COMPUTING
SCIENCE

Proceedings of the 10th Overture Workshop

Nico Plat, Claus Ballegaard Nielsen and Steve Riddle (Eds.)
Abstract

This technical report represents the proceedings of the 10th Overture Workshop, held at CNAM, Paris, France on 28 August 2012, as part of the FM 2012 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 10th Overture workshop emphasised topics that mesh closely with the themes of FM 2012 itself, particularly application experience in industry, validation of tools and methods and the development of tools.
Bibliographical details

PLAT, N., BALLEGAARD NIELSEN, C., RIDDLE, S. (EDS.)

Proceedings of the 10th Overture Workshop
[By] N. Plat, C. Ballegaard Nielsen, S. Riddle (Eds.)


(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1345)

Added entries

NEWCASTLE UNIVERSITY

Abstract

This technical report represents the proceedings of the 10th Overture Workshop, held at CNAM, Paris, France on 28 August 2012, as part of the FM 2012 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 10th Overture workshop emphasised topics that mesh closely with the themes of FM 2012 itself, particularly application experience in industry, validation of tools and methods and the development of tools.

About the editors

Nico Plat is technical director at West Consulting B.V., The Netherlands (www.west.nl), and also acts as an enterprise and software architect for various projects by the company. He currently is Chairman of the Overture Language Board.

Claus Ballegård Nielsen is a PhD fellow in the Electrical and Computer Engineering group under the Department of Engineering at Aarhus University. In his PhD he is researching the use of formal methods and tools in the area of System of Systems. He is closely associated with both the Overture project and the COMPASS research project in which parts of the Overture community are involved.

Steve Riddle received his PhD from the University of Bath in 1997; his thesis concerned the use of partial specifications and refinement theory to aid the process of explaining complex systems. His research interests include requirements engineering and traceability, formal specification and evidence-based argumentation. He is currently a member of the EU FP7 Project COMPASS.

Suggested keywords

VDM
FORMAL METHODS
TOOLS, MODELLING
VERIFICATION
ANALYSIS
SIMULATION
Proceedings
of the
10th Overture Workshop

Paris, 2012

Edited by
Nico Plat
Claus Ballegård Nielsen
Steve Riddle

August 28, 2012
Introduction

Overture (www.overturetool.org) is 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 technical report represents the proceedings of the 10th Overture Workshop, held at CNAM, Paris, France on 28 August 2012, as part of the FM 2012 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 most recent (9th) workshop, co-located with FM 2011, included several reports of development in control and embedded systems design. The 10th Overture workshop emphasised topics that mesh closely with the themes of FM 2012 itself, particularly application experience in industry, validation of tools and methods and the development of tools.

For the 10th workshop, during the morning session we were delighted to welcome contributions with the following titles:

- The Co-Simulation of a Cardiac Pacemaker using VDM and 20-sim by Carl Gamble, Martin Mansfield, and John Fitzgerald.
- Supporting the Partitioning Process in Hardware/Software Co-design with VDM-RT by José Antonio Esparza Isasa, Peter Gorm Larsen and Kim Bjerge.
- Evolution of the Overture Tool Platform by Joey Coleman, Anders Kaels Malmos, Claus Ballegård Nielsen and Peter Gorm Larsen.
- Towards an extensible core model for Digital Rights Management in VDM by Rasmus W. Lauritsen and Lasse Lorenzen.
- Towards a Co-simulation Semantics of VDM-RT/Overture and 20-sim by Joey Coleman, Kenneth Lausdahl and Peter Gorm Larsen,
The afternoon part of the workshop was dedicated to getting practical experience with the Overture platform. The intention of this was to get more developers able to (and interested in) actively contributing to the further development of the Overture toolset. A few presentations were given, after which the workshop participants had the opportunity to get hands-on experience working with development and compilation of different aspects chosen individually. In this process the core developers helped the participants in order to enable more people to contribute to the Overture open source initiative after the workshop.

The organizers of the workshop were:

- Nico Plat (West Consulting BV, The Netherlands)
- Claus Ballegård Nielsen (Aarhus University, Denmark)
- Steve Riddle (Newcastle University, UK)

Members of the program committee were:

- Nick Battle (Fujitsu, UK)
- John Fitzgerald (Newcastle University, UK)
- Sakoh Hiroshi (Designers Den Corporation, Japan)
- Cliff Jones (Newcastle University, UK)
- Peter Gorm Larsen (Aarhus University, Denmark)
- Nico Plat (West Consulting BV, The Netherlands)
- Steve Riddle (University of Newcastle, UK)
- Shin Sahara (SCSK Corporation, Japan)
- Marcel Verhoef (Chess IT, The Netherlands)
- Sune Wolff (Terma A/S, Denmark)

Acknowledgments

The workshop is supported by the Centre for Software Reliability (Newcastle University) and the EU FP7 project COMPASS\textsuperscript{1}.

\textsuperscript{1}http://www.compass-research.eu/
## Regular papers

- Supporting the Partitioning Process in Hardware/Software Co-design with VDM-RT  
  *José Antonio Esparza Isasa, Peter Gorm Larsen and Kim Bjerge*  
  5

- Evolution of the Overture Tool Platform  
  *Joey W. Coleman, Anders Kaels Malmos, Claus Ballegård Nielsen and Peter Gorm Larsen*  
  13

- Towards an extensible core model for Digital Rights Management in VDM  
  *Rasmus W. Lauritsen and Lasse Lorentzen*  
  20

- Towards a Co-simulation Semantics of VDM-RT/Overture and 20-sim  
  *Kenneth Lausdahl, Joey W. Coleman and Peter Gorm Larsen*  
  30

  *Shigeru Kusakabe, Yoichi Omori and Keijiro Araki*  
  38
Supporting the Partitioning Process in Hardware/Software Co-design with VDM-RT

José Antonio Esparza Isasa, Peter Gorm Larsen and Kim Bjerge
Department of Engineering, Aarhus University
Finlandsvej 24, DK-8200 Aarhus N, Denmark
jiae@iha.dk, pgl@iha.dk, kbe@iha.dk

Abstract—For modern embedded systems it is not evident what functionality to place in software versus in hardware. New platforms enable Hardware/Software Co-design such that the decisions can be delayed as long as possible. This paper proposes a new methodology using VDM-RT in support of the partitioning process. This approach allows system engineers to perform design space exploration at an abstract level earlier in the development process. Based on the information gained during this analysis, system engineers are able to allocate the required system functionality in hardware and software blocks. This partitioning process can be done through a rigorous study of the design trade-offs by applying the VDM-RT methodology.

I. INTRODUCTION

Modelling system-wide functionality has been done in a Hardware/Software Co-design setting for a number of years. The advantage of this kind of approach is that it allows decisions about how to optimally partition the required functionality between the hardware side and the software side. There are different refinement-oriented approaches supporting this process in a systematic step-wise fashion. However, most of the existing work in this area makes use of very low-level models of the software parts. The work presented in this paper aims to improve that situation by adjusting the VDM-RT development process [8] for embedded systems to better support this partitioning process.

In Section II we explain more about partitioning in hardware/software co-design. Afterwards the general VDM-RT development process is described briefly in Section III. This is followed by the main contribution by this paper extending the VDM-RT development process with support for hardware/software co-design in Section IV. Then Section V illustrates how this new process can be used in both a very small case as well as for a significant case study. Finally Section VI and Section VII give a short overview of related work and concluding remarks.

II. PARTITIONING IN HARDWARE/SOFTWARE CO-DESIGN

Efficient embedded systems require hybrid heterogeneous solutions that are able to integrate hardware and software components. A challenge that results from the combination of hardware and software components is the creation of an optimal architecture that implements the system functionality. This is also known as the partitioning problem. We present an approach that starts with a initial model representing a partitioning solution in software and then moves functionality to hardware until timing constraints are met. The optimal design solution is influenced by multiple factors like price, silicon area, power and performance. In order to cope with a variety of implementation details in the design process, multiple authors agree on the possibility of using a heterogeneous modelling approach [4], [15], [13]. The abstractions that are introduced in those models are as important as the modelling itself. Abstraction allows engineers to leave out selected details focusing on the essential problem and enables a step-wise refinement approach where detail is added as the model evolves. The purpose is to achieve a better level of problem understanding and a higher level of confidence in the design solution. Abstraction applied to modelling allows an efficient system simulation by leaving implementation details out. It enables the engineer to explore and compare different design alternatives for different partitions of hardware and software components.

Current methodologies in embedded systems designs are derived from the System-Level-Design (SLD) approach. SLD pursues the paradigm of specify/explore/refine [4]. It starts the design process by modelling at the highest level of abstraction and then performs refinement of the model down to a final implementation. Hardware/Software Co-design is a promising SLD approach with a top-down methodology [16]. This approach is a model-driven engineering methodology that aims to derive a system architecture from behavioural models. Deriving the system architecture implies performing the partitioning process that results in the definition of the hardware and software blocks in a well-founded manner. The application of modelling techniques play a major role in the development of new embedded systems. Models and abstractions are key tools to master the complexity present in today’s Hardware/Software systems [13], [4]. Approaches like Hardware/Software Co-design are gaining importance since they are focused on developing the functionality to be delivered, postponing the partitioning and functionality allocation process to a later stage in the process.

Every system specification presents a collection of desirable targets that must be fulfilled in order to produce a successful implementation. Some targets are specified in the form of requirements that are mandatory to satisfy. Other targets, are design variables that would be nice to achieve. The design criteria for the different targets are used in trade-off studies to evaluate and comparing different design solutions.
The challenge here is to compromise in order to find the optimum balance in the trade-offs between these criteria. From a partitioning point of view, the most important criteria are: performance, flexibility, energy efficiency, complexity and used silicon area [1], [12].

III. THE VDM-RT DEVELOPMENT PROCESS

Larsen et al. [8] proposed a development process for the development of distributed real-time systems based on the VDM languages. This development process has four modelling stages: the sequential modelling stage, the concurrent modelling stage, the real-time modelling stage and the distributed real-time modelling stage. Each modelling stage produces a new model that is focused on capturing specific system aspects. The most relevant modelling stages in the work presented here are the real-time and distributed real-time modelling stages.

These stages make use of the real-time version of the VDM language: VDM-RT. This version incorporates a notion of time that is provided by the VDM interpreter. The models produced using this notion of time are more accurate and closer to reality than the previous model. The main advantage of a real-time model is the possibility of conducting complex analysis of the model execution over time. For example, in a real-time model it is possible to analyze whether the real-time deadlines present in the system are met or not.

This distributed-real time modelling stage enables the exploration of different deployment scenarios in a real-time setting. In addition to the real-time analysis introduced in the VDM real-time model, it is possible to study the system performance in a distributed setting, in which several networked processing nodes are executing different parts of the model. In order to conduct such an analysis, additional constructs are introduced. The most relevant ones are the classes CPU and BUS. A CPU represents a concrete execution environment running a Real-Time Operating System (RTOS). It is possible to specify a certain processing frequency as well as a scheduling policy. The BUS class represents a communication channel between CPUs. Each bus represents a point-to-point connection and incorporates a certain transmission policy. Once a certain architecture has been defined by the means of CPUs and BUSes, the modeller is able to deploy different parts of the model in the different CPUs and execute the model. The ultimate goal of this phase is to find a particular architecture that satisfies the system real-time requirements, however no specific guidelines in the VDM-RT context have been presented before on how this design-space exploration shall be conducted and on how certain hardware/software architectures shall be modelled.

