUS`** involved with the given bottleneck. Updates could also easily be made by changing the allocation of the system component instances to different **`CPU`**s. Both of these kinds of changes are made by adjusting parameters of the appropriate hardware component in the `CM` system and then running all the scenarios again. Alternatively, the overall system architecture can be adjusted by changing the number of system components, for example adding flare dispensers. This is also done in the **`CM`** system class, but it may also be possible that it is necessary to make minor adjustments in the **`World`** class. In any case, these adjustments are extremely fast to make and to explore the behaviour of selected scenarios.

Visualizing CM Traces

When a simple scenario, such as the one shown on the left, is executed, the interpreter of VDMTools produces a file where every internal event is logged along with a time stamp and an indication of the place in the model where it appeared. For example, whenever an operation is requested, activated or completed. In the same way, events are logged whenever a message is handled on the bus. Thread events such as creating, swapping in and out and termination of threads.

This detailed logging also enables validation of important timing requirements. After the simulation run, it is possible to visualize such traces and in this way get a much better impression of the load of the different **`CPU`**s and **`BUS`**es with the given scenario.

The log files produced by the VDMTools interpreter can be visualized on top of the Eclipse platform (Eclipse 2006) using the *showtrace* plug-in, which is a part of the Overture tool set (Overture 2006). Once such a log file has been read, it is able to visualize the execution in terms of overviews of the **`CPU`**s and **`BUS`**es as show in Figure 5 and detailed for a specific CPU as shown in Figure 6. Note that it is possible to see both how busy the different components are as well as the details for when each thread is swapped in and out in the given scenario. We imagine that it will be possible some time in the future to visualize violations of timing requirements directly at views such as these.