IV. ADOPTING THE VDM-RT PROCESS TO SUPPORT HARDWARE/SOFTWARE CO-DESIGN

We propose an extension of the VDM-RT development process in order to represent and analyze different hardware/software architectures [3]. This extension introduces modelling guidelines in the creation of the distributed real-time model in order to support partitioning decisions. This phase takes the VDM-RT real-time model as a starting point to study different hardware/software architectures. The relation between the newly suggested exploration phase and the modelling stages from Section III can be seen in Figure 1. The ultimate goal of this study is to find a suitable architectural candidate that fulfills the system requirements. Note that finding a suitable architecture does not imply finding the optimal one from a system perspective. In order to analyze different architectural candidates implementing functionality in different hardware and software units, one needs to be able to model these different architectures first. The VDM-RT development process proposed in [8] does not include specific guidelines or modelling advice to represent hardware partitions. Being able to create a representation is a precondition for design space exploration in a hardware/software co-design scenario. This work presents a number of guidelines and modelling advice so that hardware partitions can be incorporated in models created by following the VDM-RT development process mentioned above.

There are two relevant problems for exploration of hardware/software architectures. The first problem, also referred as hardware offloading, is the introduction of a partition to deploy a particular functionality in a single hardware block. The second one involves more complex architectural exploration and consists of the incorporation of multiple hardware partitions in the model. In order to tackle both problems from a VDM-RT model a systematic way to model hardware partitions is needed. Finally, in both single and multi-hardware partition architectures the communication between general-purpose CPUs and hardware partitions needs to be represented. The rest of this section presents how to represent and analyze these two problems and their communication aspects. We will use several small examples to show the application of our methodology in concrete VDM models.

A. Modelling a single hardware partition

In order to incorporate a single hardware partition into a distributed real-time model we assume that offloading the execution of a component to a hardware block will result in a greater performance as compared with its software implementation. Based on this, we propose the application of the following guideline:

**Guideline 1:** A VDM-RT CPU that represents a hardware partition should be configured with a processing speed several orders of magnitude higher than a general-purpose CPU.

The VDM snippet below illustrates the application of guideline 1. In this case the system is composed by a general-purpose CPU (named gpCPU) and a hardware partition (named hwP). Both the general-purpose CPU and the hardware partition are modelled by the use of the general CPU class provided by VDM-RT. As it can be seen, the processing speed of the CPU representing the hardware partition is four orders of magnitude higher if compared with the general-purpose
CPU. This speed increment represents an estimation of the superior performance offered by a hardware partition.

```
gpCPU : CPU := new CPU(<FP>, 1E5);
hwP : CPU := new CPU(<FP>, 1E9);
```

Listing 1: VDM-RT CPUs representing a general purpose CPU and a hardware partition.

B. Modelling several hardware partitions

Modelling several hardware partitions is a more elaborate case that requires the application of the following guideline:

```
socket : BUS := new BUS(<CSMACD>, 72E5, {gpCPU,gpCPU2});
register : BUS := new BUS(<CSMACD>, 72E9, {gpCPU,hwP});
```

Listing 2: VDM-RT buses representing a software and a hardware communication channel.

The reason behind this guideline is that, if the modeller chose to deploy several active components in the same hardware block, he would be modelling a very fast general purpose CPU running an RTOS on top. This situation could provide false performance results not related to the real architecture and possibly lead to erroneous design decisions.

Guideline 3: Each hardware partition should be deployed on an individual CPU configured as a hardware block.

```
gpCPU : CPU := new CPU(<FP>, 1E5);
hwP : CPU := new CPU(<FP>, 1E9);
hwP2 : CPU := new CPU(<FP>, 1E9);
```

Listing 3: VDM-RT CPUs representing a general purpose CPU and two hardware partitions.

In this example, three active components are deployed: channelAccess, packetFiltering and displayer. In this case the modeller is interested on evaluating the performance of a solution in which both channelAccess and packetFiltering components are offloaded to hardware blocks and the displayer component remains in a general-purpose CPU implemented in software. The VDM snippet
Figure 2: Proxy class between the VDM-RT bus and the modelled logic.

Listing 4: Component deployment for partitioning evaluation.

```java
hwP.deploy(channelAccess);
hwP2.deploy(packetFiltering);
gpCPU.deploy(displayer);
```

C. Modelling complex communication between partitions

In guideline 2 we proposed the application of buses configured in a specific manner to represent communication between hardware and software blocks. This approach will be appropriate to represent simple communication schemes. However, there can be cases in which the communication is not that simple in a real system and makes sense to capture this complexity in the model. In order to mitigate this problem, we propose the application of proxy classes at both ends of the bus.

Guideline 4: Use a proxy class in order to incorporate a more complex behaviour in the transmission/reception of data. As an additional advantage, proxy classes separate access to the physical communication medium from the system logic, which leads to a clearer model structure.

Figure 2 shows the application of guideline 4. A class named Proxy is used to model a more complex bus behavior. This class stores temporarily and modifies if necessary the data that is received (Incoming Data) and sent (Outgoing Data) before it is made accessible to the Logic or to the Bus classes. The Proxy class represents the component boundary with the rest of the system and can be used to abstract more detailed communication issues.

V. Case Studies

This section presents two case studies that use the guidelines presented in section IV. The first case study is focused on the control of a servo. This case is introductory and based on the workshop material provided by the Xilinx hardware manufacturer. The second case study considers time synchronization in Audio Video Bridging (AVB) networks. This is a more complex case and it is based on a real industrial problem, proposed by the consumer electronic manufacturer Bang & Olufsen.

A. The Servo Case Study

The servo case study is technically simple and its consideration brings a number of advantages. The servo control problem can be modelled in VDM-RT in a simple manner and the modelling results can be associated to physical results in an intuitive manner. In addition, the servo control architectures discussed here can be implemented faster as compared to more complex case studies, so the modelling results can be related to actual physical results.

1) Servo control and associated real-time constraints: A servo or servomotor is an electromechanical device able to rotate its axis between 0° and 180°. Servos are controlled through Pulse Width Modulation (PWM) signals. These signals have a high and a low level. The high level is also referred as pulse and its duration can be changed in order to modulate it (therefore the name “Pulse Width Modulation”). Different pulse durations will move the servo to different positions, but always within the range of 0° to 180°. Additionally these control pulses need to be provided continuously and at a certain frequency to keep the servo under control. Therefore the control of a servo motor introduces two real-time constraints: A) The pulse duration should be within a certain interval and B) A pulse should be provided every \( x \) milliseconds. In this work we use a servo that requires a pulse duration from 1 to 2 milliseconds every 18 milliseconds.

PWM signals can be generated from software or hardware blocks and it is the embedded system designer’s responsibility to choose for a particular implementation, hence the partitioning problem. These possibilities are explored below by applying the methodology proposed in section IV. In order to evaluate both architectures a common scenario is used, in
which the system has to generate a PWM signal that satisfies the real-time constraints presented above.

2) Architecture 1: Software only solution: The first candidate is a pure software solution in which the logic behind generating PWM signals is implemented in software. In order to model this design alternative the system deployment features a single hardware partition in which the General Purpose Input Output (GPIO)\(^1\) block is deployed. The GPIO is deployed as a hardware block to create a more realistic representation of the real system, in which GPIO units are separated from the CPU. The active classes, responsible for system operation, are deployed in one VDM-RT CPU configured as a general-purpose processor.

Model execution reveals that this architecture would not be suitable in the case where additional software functionality needs to be run in the same CPU. The fulfillment of the real-time deadlines depend on the CPU load. The higher the CPU load the higher the introduced delay on the generation of the control pulse. Since the CPU load represented in this model is increasing over time, the delay on the pulse generation is increasing over time as well. These results can be seen in figure 3. The conclusion that can be drawn is that a software solution that does not involve a hardware partition is feasible only if the CPU load is low and predictable.

![Figure 3: PWM signal generated by the software solution model.](image)

3) Architecture 2: Combined hardware/software solution: This second architectural candidate makes use of a hardware partition running the PWM generation logic. In order to model this design alternative the system uses three VDM-RT CPUs. One CPU is configured as a general-purpose processor and executes the initialization of the PWM generation logic. A second CPU is configured as a hardware partition (application of guideline 1) and executes the PWM generation logic. Finally, a third CPU configured as a hardware partition is used to model the GPIO block. As in the previous architectural candidate, the execution in the general-purpose processor will be delayed by a simulated increment in the CPU load. All the CPUs are connected using buses configured as described by guideline 2.

Model execution reveals that this architecture performs adequately, generating correctly the PWM control signal independently from the CPU load. These results can be seen in figure 4. The conclusion that can be drawn is that off-loading the PWM generation to a hardware block is the most appropriate solution since it allows a continuous fulfillment of the real-time deadlines. On the other hand, such a solution requires implementing the PWM generation in a hardware description language if the selected platform does not incorporate it. Such an implementation will require additional development time and will be more complex.

![Figure 4: PWM signal generated by the combined hardware/software solution model.](image)

4) Results discussion: Both software and combined hardware/software architectures were tested in a real setup. The implementation was provided by workshop material from Xilinx. The signals generated by both architectures were fed into a RC servo and monitored using a digital scope simultaneously. This setup made it possible to see PWM control signal and the physical results at the same time. The information provided by the models was validated against the actual implementation by analyzing the PWM waveforms and by profiling the software execution in the target hardware. The profiling data allowed to relate execution results to the information provided by the overture RT log. No additional model calibration was performed after evaluating the real implementations. The model predictions were correct, meaning that the real-time deadlines were constantly missed by the software solution and correctly fulfilled by the combined hardware/software solution. The consequences of missing the deadlines in the software solution was initially a small jitter in the servo axis that progressively increased resulting in wide oscillations. A prolonged erroneous control based on this PWM signal would have damaged the servo mechanics.

B. The AVB Case Study

This case study is focused on the Audio Video Bridging protocol. AVB uses several IEEE protocols\(^2\) to transmit multimedia content to a set of devices connected to the same network. Synchronization plays a major role in AVB networks and the devices connected need a common notion of time. Synchronization in AVB is provided by the 802.1AS time

---

\(^1\)GPIOs are microcontroller interfaces that can present two states: high or low. Toggling between them can be done from the software running in the microcontroller CPU.

\(^2\)IEEE protocols refer to the Institute of Electrical and Electronics Engineers' standards.
1) The time synchronization protocol: This protocol uses two kinds of messages to keep the network in sync. These messages originate from a master clock and propagate through the network. The first message is the sync event and it signals the point of time at which the time correction originated from the master clock. The second message is the follow-up message and contains the necessary information to compute the time and compensate for network delays in sync propagation to any device in the network.

One kind of device in an AVB network are end-point devices. These devices only consume time corrections and are composed of several blocks, of which the most relevant for this case study are:

- **Media Dependent Layer**: interfaces to the physical transmission medium. It contains the hardware buffers that store the incoming information from the network.
- **Slave port**: receives the time correction information sent by the master clock. It is responsible for local timestamping of the sync events received. It makes use of the media dependent layer.
- **Site sync module**: distributes the information from the ports present in the system to the clock entities and vice-versa. It executes the logic required to associate local timestamps to master clock timestamps and perform the required corrections to compensate for network delays.
- **Clock slave**: keeps a time reference that is subject to corrections. The corrections are managed by the site sync entity.
- **Application**: makes use of the time corrections received by the clock slave.

2 AVB is based on 802.1AS, 802.1Qat and 802.1Qav. The first is described in this paper but the rest are out of the scope of this paper.

2) **Architecture 1: Highly software-based architecture**: This architecture is composed by one hardware partition and one general-purpose CPU. The hardware partition contains the Media Dependant Layer and represents the hardware buffers in which incoming information will be stored. The general-purpose CPU is running the rest of the components presented above. Both CPUs are communicated through a bus configured as described in guideline 2.

Model execution reveals that this architecture introduces a considerable delay on the timestamping process. There is a time difference of 40000 ns between the reception of the sync
event in the hardware buffer and the timestamping by the slave port. As an initial attempt to mitigate this delay, the frequency of the general-purpose CPU was increased by a factor of 10. The error introduced by this modification was reduced by three orders of magnitude.

This analysis shows that the time precision in this architecture varies depending on the CPU speed and load. Additionally, this error negatively affects the maximum synchronization resolution that can be achieved, causing this architecture to be unable to distinguish between two consecutive time corrections.