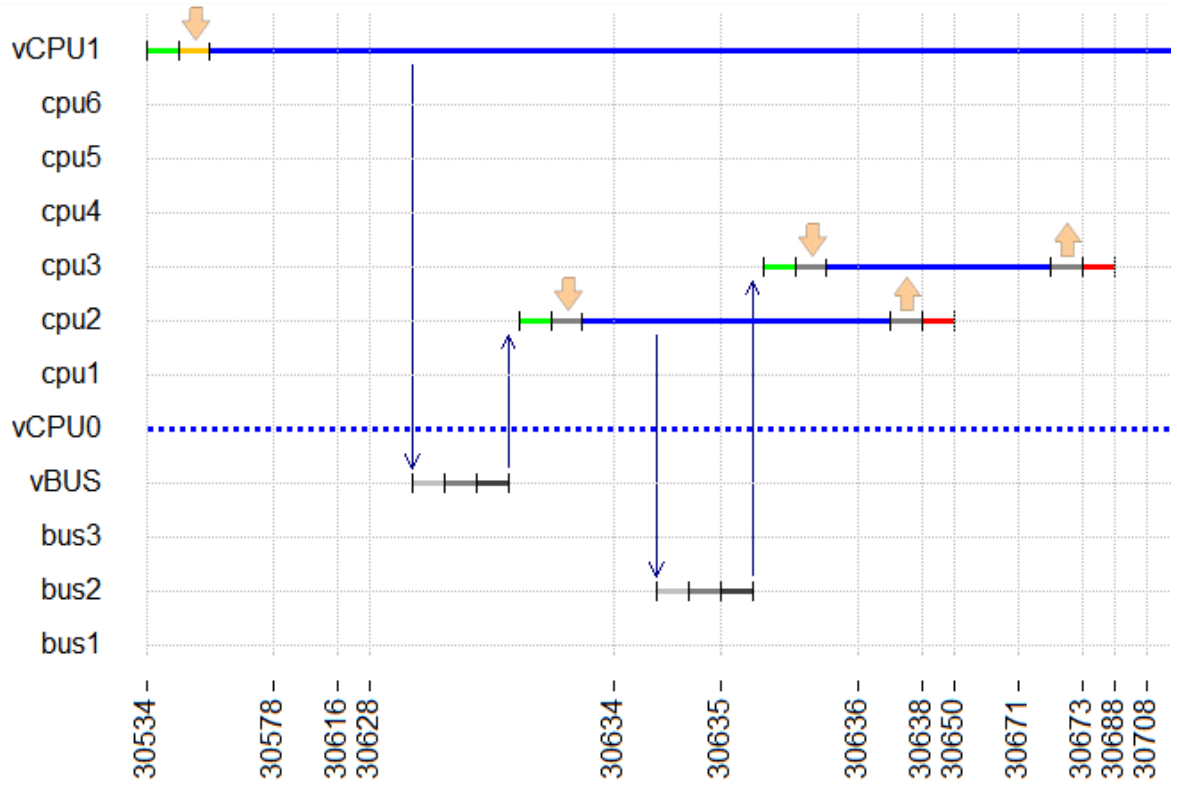

Figure 5: Graphical overview of the system activity at the arrival of the second missile

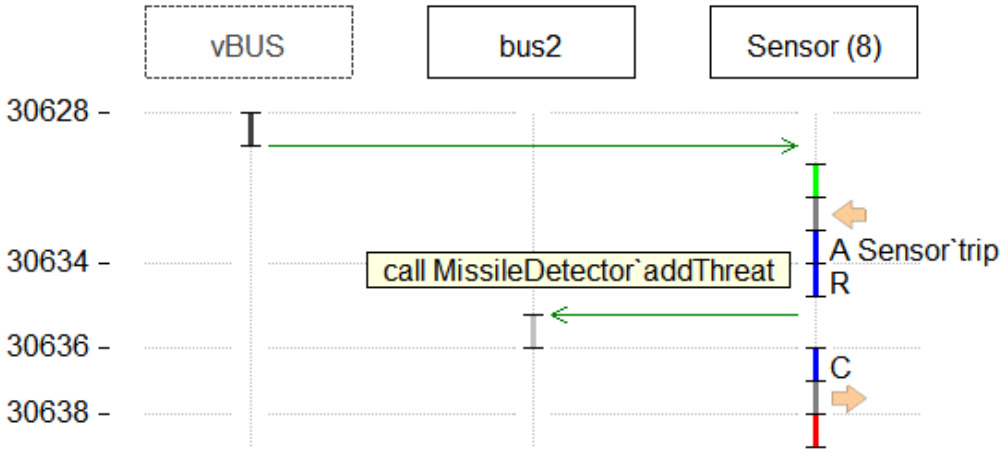

Figure 6: Detailed execution trace of cpu2 where sensor 8 is triggered

Related Work

There already exist a large number of simulators that can be used for modelling and simulation of computer-based systems, for example Matlab/Simulink and the open source counterparts SciLab and Ptolemy. In principle, these environments provide similar functionality to the approach presented in this article, sometimes using additional extensions such as TrueTime and Syndex for architecture exploration.

However, common to all these tools is that they either have *no description* of the actual software functionality or they have a very detailed, *low-level description* of the functionality which is error-prone to write and hard to maintain. For example, Stateflow allows the specification of state transition models, but the action code has to be written as a so-called Matlab s-function in order to be executable. Alternatively, C-code can be compiled directly into the model. In both cases, the level of abstraction is low, certainly not amenable for the type of architectural exploration we envisage here.

Instead, we propose to use VDM++ to model the software of the system. VDMTools provides a round-trip engineering facility to UML, which allows a close correspondence to the tools and languages that software engineers use. In fact, there is no need to make a separate model, the VDM++ model can be used later on as the top-level specification of the software to be implemented. Last but not least, due to its formal basis, the same VDM++ models can also be subjected to formal analysis tools to infer correctness; this provides a gradual transition path for our exploratory approach to the design and implementation phase whereas all other approaches require a paradigm shift towards the target language and platform. Only in a limited number of cases can automation (by means of code generation) help to overcome this hurdle.