3) Architecture 2: Mostly software and off-loaded Slave Port: This architecture is composed of three CPUs: one CPU is configured as a hardware partition and the others as general-purpose CPUs. The Media Dependant Layer is deployed together with the slave port component within the same hardware partition. Even though we are deploying two components in the same hardware partition, we are still following guideline 1, since the Media Dependant Layer model has no active behavior at the level of abstraction of this model. The hardware partition is connected through a bus to the first general-purpose CPU that is running the site sync module and the clock slave. Finally, this general-purpose CPU is connected with a second one through a low-speed software bus. The latter CPU is running the application consuming the time corrections.

Model execution revealed that this architecture introduces an outstanding improvement as compared to the highly software-based architecture. The execution speed of the timestamping process is increased orders of magnitude since it is running in hardware. Since the sync event is timestamped in the same hardware partition in which the hardware buffers are implemented, communication through the high-speed bus is avoided. This keeps the introduced error to be within the same order of magnitude. As a consequence of this reduction of error in the timestamp, the resolution that can be achieved by this architecture is four orders of magnitude better than the resolution offered by architecture 1.

The analysis of this architecture helped to conclude that: A) The most critical and time-sensitive process is the sync event timestamping and B) The delay is considerably mitigated if the timestamping process is carried out in the processing unit that is buffering the network traffic. This information can be used in the rest of the exploration stage. Since we have already detected what seemed to be critical part, we can stick to the slave port off-loading and try to explore other improvements in the rest of the system.

4) Architecture 3: Off-loaded Site Sync and Slave Port: This architecture is seeking to improve the performance exhibited by architecture 2 by offloading to hardware the Site sync module. Therefore the only change in here as compared to architecture 2 is that the Site sync module is deployed in the hardware partition together with the slave port.

The execution of the model revealed that the error introduced in the timestamping process is over four times higher as compared to architecture 2. These results seem to be surprising since, generally, hardware implementations perform better than software ones. Note that in this case we have intentionally broken guideline 1, that propose that each component presenting an active behavior must be deployed in a separate hardware partition in order to evaluate its hardware performance. The results provided by the model do not relate correctly to an architecture in which both site sync and slave port are off-loaded to hardware. Instead, the model represents a very fast general purpose CPU with an Real-Time Operating System (RTOS) running two threads. This shows the importance of a correct hardware candidate to single VDM-RT CPU mapping to avoid making wrong architectural design decisions.

5) Architecture 4: Multi-hardware partition: This last architectural model tries to capture the solution described in architecture 3 correctly. The architecture is composed of three CPUs. We have configured the first CPU as a hardware partition and deployed together the media dependant layer and the slave port. We have used a second hardware partition to deploy the site sync module. Finally, we have used a general-purpose CPU and deployed the clock slave and the application blocks. All the CPUs are communicated through high-speed busses configured as described in guideline 2. This architecture reflects the application of guideline 3.

Model execution revealed that this architecture does not perform better than architecture 2, even though there is an additional component off-loaded to hardware. The reason behind the lack of improvements is that site sync is not involved in timestamping (the time critical part of the endpoint device). However, this architecture will be an advantage in case time correction forwarding devices or time correction sources have to be implemented. Both types of devices would benefit from a site sync hardware implementation being able to handle time corrections with a delay within the nanosecond range. Therefore the scalability of this fourth architecture is the highest one as compared to the previous ones. Finally, this architecture is the most expensive in terms of development time and used silicon area as compared to the previous ones.

Figure 6: Comparison of the architectures explored.
conflicting issues have been detected. The models have served to direct the attention of the modeller to aspects that will require further analysis in case a certain hardware/software architecture is selected for implementation, e.g. in case the system engineer decides to implement architecture 2 he must be aware of the particular problems regarding scalability. The figures regarding error performance are not absolute but settle the basis for comparison between the architectures. Finally, the four architectures presented in this case study are compared in figure 6. In this chart each architecture is evaluated in terms of performance, scalability, development time and used silicon area and assigned a figure ranging from 0 (very low) to 100 (very high).  

VI. RELATED WORK  
Mischkalla et al. [10] and Prevostini et al. [11] propose the combination of SystemC with UML profiles in order to create a sustainable modelling-implementation process through the development process. Such an approach is one of many related Hardware/Software Co-design modelling methodologies. It is especially interesting since two modelling technologies are combined providing a semi-formal UML model combined with SystemC. SystemC is a System-Level Design Language based on a C++ class library that supports software and hardware modelling capabilities. The combination of graphical notations applied in the combined SystemC – UML profile-based processes are interesting and the synthesis possibilities outstanding. However SystemC is not a formal modelling language like VDM-RT presented in this work and UML profiles can be limited when it comes to model execution. 

A formal modelling methodology is proposed in [5]. The Software Hardware Engineering methodology is based on the POOSL modelling language. This methodology is based on a formal object-oriented modelling language similar to VDM. It has been applied in a number of industrial projects and several tools have been developed that allow model compilation. In his thesis, Verhoef introduces a real-time extension for the formal modelling language VDM [14]. This extension provides the necessary constructs for the evaluation of different system architectures in a distributed real-time deployment. A complete development process entirely based on the VDM method is proposed in [9], [8]. This methodology makes use of the VDM-Specification Language, the VDM++ modelling language and, finally the VDM-Real Time extension. This process makes use of formal models during the whole project life-cycle. The tool support for this process is provided by Overture [7] and by VDMTools [2]. There is no previous work related to the application of the VDM methodology in the Hardware/Software Co-design field. 

VII. CONCLUDING REMARKS  
This paper has demonstrated how it is possible to adjust the existing VDM-RT process to enable it to support the partitioning process in the Hardware/Software Co-design process. This means that following the supplied guidelines one can make use of such abstract models to get a good feel for the potential bottlenecks in different partitioning candidates. This also means that based on abstract models constructed in this fashion it is possible to judge in a well-founded manner what the best design alternative might look like. This can potentially save time in the design exploration phase where the system architect has many possible solutions. We hope that we will eventually be able to get additional tool support assisting the analysis. Once this is realised we hope that others may successfully benefit from the guidelines supplied in this paper. 

ACKNOWLEDGEMENT  
We would like to thank Joey Coleman and the anonymous reviewers for providing valuable input on drafts of this paper.

REFERENCES  
[15] Waddington, D., Lardieri, P.: Model-Centric Software Hardware Engineering methodology is based on the  
Evolution of the Overture Tool Platform
Joey W. Coleman, Anders Kaels Malmos, Claus Ballegaard Nielsen and Peter Gorm Larsen
Department of Engineering, Aarhus University
Finlandsvej 24, DK-8200 Aarhus N, Denmark
jwc@iha.dk, akm@iha.dk, clausbn@iha.dk, pgl@iha.dk

Abstract—The goal of the Overture project is to develop an open platform to supply tool support for the creation of formal models of systems using the various dialects of VDM. This paper presents a vision of how other new developments can be built on top of the Overture project. In this way it may serve as a more general platform, including languages with similar notations and reuse of functionality in many contexts.

I. INTRODUCTION

When the Overture open source project [7] was originally started back in 2003 the main intent was to:

1) create a new version of the VDM notation that would make it cleaner, selecting the best parts from the old VDM dialects; and to
2) create a new tool making it easy for researchers and practitioners to experiment with extensions both of the language as well as of the tool features.

However, since Nick Battle ended up implementing VDMJ for all three existing dialects of VDM in his VDMJ implementation [3], we ended up incorporating that into Overture and got a lot of additional functionality in one go. Unfortunately, this also meant that, for a long period, we had two different Abstract Syntax Trees (ASTs) that were used for different components inside Overture. A unification of these has been underway for a while now with several attempts to get the right level of automation into the production of a new AST. This is now nearing completion so we will soon have a single AST used for all features inside Overture. Naturally, when we have different dialects of VDM this also means that we have different dialects of the underlying ASTs, but with a considerable overlap. This fact also means that it should be practical to use Overture as a more general platform for other formal languages that are not as close to the existing VDM dialects. This makes the most sense in cases where there is some level of overlap in the underlying language ASTs.

This use as a general platform has already started in two European research projects. When the DESTECS project [1] was defined it was structured as an extension on top of the Overture platform. The same structure was adopted for the COMPASS project [2], subsequently including a new COMPASS Modelling Language (CML). We envisage that this trend will continue in the future where extensions will be performed in different ways.

In this paper we illustrate how such general platform usage is already taking place. We would like to encourage more researchers interested in model-oriented formal languages to make use of the Overture platform in similar ways in the future. We believe that this will result in beneficial situations where reuse of functionality between the existing extensions is possible and, furthermore, the development in new extensions may also be reused by some of the existing VDM dialects [5].

This paper is structured such that this introduction is followed by Section II which provides a short overview of the current family tree making use of the Overture platform. We describe some of the important elements of the Overture Community in Section III. Afterwards Section IV explains technically how extensibility of the Overture platform is provided. Section V illustrates how this extensibility has been used in the DESTECS project. Then Section VI provides the vision of what will happen in this regard in the COMPASS project. This is followed by Section VII where expectations of how some of the plug-ins developed in the COMPASS project can feed back enabling reuse of the new functionality for a subset of the existing VDM dialects. Afterwards Section VIII explains about the future plans for extension functionality on the Overture platform. Finally Section IX concludes the paper and looks ahead for the prospects of how this can evolve in the future.

II. THE OVERTURE FAMILY TREE

Overture was built on top of the Eclipse platform [3] to create an open and extensible tool that could support the development of models in the three VDM dialects. Since then this openness has been utilised to expand Overture’s functionality and scope of use.

Figure 1 illustrates the current state of the platform and shows how it has evolved from the initial Eclipse integration with VDMJ, represented by the VDM Language Family box, and into the collection of tools it contains today. Two European research projects have pushed the Overture platform to expand in two directions:

a) DESTECS: enables co-simulation using multiple modelling paradigms at once. VDM-RT models are integrated with models (of the same system, though different aspects) executed in another tool, and then run simultaneously, synchronized with each other. The openness of the platform has allowed this necessary integration with an external, proprietary tool.

b) COMPASS: a comprehensive model-oriented formal languages and tools platform.

1Design Support and Tooling for Embedded Control Software at http://www.destecs.org
3Eclipse development platform: http://www.eclipse.org/
b) COMPASS: uses a subset of VDM-SL and some VDM++ elements to support CML, and extends some of the Overture/Eclipse user interface as well. This not only moves the platform into the area of System of Systems (SoS) [11], it also proves that Overture is a platform that can be used outside a strict VDM context.

The remainder of this section will present DESTecs and COMPASS in further detail.

A. DESTecs

The DESTecs project has developed tool support for co-simulating continuous-time models and discrete-event models [4]. The goal is to aid the multidisciplinary development of embedded real-time control systems in order to build more dependable systems. In this project, Overture has been extended to create a link between discrete-event controllers defined in VDM-RT and continuous-time models created in the 20-sim tool\(^4\). The continuous-time models use differential equations and the Bond graph notation as their semantic basis, as compared to set theory and logic for the discrete-event models.

In the DESTecs toolset the VDM-RT dialect is generally unchanged and the modification to Overture has been in terms of the high-level underlying semantics and tool internals. As the co-simulation provides integration between existing modelling techniques, the DESTecs focus has been on enabling the exchange of data between simultaneously executing models. This is achieved by marking elements that are shared between models in a co-simulation contract and letting the individual simulation tools interact through this contract.

---

\(^4\) 20-sim: [http://www.20sim.com](http://www.20sim.com)

B. COMPASS and CML

COMPASS is a modelling framework focused on providing a sound formal foundation to support the analysis of SoS architectures and their properties. COMPASS introduces CML, a new purpose-built formalism based on VDM and the CIRCUS formalism [16]. Overture will be used as the tool foundation on which CML models will be developed and simulated. CML itself will allow for the analysis of SoS models through simulation, test automation, static analysis, theorem prover integration and model checking.

As shown in Figure 1, COMPASS will establish a connection from Overture to two external tools: Artisan Studio\(^5\); and RT-tester\(^6\).

a) Artisan Studio: is a modeling toolset that supports the Systems Modelling Language (SysML)\(^7\). In COMPASS a formal link between SoS models expressed in SysML and their expression in CML will be created. This will allow for a greater degree of detail in the overall SoS model. Integration between Overture and Artisan Studio will allow for CML models to be expressed as a graphical architectural model in SysML and vice versa.

b) RT-Tester: is a tool that provides industrial-strength support for automated test generation, test execution, and test evaluation of complex real-time systems. Integration between Overture and RT-Tester will enable dynamic analysis of a subset of CML. The linkage between the two tools is intended to form the initial steps of model-in-the-loop development of running systems.


\(^7\) OMG SysML at [http://www.omgsysml.org/](http://www.omgsysml.org/)
From a VDM perspective, CML has the strongest relation to VDM-SL and small parts of VDM++. The surviving parts are the expression language constructs, parts of the state model, and parts of the model structuring elements. The expressions in CML are a proper subset of VDM expressions, and these are the only parts of VDM that have survived untouched. The top level model structuring elements in CML are classes and processes. While classes are comprised of the same top elements as a VDM++ class, the syntax and semantics of operations are greatly modified. A process defines the active part of a model and can be mapped to a CSP process with state, where the state is defined with VDM-like structures. The concurrency elements of VDM++ are entirely replaced in CML, using a CSP-like model for process execution.

III. THE OVERTURE COMMUNITY

The Overture platform is licensed under the GNU Public License (GPL) [6], and contributions to the platform are welcome under the usual conventions associated with projects that use the GPL. This has been vitally important to the development of the Overture platform: as an example, the intellectual property rights to the VDMJ interpreter are owned by Fujitsu, but the code was generously contributed under the GPL. All code written in the efforts of the various EU projects that have contributed has also been contributed under this license.

The decisions by the various contributors to collectively license their contributions under the GPL has allowed all interested parties to contribute without worrying about unclear legal conditions.

The past development of the Overture platform has come primarily from MSc students at Aarhus University (previously the Engineering College of Aarhus), Marcel Verhoef, and Nick Battle at Fujitsu in the UK. This has resulted in concentration of people with detailed knowledge of the structure and operation of the platform, and these people have been key contributors to the DESTECS project and the (new) COMPASS project. Clearly this has made the adoption of the Overture platform for those projects much easier, but the absence of these people outside these locations will pose a challenge to others wishing to adopt the platform.

A challenge for the Overture community is to improve the structure and clarity of our code and documentation. This is presently an ongoing effort, and some of the work done in the COMPASS project will feed into this. The COMPASS project represents an opportunity as it is the first research project where development on the Overture platform has been planned from the outset to be distributed across many sites.

IV. EXTENSIBILITY OF THE OVERTURE PLATFORM

The Overture platform is comprised of three main sections: one containing the core libraries that implement the VDM language family and associated language tools; one containing all of the IDE-related code and its integration into the Eclipse platform; and, last, a small one containing useful tools to assist with the build process and to interface with other tools. Of these, the first two areas—core and IDE—give the structure of the platform and allow it to be extended. Each section corresponds directly to a folder in the source repository.

The tools section contains the secondary tools that support either the overall build process or the various libraries in the core and IDE sections. The most interesting part of the tools section is the ASTgen (Abstract Syntax Tree generator) library that is used to generate the ASTs for the VDM dialects. It should be noted that the ASTgen tool is properly general: it can be used to generate ASTs for many different languages.

The core section contains the components necessary for manipulating models in the VDM family of languages, including the parser, type checker, proof obligation generator [14], interpreter [9], code libraries for functionality outside the core of VDM [12], combinatorial testing [8], mapping to UML [10], and the unit testing framework for models. All code within this section must be self-contained and have no dependencies on Eclipse or other GUI interfaces; at various points in the past, development code has been rewritten to eliminate such dependencies. It is currently possible to use functionality in the core strictly from a command-line prompt, and a portion of the Overture community does precisely that.

The VDM language parser2 in the core section is a hand-written version that instantiates an AST based on classes created by the ASTgen tool. It is difficult, at present, to effectively modify the parser for VDM language extensions, especially as compared to modifying a parser generated by a tool. Modifications of the parser have, fortunately, been infrequent and have generally involved the original contributor in the process.

The IDE section contains all of the code that is necessary to create the Overture Tool perspective in Eclipse and interface with the functionality in the core. The only IDE platform presently supported is Eclipse; it should be possible to build an IDE in other platforms, but this has not yet been done.

Extensions to the capabilities of the Overture platform at the level of the VDM models are written in the core section, and must be designed to be independent of any specific user interface (including Eclipse). The interface to use these extensions within Eclipse is then added within the IDE section, and must be designed to import the core functionality as a library. This often leads to a specific extension to the Overture Tool platform having its code split across the two sections, but it helps to enforce an appropriate separation of core functionality and interface functionality.

If an extension should require new functionality in the AST of one of the VDM dialects, then the ASTgen script for that dialect is modified and the AST is regenerated using the ASTgen tool. The low frequency of changes to the structure of the dialects’ ASTs means that the generated code is kept in the version control repository and regenerating the AST is only done manually as needed. When it does happen, however, it is rare that any existing code would need to be altered unless the modifications to the ASTgen script change something fundamental to the structure of the AST; something simple like adding a data field will not cause this situation.

2The first version of this was developed by Marcel Verhoef while he conducted his PhD work.

2Contributed by Nick Battle.
So, the basic structure of a simple plugin for the Overture platform has two sections: the core functionality in a folder in the core section, and the (optional) user interface in the IDE section.

The core functionality of the simple plugin imports—at a minimum—the VDM AST libraries and the typechecker, otherwise it is unlikely to be able to do anything. It will usually also need to import the VDM interpreter, if it should need to run or evaluate any portion of a VDM model. Beyond that, unless it interacts with some other portion of the core platform, the core functionality of a plugin should be self-contained.

The code in the IDE section imports its core section code as a library, and is written as a regular Eclipse plugin with the usual assortment of views, perspectives, and so on. It is also possible to write the IDE code as an extension of the main Overture IDE using the Eclipse extension points that defined in the structure of the Overture IDE. This conforms to the standard Eclipse model for plugins to the Eclipse platform.

V. Extensions from the DESTECS Tool

The DESTECS Tool is the first major extension of the Overture platform and is developed primarily in a separate source code repository from the main Overture repository. This has served as an existence proof for the claims that the Overture platform actually is extensible, reusable, and properly a platform in itself.

Structurally, the DESTECS tool is a combination of the VDM-RT-related components from the Overture platform, the necessary code to communicate with the external 20-sim [1] tool, and the coordinating co-simulation engine to manage the execution of a collaborative models (a co-model) on the Overture/VDM-RT side and the continuous-time 20-sim side. Note that 20-sim is used to model physical systems using differential equations and it has the ability to co-simulate such co-models.

In terms of build behaviour, the DESTECS build process takes the compiled libraries from the Overture platform as dependencies so that it can compile its own code. Once compiled, the DESTECS and Overture libraries are bundled together to form a standalone Eclipse-platform binary, and that is added to an installer with the 20-sim tool. The installer is the final product of the build process and is, of course, used by end users who wish to use the tool.

The DESTECS source code repository follows the same structure as the Overture repository, with the folders for code: core, IDE, and tools. In reverse order, the tools section contains only a tool to take a specification of the protocol used to communicate with 20-sim and turn it into Java files used by core components. The IDE section simply contains the necessary Eclipse perspective and views to use the functionality in the core section. Finally, the core section contains the control logic for communicating with 20-sim and the co-simulation engine for coordinating between VDM-RT and 20-sim.

The core section also contains a library that subclasses some of the core VDM-RT classes and provides a new task scheduler and the necessary logic to work with the co-simulation engine. This extension library is used to access VDM-RT functionality in place of the one provided in the Overture platform.

Overall, the DESTECS tool makes heavy use of the VDM-RT facilities—both in terms of core logic and in Eclipse IDE code—provided by the Overture platform. The contributions of the DESTECS code are primarily in the linkage to an external tool, the co-simulation engine, and the modifications to the VDM-RT task scheduler to support the co-simulation engine.

VI. The Vision for Intended Reuse in COMPASS

As discussed in Section IV Overture is comprised of three parts: an independent core, an IDE part integrating core parts into Eclipse and helping tools. COMPASS will adhere to that structure, and reuse will therefore be discussed in term of these parts.

A. Reuse in the Core Part

Presently the CML language is in its infancy and it is under active development. So far only the core of the CML language syntax has been defined, and the formal semantics remains to be completed. This leaves room for potential changes to the core of CML as the development of the formal semantics proceeds.

With the current state of the CML language, significant effort has been made to make CML expressions a proper subset of VDM expressions. The major gain in this approach is that it enables reuse of the formal semantics of VDM. But most importantly, with respect to Overture, it also enables reuse of existing parts of Overture. The key thing to enable this is that the generated AST of a CML expression construct must be identical to the corresponding VDM expression construct. So, the CML parser must build an AST that consists of nodes that are compatible with the existing VDM dialects. Generating the CML AST where expression nodes are compatible with the expression nodes of the VDM dialects is a matter of sharing the expressions portion of the ASTgen script. The connections for this are illustrated in Figure 2, where boxes with solid lines represent independent tools, boxes with dashed lines represent input files, and boxes with dash-dot lines represent running code in the Java runtime. Note that the figure separates steps done during the build of the tool from steps done during normal use of the tool; specifically, the build process uses Bison and ASTgen to create the parser and AST, while at runtime the generated libraries are used to process CML models into CML ASTs.

The AST generation process facilitates reuse by the COMPASS tool of existing Overture platform core parts that act on VDM expression nodes. This includes portions of the VDM typechecker, the VDM interpreter and any other analysis and operational code that processes expression nodes. Thus, the CML typechecker and interpreter are essentially getting expressions “for free” in cases where the semantics are equivalent. This kind of sharing benefits in both directions. Defects, errors and improvements found in one language now potentially benefit several languages.

The CML parser is generated by GNU Bison and this makes reuse of any part of the hand-written VDM parser infeasible. The choice to use Bison is due primarily to compatibility needs for interaction with other tools used within the COMPASS
project. Also, the cost involved in implementing a hand written CML parser, for the sake of reusing the expressions part of the VDM parser, is too high as compared to using a parser generator. Despite the use of Bison, the CML parser is still able to use an AST generated with ASTgen, maintaining compatibility with VDM expressions. This shows that Overture is extendable through the use of different parsing technologies, supported by the use of ASTgen.

B. Reuse in the COMPASS IDE

The structure of the COMPASS IDE will be very similar to the general Overture VDM IDE. It will be constructed as an Eclipse plug-in that provides a CML perspective with view for –at least– an editor, a simulator/interpreter, a debugger, and view for the various plugins.

The basic function of all these views is to process some given tree of AST nodes and deliver a graphical response. As a particular example the CML editor will receive a newly instantiated AST every time the user makes changes to the currently displayed CML model. In response to changes, CML keywords and syntax errors are visualized in the editor window. This functionality is already present in the VDM editor and we therefore intend to reuse it in COMPASS. In fact, corresponding views for all of the above views already exist in Overture for the VDM dialects, making all of them reusable in COMPASS to varying extents.

As noted in section VI-A this sharing has benefits in both directions. As an example one can easily imagine that the CML editor is extended to perform code completion (functionality assisting completion of partial language constructs) on CML. Since they have compatible AST nodes this feature could be extended to the VDM dialect. In the more general view we can imagine basic support for the most common language features to be available and easily extendable for new languages to come.

VII. RECIPROCAL DEVELOPMENT FROM COMPASS PLUG-INS

In addition to the COMPASS core and IDE, the following plug-ins will also be developed

- Proof obligation generator (POG)
- Theorem prover
- Model checker
- Refinement checker
- Static fault analyser
- Link to the RT-Tester tool for test automation [13]
- Link to Artisan Studio for SysML/CML integration [2]

The basic plug-in structure is identical to the general Overture structure, namely comprising a core and a GUI component. The core component will be independent of the GUI component, and is accessible through a common interface. The interface has methods that accept AST nodes, on which
the plug-in performs the promised analysis/operations on. The GUI component will utilize the core component and expose its functionality in an Eclipse view.