In summary, this paper proposes a novel simulation based evaluation approach where models of different system architectures can be investigated efficiently because a high-level abstract description of the software functionality can be maintained at all times.

Concluding Remarks

The case study has illustrated how the VDM++ approach enables the production of a model of software components that can be deployed to different processors that communicate over different communication media with different capacities. It has been demonstrated how that model can enable the formulation of system-level timing requirements and how potential bottlenecks can be discovered at a very early stage in development. In addition it has been illustrated how graphical overviews of traces of the interpretation of a given scenario executed on the model with the different processors can be displayed.

However, this is by no means a solution that solves all the problems of a system architect in the early stages of the system life cycle. However, we do believe that this approach has a lot of potential and extension possibilities that can be used by system architects in a very cost-efficient way.

There are many different directions that we envisage this approach can be extended. First of all, we have already started investigating how this technology can be combined with continuous time models of physical processes in the environment, made in different formalisms. We think that such an extension is particular valuable in order to be able to bridge the gap between traditionally disjoint disciplines such as mechanical, control and software engineering. In addition we believe that the support provided for the validation of system-level timing properties over the traces from the simulations can be significantly improved by specifying explicit properties over those traces.

We see the approach proposed here as a means to reduce complexity while the rigor is increased at the same time. Since we target this to be used in the very early life cycle stages we envisage that the ability of being able to execute different scenarios can provide important feedback to the system architect both about the intended system behaviour as well as potential timing bottlenecks for the system. Discovery of those limitations can be important in the dimensioning and structuring of the system architecture.

Acknowledgements

We would like to thank Michael Alrøe, Troels Fedder Jensen, Val Jones, Brian Larson, Nico Plat, Shin Sahara and Peter van den Bosch for their valuable comments and support. This work has been carried out as part of the Boderc project under responsibility of the Embedded Systems Institute. This project was partially supported by the Netherlands Ministry of Economic Affairs.

References

Ahlström, K., Torin J. and Johannessen P., Design Method for Conceptual Design of "By-Wire" Control: Two Case Studies, *Proceedings of the Seventh International Conference on Engineering of Complex Computer Systems*, 2001.

Andrews, Derek et al, *Vienna Development Method - Specification Language part 1, Base Language*, ISO/IEC 13817-1, 1996.

CSK Systems, *Development Guidelines for Real-Time Systems Using VDMTools*, Technical Report [1] , 2006

Eclipse, *Eclipse - An Open Development Platform*, http://www.eclipse.org, 2006.

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

Mukherjee, P, Bousquet, F., Delabre, J., Painter, S. and Larsen P.G., Exploring Timing Properties Using VDM++ on an Industrial Application, *Proceedings of the Second VDM Workshop*, 2000.

Overture - *Open-source Tools for Formal Modelling*, http://www.overturetool.org 2006.

Verhoef, M., Larsen, P.G., and Hooman J., Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. *Proceedings of FM 2006: Formal Methods*, August, 2006. Springer, LNCS 4085, pp 147-162.

Biography

Marcel Verhoef studied computer science at Delft University of Technology in the Netherlands (MSc, 1993), in the area of computer languages and compilers. He has worked in industry for most of his career. He works for Chess since 1998. He has worked as a systems architect for clients such as the European Space Agency, the Dutch department of Defense, Siemens VDO Automotive, Oce Technologies and Philips. In several of these projects, he applied formal methods. He represents Chess in the BODERC project at the Embedded Systems Institute. He is due to defend his PhD in 2007.

Peter Gorm Larsen studied computer science at the Technical University of Denmark (MSc 1988, PhD 1995) with focus on semantics, computer languages and tool support. He has worked in industry for most of his career. For 13 years he worked with IFAD and was the main architect of VDMTools and he was responsible for support of VDMTools world-wide. For 3,5 years he worked for Systematic Software Engineering mainly doing business development for large defense projects. He is now a full professor at the Engineering College of Aarhus and in addition has his own one-man consultancy company PGL Consult.