This structure follows that of already existing plug-ins in the Overture platform. For some of the COMPASS plug-ins, we can imagine them extended to support subsets of the existing VDM dialects. For others, they will probably have to be implemented in their entirety, which in many cases is nontrivial. However, the key thing here is that there is a possible feedback of development effort from COMPASS to the existing VDM dialects. We will briefly go through each of the COMPASS plug-ins to explain the potential avenues of feedback.

The planned CML POG will be inspired by the existing one for VDM [14]. It will detect all the places where a CML model potentially has inconsistencies and for each of these create proof obligations. The generated obligation will subsequently be made available for the CML theorem prover. For this link to the theorem prover, it might be possible to extend it to be utilised by the VDM POG.

Attempts in the past have been made in Overture to implement a theorem prover for the VDM dialects [15]. This however has not been developed to a degree where it is usable in general. The COMPASS theorem prover plug-in will transform CML into a format that can be processed by a theorem prover. If deemed successful for CML, this work could eventually be utilised to bring the original VDM/theorem prover plugin to a usable state.

The model checker plug-in will transform CML into a format analysable by a model checker. This work will probably not be usable in a direct way in terms code sharing, but more as a basic template with some parts being reusable for implementing one for the VDM dialects.

The CML refinement checker will support formal refinement of CML models of constituent systems as well as the refinement of protocols between models. The structure of this plugin could then eventually be used as a starting point for refinement/reification support in VDM.

The static fault analysis checking for CML will focus on bringing fault injection capabilities to the COMPASS tool when simulating a CML model. The framework for this plugin should be adaptable to the VDMJ interpreter, though it is less clear as to the structure necessary to allow this sort of interruption and modification of the interpreter’s behaviour.

For dynamic analysis of the CML models a link to the RT-Tester tool will be created, giving the COMPASS tool access to the full power of RT-Tester. The COMPASS RT-Tester plug-in and the RT-Tester tool will communicate via a JSON-based protocol, making it possible for other external tools to take use of this communication link. In the future this could be extended to support dynamic analysis of VDM models.

A mapping between CML constructs and SysML will be defined as part of the COMPASS work. This mapping will be utilised by the external tool Artisan Studio to support SysML/CML round-tripping. In this case round-tripping is simply the ability for Artisan Studio to export a CML model based on a SysML model, allow the user to manipulate parts of it in the CML plugins on the Overture platform, then return to Artisan Studio and have any changes reflected in the SysML model. A plugin providing limited support for round-tripping between UML and VDM already exists in Overture. It is easy to imagine that the development made here will be extendable for the existing Overture plug-in.

As described in this section there are many potential routes of feedback to the existing parts of the Overture platform from the COMPASS work. Both in terms of direct sharing and in terms of ideas.

VIII. SUPPORT FOR FUTURE EXTENSIONS

At the moment the Overture platform has been proven to be a flexible and adaptable tool. The source code in the main Overture repository supports three major VDM language dialects and a variety of plugins extend that functionality. The DESTECS tool extends the Overture platform into a tool capable of doing co-simulation in conjunction with an external continuous-time simulator. There are many improvements that can be made, however.

At the technical platform level, there are various tasks that need work. Testing is one important example: running a comprehensive test of the functionality of the tool is difficult. At present we can automatically test the core libraries, but to test the Overture IDE in the Eclipse platform we need to actually exercise the tool manually.

There are two ways of using the core VDM libraries: either through the Overture IDE in the Eclipse platform, or directly through the command-line interface. (Arguably the DESTECS tools counts as a third, but it is still Eclipse-based; and the COMPASS tool could become a fourth, but it is also still Eclipse-based.) It would be useful to create a third method: not only would it act as a check on our development to ensure that we do not depend too heavily on the Eclipse project, but at a more ambitious level, it could give us another route for integration with non-Java-based environments. As a particular example of a third method, an IDE based on jEdit\(^\text{10}\) would result in a light-weight instantiation of the Overture platform – jEdit plus the Overture core libraries would be about a quarter of the size of the Eclipse-based Overture IDE– that may provide an easier route into the VDM world for people not so familiar to Eclipse and other heavyweight IDEs.

Within the core Overture libraries there are several projects which would be of use, all of which are about generalising the platform’s language support even further. For users that would normally create a formal model by using refinement and data reification, the most surprising thing about the Overture platform is its lack of direct support for either feature. As noted earlier, the COMPASS project will need to implement refinement support for the CML language and this would be a good starting point to generalise this to refinement support for the VDM dialects.

The work to normalise the structure of the VDM dialect ASTs so that they are all generated by the same tool, with a large overlap between the individual ASTs, has included an effort to allow easy extension of ASTs. The method to this creates a new AST that is a strict superset of the old, merely

\(^\text{10}\)Hosted at http://www.jedit.org/.
by specifying the old AST description to import in the script file and then adding only the new elements. This gives the user the required new AST, and also allows a new AST object to be “flattened” into the old AST for use in components that do not (yet) support the new AST directly.

One of the things we do presently is create the ASTs for the VDM dialects automatically, and implement the typechecker, proof obligation generator, and so on using visitor-pattern interfaces present in the AST. This practice has prevented a large amount of duplication in our work, and the rare changes to the VDM ASTs take a relatively small amount of work because of this. It would be useful to build a generator tool for the typechecker that works in a similar manner, automatically generating the typecheckers for each VDM dialect from a set of typing rules, many of which would be held in common across the dialects. We can presently generate scanner/lexers, parsers, and ASTs; the next logical step would be to start generating the static analysis libraries.

Moving in the other direction from the AST, the lexer/parser code used to read VDM models is an efficient but hand-coded library specially built for the task. Although we have benefited from the performance of the lexer/parser, we would consider moving to a set of parsers generated by bison (or similar) for the VDM languages for the same reasons as made the COMPASS project choose bison for the CML parser. It is useful to have multiple tools able to read the same source files and parse them with the same grammar, even if they do very different things with the resulting AST.

IX. CONCLUDING REMARKS

The key enabling entity for reuse between languages in Overture platform is the single representation of common AST nodes. This allows the reuse of analysis and operational components, provided that the shared portions of the ASTs is relevant to the component, of course. In the VDM dialects there is a natural and substantial overlap of the individual dialects’ ASTs. However, as we have discussed, we can make it possible to fit in more distantly related languages in Overture through the use of the existing structure and tools in the platform. We also discussed the potential for feedback from the new plugins for CML back to the VDM dialects, which show a possible route for sharing to go in the other direction as well. This illustrates how the Overture platform can be extended beyond what was originally imagined. We therefore encourage others to participate by extending the Overture platform in a similar fashion, and to use the ideas in other projects; the former has immediate benefits, and the latter may allow for integration between platforms in the future.

Acknowledgements: First, the authors would like to thank our reviewers for their insightful comments into this paper and the Overture community. The authors would also like to thank all the stakeholders who have taken an active role in the development of the Overture platform. In addition we would like to thank the EU for the FP7 projects DESTECs and COMPASS that support the work related to this paper.
Towards an extensible core model for Digital Rights Management in VDM

Rasmus W. Lauritsen
Department of Engineering, Aarhus University
Denmark
Email: rala@iha.dk

Lasse Lorenzen
Bang & Olufsen
Email: llz@bang-olufsen.dk

Abstract—In this article two views on DRM are presented and modelled in VDM. The contribution from this modelling process is two-fold. A set of properties that are of interest while designing DRM systems are presented. Then, the two models are compared to elaborate our understanding of DRM elements and their interplay. This work is an exploration towards realizing a core model and terminology upon which extended models can be built.

Index Terms—Digital Rights Management, Core model, Entertainment System, VDM++, Systems of Systems

I. INTRODUCTION

As part of the B&O-case study in the COMPASS project\(^1\) we wish to model a range of properties for DRM systems such that B&O can integrate DRM into their products in a well-founded manner. Two properties of particular interest are security and composability. In this paper we study two theoretical models for DRM and compare the VDM models of these. This work contributes to establishing a core model for DRM which will then make up the foundation for further studies on the DRM properties. The core model for DRM should contain common elements for DRM such as Content Owner, Compliant Device and License. With a suitable core model it is our hope that modelling DRM properties becomes easier. As an example of this core-model approach suppose we wish to model a newly designed media player. Ideally, such work only takes a specialization of an existing core-model component to obtain a view on how well the new kind of device integrates with the rest of DRM. In general, we model properties and features for DRM systems using the terminology and modelling constructs from the core model. One aim for the core model is to contribute with a modelling framework for DRM to the VDM/Overture community, inviting others to study and model DRM systems.

This paper is organized as follows: Section II offers a bird’s eye view of DRM, framing what makes a DRM system. Section III elaborates on our research motivation which is based on B&O’s programme towards seamless integration of DRM in Home Entertainment Systems (HES). As this work also targets people outside the VDM and Overture communities we give a short background on VDM in Section IV. Following these preliminary sections we consider two views for DRM as candidates for our core model. The first model is that of Ku et al. [11]. Ku et al. analyse and survey DRM and aim at addressing all stakeholders in a DRM setting. Summarizing the Ku et al. model is the topic for Section V. The topic of section VI is our contribution on modelling the Ku et al. model in VDM. We derive the requirements to the DRM entities from their functional description in [11]. From this a generic VDM model is built. We conclude this section with an example using our generic VDM model to verify the property of consistent use of cryptographic primitives in a possible DRM system set up. The second model, summarized in Section VII, is based on work by Popescu et al. [9]. This model was chosen because it offers a completely different view on DRM by describing a security framework for DRM. The principal message in Popescu et al. is to describe a security architecture that allows DRM for Consumer Electronics (CE).\(^2\) This view gives a different flavour of DRM. Our contribution with modelling DRM in the style of Popescu et al. is the topic of Section VIII. Having modelled two different theoretical views with different purposes we compare the resulting VDM models in Section IX. As this work is in progress towards a unified model for DRM we list some of our ideas to future work in Section X. Finally, in Section XI we summarize our findings and conclusions.

II. DIGITAL RIGHTS MANAGEMENT

In this section we introduce the challenges that motivates the creation of a core model. Digital Rights Management is a common term for a variety of methods to limit or control the usage of digital premium content\(^3\) and the distribution mechanisms involved. For most people the understanding of DRM is related to restrictions on the content. For an example these restrictions could be on how many playbacks are allowed, who can play it and where. Other common restrictions are copyright protection and similar redistribution limiting mechanisms [10]. Strict requirements are also imposed to the entire chain of DRM entities, from copyright holder to the consumer. The

\(^1\)This research is funded by EU FP7 COMPASS project under grant agreement no. 287829. (http://www.compass-research.eu)

\(^2\)Consumer Eletronics is electronic equipment intended for everyday use most often in entertainment, communications and office productivity.

\(^3\)Premium content and services are those that the user explicitly pays for.
DRM system specifies who signs content, how it should be signed, how to store keys for decryption [9], [11]. Focus on copyright holders and content providers is a common trend for DRM systems today. As a result DRM systems are seen as a nuisance by most users. In complex multimedia configurations such as a Home Entertainment System, DRM is becoming a complex issue for both users and multimedia manufacturers.

As described in this section, DRM is much more than just rights management — it is just as much a mechanism to describe all of the entities in the digital content domain.

III. HOME ENTERTAINMENT SYSTEMS

A modern Home Entertainment System (HES) is often composed of TVs, audio-players, other supporting devices and content sources connected to each other, often through a local network [8], including the ability to get premium content from the Internet. A device in a HES is something that either can play content or control other devices. Devices can range from a TV, an IR-remote-control, to a tablet-computer or a laptop and anything in between such as phones and speakers. The content sources in such a HES, range from DVD or BluRay to content stored on own Network Attached Storage (NAS) or Internet services such as YouTube\(^4\), Dailymotion\(^5\) and Metacafe\(^6\) or premium Internet services such as Netflix\(^7\) and Spotify\(^8\) [1]. There is the possibility that content can be provided by a temporary device, such as an USB-stick or a smartphone, and this content will come and go as the device enters or leaves the HES.

Up until now, there has been no or little understanding of the user identity concept in HES systems. Controlling such systems is usually carried out by means of a remote-control that has no indication of who is using it. With many different activities going on at the same time, there is no way a HES can relate users to activities. But with personalization and premium services it has become important to be able to identify who is using the HES and where.

DRM is one of the most important factors to consider in the HES. If the HES is not compliant with DRM regulations, then only user-generated or unlicensed digital content can be used in the HES. DRM compliance is often necessary to access premium services and other types of paid content that the consumer expects from a HES. The DRM compliance in modern HES is however not easy as described in Sohn [10]. A HES is an additional feature emerging from a set of devices and sources when they cooperate. However, given that most devices and sources live their own life independently of the HES it is not possible to pinpoint where the exact HES is located. This makes it very difficult to validate a HES for compliance with the DRM regulations. Currently the method is to guarantee compliance on all devices and sources and then assume that the HES is then also compliant. A formal modeling technique such as VDM will help understand if a HES is in compliance with the given DRM regulations to ensure compliance.

IV. VDM

The Vienna Development Method (VDM) was created at IBM’s Vienna Labs in the 1970s. VDM is one of the longest established formal methods in use today. A core concept of VDM is models, and how these models can be used to investigate and understand properties as well as mitigate complexity of systems. Using a model-driven engineering approach and formal methods to understand new systems or problems is a commonly accepted good strategy see Fitzgerald et al. [3]. It has been successfully applied in many industrial settings see Woodcock et al. [12]. To express concise models VDM comes with a formal language. A format language is a language that has a precise mathematical semantic, allowing us to annotate our operations and types with invariants, pre- and post-conditions, using a precise discrete mathematical syntax.

For a good modern text book on VDM see Fitzgerald et al. [4]. The first standardized version of the VDM language is VDM-SL. In this paper we use the language variant VDM++. VDM++ is derived from VDM-SL which extends VDM-SL with an object-oriented notation among other things.

We are using VDM in a retrospective setting as we are using VDM to comprehend two proposed views on DRM. Besides utilizing an understanding of the two DRM models we gain from creating a static VDM model, we are building models that can be executed and investigated by means of run-time manipulation. We are also taking advantage of the Overture Traces extension, see Larsen et al. [6]. The traces extension allows us to generate and execute combinatorial tests against various properties in the model.

V. THE KU ET AL. MODEL FOR DRM

In this section we summarize how DRM is viewed in Ku et al. [11]. The elements Ku et al. find in DRM are summarized in figure 1 on the following page. The interplay between these elements is briefly summarized here:

1. The Content Owner inputs some new content to the Content Protection mechanism with a description\(^9\) of how it may be used.
2. The Content Protection mechanism of the DRM System responds with a Protected version of the content and a license.
3. The Content Owner distributes the Protected Content through the Distribution mechanism.
4. The Content Owner sends the licenses to the License Broker which sells them to viewers.

\(^4\)http://youtube.com
\(^5\)http://dailymotion.com
\(^6\)http://metacafe.com
\(^7\)http://netflix.com
\(^8\)http://spotify.com
\(^9\)Such description are the rights granted on the content by the resulting license and are typically expressed in a Rights Expression Languages (REL). In this model we have a quiet simple version of REL and may be subject to future work.
5 The End User acquires material from the Distribution mechanism and examines its meta-data to figure out which license to acquire in order to play the content.
6 If the viewer does not already have a license for the acquired content, one can be bought from the License Broker.
7 When a viewer buys a license from a License Broker two things happen, a license is transferred to the viewer and money is transferred to the License Broker.
8 All or some of the money transferred to a License Broker is subsequently transferred to the appropriate Content Owner.

Ku et al. focuses on six important areas that a DRM model typically covers. The first "Content Identification and Meta-Data" is concerned with identifying content uniquely and enclosing meta-data to give a user-friendly interpretation of the content identifier. Secondly "User Identification/Authentication" is treated and included from the need to identify paying users in a secure way. Third main topic to cover is, "Digital Watermarking" which is a set of techniques to alter the content without changing the experience of listening or watching the content while still allowing forensics to identify a particular piece of content in part or as a hole. Fourth on the list of things to consider is “Content-Based Identification”, which is about identifying the content by producing a fingerprint of it. The main difference between watermarking and fingerprinting is that watermarking is intrusive; changing the content, whereas fingerprinting is passive. However, fingerprinting does require maintenance of a database mapping content to fingerprints. The fifth concept of DRM is that of "Secure Containers" which covers means of restricting access to the content, e.g. by encrypting it. The sixth and last component considered is “Rights Expression Languages" (REL). REL’s allow its users to articulate under which circumstances content can be used. With this summary of what need to be included in a DRM system from Ku et al. we continue in the following section by presenting our work modeling it in VDM.

VI. MODELLING DRM LIKE KU ET AL. IN VDM

This section presents work carried out for creating a generic VDM-model representing the view on DRM systems laid out in [11]. The VDM model is generic in the sense that it comprises the elements necessary to model any concrete DRM system.

The following describes each of the constituent model elements available in the VDM model for describing a given DRM system. Section VI-B on the next page gives an example on how to utilize the model elements to model a concrete DRM system setup and allow the Overture Tool to check for cryptographic parameter consistency.

A. The DRM model

The approach for creating this model follows techniques in Larsen et al. [5]. Thus, in addition to the DRM entity classes a World class for entry points and an Environment class modeling interaction with entities outside the core-DRM system are added. For a comprehensive treatment of VDM++ see Fitzgerald et al. [4]. Figure 1 describes the principal elements in the Ku et al. view on DRM. From this figure the following VDM++ classes immediately spring out: Content Owner, Content Protection, Content, Protected Content, License, License Broker, Viewer (Renderer) which is depicted as user on the figure, Distribution and Money.

To determine which properties and operations to include for each of the entities above the following sections derive their requirements based on their operations described in section 2.2 of Ku et al. [11].

1) Content Owner: The content owner initially has some "raw content". They may be forced to format this raw content in a particular format as required by the DRM system. Optionally, the content owner may add a watermark to the content. As part of entering their content to the DRM system a set of rights must be specified for the content. The DRM system will produce a set of licenses and a protected version of their content. These will be disseminated to relevant License Brokers and a distribution channel respectively. Upon a purchase of a license between a license broker and a viewer the content owner may receive payment. These operations give raise to the following operations and instances variables on our VDM ContentOwner class. The details on the operations in each operation and function below how are available in a technical report at AU by the present authors [7].

2) Content Protection: is the component of a DRM system that encrypts and packages content for distribution.
This component also creates licenses for accessing Protected Content. The fact that Viewers need to access the protected content creates a link between the Content Protection and Viewer entities. This link is not made visible in Ku et al. As a first attempt to capture this link the model implements the idea of a Protection Domain in the VDM++ class ProtectionDomain. Viewer’s and ContentProtection classes are both instantiated with a ProtectionDomain instance. Enforcement of access limitations are implemented in the Viewer class that makes sure only to access un-protected content or ProtectedContent instances created by a ContentProtection instance in the same domain as it self.

3) **Content**: represents raw content before protection. Content has two properties namely its Format and Watermark. Naturally, our VDM++ class Content carries these two properties. Otherwise content is abstract only carrying an additional id for comparison.

Protected Content: is Content which has entered the DRM system and has been protected by some Content Protection entity. Protected content is modeled using aggregation rather than inheritance here to reflect the fact that the content is “inside” the Protected Content. Our ProtectedContent classes therefore carries two instances one for its content and one for the protection domain that protects this content.

4) **License**: grants access to perform a set of actions with a piece of content on a Viewer in some Protection Domain. Therefore, the License class naturally carries a pointer to an instance of the ProtectedContent class and an instance of RELInstance. The license is basically an immutable data carrier and its values could be public. However to, facilitate open behavior for extension models it is a design choice to have accessors methods instead.

5) **License Broker**: receives Licenses from ContentOwners and set these available for sale. This enables the LicenseBroker to serve a Viewer with a License for a given piece of ProtectedContent. Transferring a license to a Viewer given an instance of Money and an identifier for content is the operation SellLicense. SellLicense may fail if no license can be found and in such cases it returns the value <None>. One interesting detail unclear in the Ku et al. article is how and when money is transferred from the LicenseBroker to ContentOwners. In this model it is captured with the method GetPayment to be called by a ContentOwner in case money is relevant.

6) **Viewer**: a Viewer is capable of playing back content protected by a Content Protection instance in a Protection Domain the Viewer can access. Hence, Viewer’s carry a list of Protection Domains from which they can access Content. To actually access Content (or try to) a Viewer has a Play operation which takes a piece of content as argument and returns a boolean stipulating whether access was successful or not. The Viewer also carries a set of licenses that have been acquired through invocation to its BuyContent operation. To support playback of Content a viewer has a PlayContent operation.

7) **Distribution**: allows ContentOwners to publish Content and allows Viewers to browse all published content. Hence, the Distribution has a PublishContent operation and a BrowseContent operation.

B. Modeling consistent cryptographic parameters in a DRM system

In this section an example using and extending the core model elements elicited from Ku et al. above is presented. It is our goal to create a model that allows the DRM system designer to model different kinds of encryption for protection and for accessing content in the Viewer.

The kind of encryption used is stipulated by parameters such as, algorithm, mode of operation etc. The model of crypto-parameters here is built on [2].

A scenario is created such that if the crypto-parameters are correct it will successfully play a piece of music. If the crypto-parameters are incorrect, the model should fail to play back the content.

Five additional elements are added to the model. One element, CryptoExample, contains the scenario mentioned to run see Listing 1. Another element, CryptoParameters, model primitives from which concrete encryption and decryption suites (collectively referred to as CipherSuites) can be set up. To instantiate this model three new elements were created. These are CryptoContentProtection, CryptoViewer and CryptoProtectedContent.

ContentProtection creates CryptoProtectedContent which in addition of being ProtectedContent also carries which cipher suite is necessary to play back the content at the CryptoViewer site.

Below a listing for the scenario which also illustrates how a DRM system is set up, here with the Crypto extensions elements created for this example.

In this section, core model elements for DRM based on the view in [11] have been derived. From this core model an example demonstrating that one property of DRM, namely consistency of cryptographic encryption/decryption parameters, can be modeled and checked by the Overture tool.

VII. THE POPESCU ET AL. MODEL FOR DRM

This section presents the view on DRM in Popescu et al. [9]

A. Summarizing the model

DRM systems rely on the constraint that they are only operating on compliant devices. A very important aspect of the compliant devices is that they are self-policing: this means that before playing any content they validate that the content is allowed to be played on said device. The paper proposes a model to ease the users experience with DRM and handle the self-policing and compliance more efficiently for products

---

10VDM source files are available on request.
class CryptoExample
values
    -- To have a complete DRM world we need to following instances:
    public www : Distribution = new Distribution();
    public aes_cb_cbc_zeropad_content_protection : ContentProtection =
        new CryptoContentProtection(CryptoParameters'AES_ENC_CBC_ZEROPAD);
    public phil_collins_against_all_odds: Content =
        new Content(mk_token("Against all odds"));
    public virgin : ContentOwner =
        new ContentOwner( mk_token("virgin music inc."),
            {phil_collins_against_all_odds});
    public virgin_enabled_viewer =
        new CryptoViewer( CryptoParameters'AES_DEC_CBC_ZEROPAD );
    public amazon_shop : LicenseBroker = new LicenseBroker();
operations
    public static PublishAndPlayScenario: () ==> ()
    PublishAndPlayScenario() ==
    {
        -- Virgin publishes phil collins's Against all
        -- odd on www selling licenses via Amazon.
        dcl rights: RELInstance := virgin.SpecifyRights({<Play>});
        dcl a : Content := virgin.FormatContent(phil_collins_against_all_odds);
        dcl b : Content := virgin.AddWatermark(phil_collins_against_all_odds);
        dcl c : ProtectedContent * License
            := virgin.EnterContent( phil_collins_against_all_odds,
                rights,
                aes_cb_cbc_zeropad_content_protection);
        virgin.DisseminateContent( www );
        virgin.SendLicenses({amazon_shop});
        -- Virgin viewer Acquires against all odds from www,
        -- buys a license, plays phil collins content
        virgin_enabled_viewer.BuyLicense ( amazon_shop,
            phil_collins_against_all_odds.id); --Acquire license
        if (virgin_enabled_viewer.PlayContent( phil_collins_against_all_odds.id,
            www ))
        -- Acquire content and play it
        then
            IO'println("Crypto is ok, we are able to play.")
        else
            IO'println("There is a mistake in the model, unable to play")
    };
end CryptoExample
in a HES. Popescu et al. list four characteristics of a typical HES affecting their view on DRM designs.

- Continuous network connectivity among all devices in one domain cannot be assumed.
- The existence of secure clocks cannot be assumed.
- Devices should not require cryptographic hardware accelerators in order to operate efficiently.
- It is essential that the design allows for revocation of potentially large numbers of counterfeit devices without significant performance degradation.

The first requirement is essential as portable music and video players can be in one HES domain. With modern requirements for low power consumption, maintaining wireless network connectivity is a serious power drain. The second and third characteristics influencing the cost of producing HES product, and it is not feasible to increase production costs even by a few cents. The last characteristic is due to the fact there are many of counterfeit products on the market. With revocation it is possible to make a decent attempt to make it a hassle to use counterfeit products.

The design goal of the proposed DRM design is to have content seamlessly float between products in a authorized domain. Popescu et al. motivate their work with the example of an user downloading song from a portable music player to the car stereo, even when both the devices are disconnected from the authorized HES domain.

B. System Model

The view on DRM presented by Popescu et al. in the paper consists of the following entities.

- Licensing Authority - Controls the certificates and revoke certificates if needed.
- CE Manufacturer - Receives a license to certify the Compliant Consumer Electronic (CE) Devices.
- Compliant CE Devices - Used to render content provided in compliance with the usage rules set by a Content Provider.
- Content Provider - Distributes content to the Authorized Domain and set the usage rules on the content. Popescu et al. only consider online content.
- Content Manager - Is a role that some Compliant CE Devices can have, which means they managed access to the content they contain. This content is supplied to the entire authorized domain.
- Domain Manager - Also a role played by a Compliant CE Device. This is the device responsible for registering and deregistering devices in the Authorized Domain and enforces that the domain remains authorized.
- Authorized Domain - Is the collection of Compliant CE Devices, Content Managers and Providers and one Domain Manager.

In this model it is important to understand that each Compliant CE Device can perform multiple roles at once. One device can be Content Manager, Domain Manager and Content Renderer without breaking the DRM design. Figure 2 show the system design in graphical notation with indication of how the control and content flow is assumed.

Popescu et al. present compliance checking to a level of detail which is unnecessary for the purposes of this paper and are omitted from our VDM model.

VIII. MODELLING THE POPESCU ET AL. DRM IN VDM

In this section we present our work on modelling the Popescu et al. view on DRM in [9]. The purpose of the VDM model is to model and validate the higher level system model that Popescu et al. presents in their paper.

A. VDM model with properties

The first step was to create VDM++ classes for each of the entities in section VII-B.

Notice that the Authorized Domain class is deliberately omitted from the list. In the system model in section VII-B the Authorized Domain is described as having a Domain Manager, a Compliant CE Device and a Content Manager. In other words, the Authorized Domain is not something that would make sense to create as class. The rest of the entities are modeled as classes in the VDM model and described in more detail in the following. Secondly it could be argued that the Compliant CE Device can play the role of Content Manager and Domain Manager besides being the Content Renderer that there is no need for the Content Manager and Domain Manager class. However the concept behind all three entities are so different that the model is stronger when clearly separated in three parts.

1) LicensingAuth: The LicensingAuth class is VDM++ equivalent of the Licensing Authority in the section VII-B system model. The main functionality of this class is to be able to control a list of valid and invalid certificates. Also included in the class’s functionality is the ability to provide both Content Providers and CE Manufacturers with updates on valid and invalid certificates. If necessary the LicensingAuth can also create new certificates.

2) CEManufacturer: For the CEManufacturer there are two identifiable functions. One is the ability to get the valid certificates from the LicensingAuth and the other is to stamp these certificates into the CEDevices produced.

3) CEDevice: A CEDevice is a device that renders content, and thus is able to get the content to be rendered from the Content Manager and is also able to enter and leave the Authorized Domain.

4) ContentProvider: Provides the Authorized Domain with content. It provides the ContentManager with information of available content and also get information from the LicensingAuth regarding valid and invalid certificates.

5) ContentManager: This ContentManager gathers the content from the ContentProvider and make the content available in the Authorized Domain. The ContentManager can enter and leave the Authorized Domain.

6) DomainManager: The DomainManager is the one who holds a registry of all the devices in the Authorized Domain and make sure that devices without compliance to the Authorized Domain can enter the domain.
class World
instance variables
licensingAuth : LicensingAuth;
domainManager : DomainManager;
spotify : ContentProvider;
synology : ContentManager;
phillips : CEManufacturer;
phillipsTV : CEDevice;
phillipsRadio : CEDevice;
certOne : Certificate;
certTwo : Certificate;

operations
public Run : () ==> ()
Run() ==

--setup and initializing all the instance variables
phillips := new CEManufacturer();
phillipsTV := new CEDevice();
phillipsRadio := new CEDevice();
phillips.AddDevice(phillipsTV);
phillips.AddDevice(phillipsRadio);

-- Creating two certificates both valid from the start
certOne := new Certificate(true, "certOne");
certTwo := new Certificate(true, "certTwo");

-- Creating the LicensingAuth with the two certificates
-- The licensingAuth is now in control
-- of if these certificates are valid or not
licensingAuth := new LicensingAuth({certOne, certTwo},{});

-- Giving the certificates to Phillips for use (we need two different)
def = licensingAuth.GiveCertificateToCEManufacturer(phillips) in skip;
def = licensingAuth.GiveCertificateToCEManufacturer(phillips) in skip;

-- Certifying the two products
def = phillips.CertifyDevice(phillipsTV) in skip;
def = phillips.CertifyDevice(phillipsRadio) in skip;

synology := new ContentManager();
domainManager := new DomainManager();
spotify := new ContentProvider("spotify");

-- They need to subscribe to the updates on certificates
licensingAuth.Subscribe(spotify);
licensingAuth.Subscribe(domainManager);

-- We need to register the CEDevices add the domainManager
domainManager.RegisterDevice(phillipsTV);
domainManager.RegisterDevice(phillipsRadio);

-- The ContentProvider needs to be added to the ContentManager
synology.AddContentProvider(spotify);

--Finally the ContentManager must be registered in the DomainManager
domainManager.RegisterContentManager(synology);

-- Now the domain is completely configured
-- This prints Domain Valid
if(domainManager.IsValidDomain()) then IO'print("Domain Valid")
else IO'print("Domain Invalid");

-- Let us now invalidate one of the certificates
-- Know certTwo is an invalid Certificate
licensingAuth.InvalidateCertificate(certTwo);

-- This prints Domain Invalid
if(domainManager.IsValidDomain()) then IO'print("Domain Valid")
else IO'print("Domain Invalid");

-- In order to make the domain valid
-- the offending device must be removed
if( not licensingAuth.IsCertValid(phillipsTV.ReadCertificate()) )
then domainManager.DeregisterDevice(phillipsTV);
if( not licensingAuth.IsCertValid(phillipsRadio.ReadCertificate()) )
then domainManager.DeregisterDevice(phillipsRadio);

-- This prints Domain Valid as the domain is once again valid.
if(domainManager.IsValidDomain()) then IO'print("Domain Valid")
else IO'print("Domain Invalid");

end World
B. VDM example of the model

In this section there is a complete setup and run in the Popescu-based VDM model. The following steps are shown in the VDM model. First all the configuration and setup is handled, followed by a test of whether the configured model is valid. Next we try to invalidate one of the certificates and test the validity of the model. As the last step, the offending device is removed from the domain and a last validation is performed.

C. Findings

In the Popescu et al. model devices are compliant as long as their certificates are not invalidated. The License Authority tracks which certificates are published to manufactures. Whenever a product series is found to be counterfeit the License Authority invalidates all certificates issued for that series of devices. The Content Provider is notified about invalidated certificates by the Licensing Authority. In turn Authorized Domains represented by Domain Managers are notified of invalid certificates when downloading new content from a Content Provider. The first thing was to test the mechanism of having an Authorized Domain and see how this could be manipulated. If there was a constant connection between the Content Provider and Authorized Domain it was impossible to end up in a scenario where the Authorized Domain would remain authorized with counterfeit devices in the domain. This is due to the fact that in the instant the Licensing Auth receives information about counterfeit devices the certificates for these are revoked and this information propagates into the entire Authorized Domain from the Content Provider. The only way to get the domain to remain authorized with counterfeit devices would be to never access the ContentProvider for new content, and then the damage is limited as the domain only holds content already paid. So in general the concept of Authorized Domain seem to work as intended.

In relation to what B&O expects from a HES, the Popescu et al. view on DRM is missing a few properties. First, the Popescu et al. model makes the content an important but implicit part of their view on DRM. However in order to really understand the users' experience with the DRM model, we need to explicitly define what content is and how it acts in the DRM model. Usage rights are described in the paper, but it is not described how this affects the model and therefore the DRM talked about in the paper is mostly focused on the certification and compliance of device rather than on how content should be treated in a DRM model. Lastly there is no user, which is consistent with the fact that the paper mostly targets certification and compliance, but is nonetheless an important element.

IX. Comparison of the two DRM models

In this section we compare the two models from Popescu et al. [9] and Ku et al [11]. We found the following significant differences during the modeling process:

1. Ku et al. aims at being complete model by including all aspects of a DRM system. However, this is done in such a way that the focus is mostly on the distribution chain and leaves out the user's home configuration. Popescu et al. assumes that the distribution chain is simpler than in the Ku et al. but has a strong focus on what is installed in the users home and how to keep such a configuration authorized. We can see the difference in the two models in the details of the VDM models. Models based on Ku et al. do not have the details of CE devices and content managers etc. that the Popescu et al. This results in a VDM model that cannot be used to verify anything about
the configurations in the user’s home, but is strong in the distribution area. And if the goal is to model the users experience with the DRM system, the Popescu et al. model is a better choice.

2 Ku et al. models lack a way of distributing license access information to end user devices. For the VDM model this means the “world” and “environment” classes become more complicated: scenarios set up prior to running the model are for this reason complicated with creating security domains and assigning these to end user devices and Content Owners. This may be wanted when modelling scenarios where the DRM system designer has specific demands on the security set up level. However, from the a point of view where only user (the end user ranging over all human entities in the model Content Owner, License Broker and the paying user playing some content) operations are interesting this complication is unwanted. In Popescu et al. the CE Manufacturer class takes care of adding any material needed in devices for participating in a DRM context.

3 Another important difference is that Ku et al. is splitting what Popescu et al. call the Content Provider into two pieces: the License Broker and the Distribution. This implicitly means that Popescu et al. assumes that licenses and content are distributed together whereas Ku et al. opens the possibility of having different entities distributing and selling licenses. Popescu et al. assumes an online digital distribution only and does not take physical media like CDs and DVDs into account. Therefore, in their model it is reasonable to assume up-to-date licenses can be downloaded with the content which is clearly not possible when including hard-media like BluRay.

4 There are no distinct entities in the Popescu et al. model similar to Content Protection for handling content protection. This is because the focus for Popescu et al. is designing a framework for handling DRM security and as such all operations described for the existing entities are concerned with content protection.

X. FUTURE WORK

A first step towards creating a core model for DRM is to bridge the two models presented in this paper. Section IX lists the differences we have found and and resolving these will be the first steps on a path to a core model for DRM. We aim to use VDM as a tool to help us understand if the new core model is in compliance with the two base models. If we need to discard elements from either the Ku et al. or Popescu et al. models, we will use VDM to help argue that such alterations are still consistent with the nature of these models.

A. Properties to verify

As a first concrete step in a path to a core model we will to model the properties described in the following sections.

1) Composability: The property of composability is about checking that new kinds of devices can be added to a DRM context, without breaking the DRM compliance of the system.

2) Rights Expression Languages: An in-depth treatment of Rights expression Languages (REL) is not in scope in either of the two articles and this paper. There is a large body of material on modeling REL which does not take the rest of the DRM content into account. The challenge for our core model is to be open such that different kinds of RELs can be supported.

3) UPnP-AV: The transport and command layers of HES systems are often implemented using UPnP-like links. UPnP-AV in particular is of interest for our case study in the COMPASS project. Thus, including a reasonable support for modelling HES device links and possibly core properties of UPnP-AV will be of interest.

Content Identification and Meta-Data: Ku et al. list a number of identification schemes which might be used like BAN, ISBN. A model that would give an idea of the complexity and work imposed by using a particular identification scheme would allow a designer of DRM systems to make an informed decision when choosing identification scheme.

User Identification/Authentication: Schemes for user identification and authentication are complex, containing cryptographic operations and accompanying key management. As with all complex constructs it is reassuring to have a model that shows a particular approach is sound. That is, users are given identifiers and authentication credentials that are actually meaningful and usable with other components in the DRM system. This property of a DRM can be assured by running a set of common scenarios on the model.

Digital Watermarking: Consider a model when content can exist in its raw form without watermarking and protection of any kind (e.g. as the artist and content owner created it). One property to check, in this case, could be content intended to be commercially sold reaches end user devices only in watermarked form.

Content-Based Identification: Fingerprinting content requires that a database that maps the content to its fingerprint is maintained. For fingerprinting to be interesting in a model, the model must include the ability to tamper with content in some way. That is, we wish to check that the fingerprinting mechanism allows us to identify content if it is played back by some End User device in the model.

Secure Container: The secure container is typically some form of encryption of the digital content. The Viewer or compliant device has a trusted storage mechanism where it has some cryptographic material allowing it to access the digital content. There are several interesting properties to verify for an implementation of this protection mechanism

Cryptographic Configuration: Check that for relevant scenarios the mechanisms applied in the end user device are compliant with that of the Content Owners. If the Content Owner encrypts his content with AES then the end user devices must not use 3-DES or some other algorithm.

28
XI. CONCLUSION

The process of modelling two views of DRM in VDM has provided us with points where the views were not telling the whole story. For example, the Ku et al. paper is not concerned with distributing cryptographic material to Viewers and Content Owners. Nonetheless this distribution is vital for making a faithful model, in particular when checking the consistency of cryptographic parameters.

An important point brought forward by making the Popescu et al. model and B&O’s interest in HES is that modelling the Viewer (as in Ku et al.) as one entity is not enough.

We find Popescu et al. to be good for establishing a secure but flexible DRM environment between CE devices. However, the Popescu VDM model is also found to lack certain elements, like explicit modeling of content, in order to serve as a complete model for DRM in a HES setting.

Working with the two articles on DRM and later modeling their views on DRM has provided two points. First, we are closer to finding what makes a widely usable core model for DRM. Secondly, we have listed properties with DRM that could be of interest to model in future work.

The next iteration on the way towards defining a core model for DRM is to synthesize our experiences from this work. We have found that the Ku et al. model provides a reasonable set of stakeholders except it omits the manufacturer of devices. With respect to entities (like license, content, money) the Ku et al. model lacks devices at the end user and modelling of their interactions. Hence, a promising way ahead is to take the Ku et al. model and extend it with the notions of devices and manufactures from Popescu et al. Based on this preliminary exploration we know where to go next and believe a comprehensive framework for rapid modelling of an evolving DRM system can be developed.

ACKNOWLEDGEMENTS

This work was supported by the EU FP7 COMPASS Project under grant agreement no.287829, Bang & Olufsen and Aarhus University. We would like to thank Peter Gorm Larsen and Bjørn Reese for providing us with valuable feedback. We would also like to thank Joey Coleman for rigorous editorial corrections.

REFERENCES

Towards a Co-simulation Semantics of VDM-RT/Overture and 20-sim

Kenneth Lausdahl, Joey W. Coleman and Peter Gorm Larsen
Department of Engineering, Aarhus University
Finlandsgade 22, DK-8200 Aarhus N, Denmark
jwc@iha.dk, kel@iha.dk, pgl@iha.dk

Abstract—When combining different formalisms that use entirely different underlying paradigms it is challenging but important to appropriately record their joint semantics. Such combined models (co-models) may be simulated using a co-simulation engine. This paper provides the initial foundations for co-simulation between the VDM Real-Time dialect, based on traditional discrete mathematics, and 20-sim, based on Bond Graphs that model collections of differential equations in a continuous-time setting. The formal semantics of this co-simulation is based on earlier work of Verhoef but here we present it using the Structural Operational Semantics (SOS) format.

I. INTRODUCTION

The process of modelling digital control systems that are meant to affect the physical world is a significant challenge [HS07]. Models of the physical world often involve differential equations that describe how the world changes over time; these models are based on continuous mathematical domains. Models of digital controllers—often software-based—usually do not reference time at all and instead treat the model of the controller in terms of discrete events that trigger specific reactions from the controller; these models are based on discrete mathematical domains. A common model that can be analysed is needed in order to be able to appropriately balance concerns from control engineering with software engineering and taking potential faults into account [Ver09].

Combining such different formalisms based on entirely different paradigms must be done with care. This paper describes the approach taken by a European R&D project called DESTECS1 [BLV+10], [DES09], with focus on the semantic description of the co-simulation and, in particular, the semantics of the real-time dialect of the Vienna Development Method (VDM) used to model digital controllers. We call these common models “co-models” consisting of a “discrete event” (DE) model and a “continuous time” (CT) model, and the joint simulation of these is called a “co-simulation”.

VDM [BJ78], [FLV08] has three dialects: The ISO standard VDM Specification Language (VDM-SL) [FL09], the object-oriented extension VDM++ [FLM+05] and a further extension of that called VDM Real Time (VDM-RT) [VLH06], [HV10]. All three dialects are supported by an open source tool called Overture [LBF+10]. An executable subset of the different VDM dialects—including non-deterministic elements—can be simulated using an interpreter feature [LLB11]. In the DESTECS project this interpreter is used for the simulation of the “discrete event” part of the system. Using VDM-RT it is possible to define a distributed architecture with multiple CPUs and busses connecting them. Multiple threads may be present on each CPU and the scheduling policy for these is parametrized per CPU. Execution of a VDM-RT model alternates between regular computational steps and steps that update and control system timing.

For the CT part of the system we make use of a commercial tool called 20-sim which is a multi-domain modelling and simulation package for modelling complex physical systems [Kle06], [vA10]. This tool is able to simulate collections of differential equations and its semantic basis is on Bond graphs [KR68], [Bro90].

In the DESTECS project the Overture and 20-sim tools are run in parallel to enable co-simulation of the co-model. In practice this means that both tools are running simultaneously and are acting as slaves to a master for the co-simulation. The master then ensures that both simulators first are initialised appropriately and then it allows one of the tools to take a small time step before control is handed over to the other tool to simulate a similar time step. This kind of lockstep operation is carried out in both a time-triggered and an event-triggered fashion so that synchronisation may happen either at set points in time or due to an event having been triggered.

In Section II we present the top-level view of the structure of the DESTECS tool, giving an introduction to Structural Operational Semantics as well as the semantics we use for co-simulation. Section III goes into detail about the semantics of VDM-RT with specific emphasis on how pending values are held and on synchronization issues with Duration statements. Due to space limitations, we only cover selected elements of VDM-RT semantics. Among the things omitted are the details of inter-CPU communications in the model and how thread priorities work during context switching.

We propose an optimization for the implementation in Section IV and outline the direction of our future efforts. Finally, we conclude this paper in Section V, noting some areas in the developing semantics that are the subject of future work.

As a framing assumption for the entire paper, we assume that the reader is familiar with basic VDM-SL notation for expressions and functions, such as described in Dawes’ reference [Daw91]. We are not assuming any particular familiarity.

1This is an acronym for “Design Support and Tooling for Embedded Control Software”.

with VDM-RT or 20-sim in detail.

II. CO-SIMULATION STRUCTURE

Conceptually the structure of the co-simulation used in the DESTECS tool consists of three pieces, as shown in Figure 1. The co-simulation engine coordinates execution, determining which of the simulators is to execute the next step. It also mediates the exchange of shared state variables and events between the simulators and records the necessary time values to keep the simulators synchronized.

The 20-sim block in Figure 1 represents the 20-sim CT simulator. Internally, this tool uses bond graph semantics to represent physical models; however, for our purposes any CT simulator that provides an interface that conforms to the semantics in Section II-B could be used.

The VDM-RT block in Figure 1 corresponds to the Overture DE simulator using the VDM-RT language dialect. We give an overview of the semantics of VDM-RT in Section III.

In [Ver09] Verhoef provides an initial description of the VDM-RT semantics and co-simulation engine but it is presented in a style that puts the VDM-RT simulator in control over the overall system. One of the goals in this work is to give a semantic model that abstracts the co-simulation engine away from the VDM-RT simulator and places control of the overall system with the co-simulation engine. This has two primary advantages: the first is that this provides a clear, modular architecture; and the second is that we can now sensibly consider requirements should we choose to replace either the DE or CT simulators.

A. Structural Operational Semantics

We use a Structural Operational Semantics (SOS) format [Plo81], [Plo04] to present the semantic definitions in this paper.

A SOS description consists of two major elements: a set of type definitions that describe the static structure of the system; and the definitions of the transition relations that describe the behaviour of the system. The type definitions may also be accompanied by context conditions — further constraints on the types that are analogous to the static checking done by a programming language compiler.

Our definitions in this paper use the basic VDM-SL type system and VDM-SL expressions to define the static structure of the co-simulation engine and VDM-RT language. Circularity in the definition is not an issue here as VDM-SL has a well-defined denotational semantics [LP95] and, in any case, it is possible to give these definitions in terms of other notations.

In a SOS definition, the entire system is modelled as a configuration containing all of the information needed to capture the state of a system at any given point. This includes, for the co-simulation system we are presenting, information about which simulator was last to execute, the values of variables shared between the simulators, the current time, and the complete states of both simulators. A configuration is typically given as a tuple.

The behaviour of a system is defined through the use of transition relations, at least one of which must involve the system’s configuration type. In a fine-grained SOS definition the overall system behaviour is typically defined using a transition relation from configurations to configurations.

The transition relations are defined through the use of inference rules where the rule’s conclusion is structured to give a pair (schema) that belongs in that particular transition relation. The least relation that satisfies all of the inference rules is taken to be the defined relation.

This will be illustrated as we give the top-level definition of the co-simulation semantics in the next section.

B. Co-simulation Semantics

To a first approximation, the semantics of co-simulation in the DESTECS tool is simply the semantics of the co-simulation engine. The co-simulation engine is, itself, just a round-robin scheduler that passes synchronization data between the simulators. This paper will not include details regarding the initialization of the overall co-simulation system, instead focusing on its normal operation. An initial draft of the full semantics is available in the DESTECS project deliverable D3.3b [LRV+11], and an updated version will be released as a technical report.

Normal operation of the co-simulation system is illustrated in Figure 1. The co-simulation system first passes control and state to the DE simulator in step 1. The DE simulator then performs a step then passes control and state back to the co-simulation engine in step 2. The co-simulation engine merges that state, and then in step 3 passes control and state to the CT simulator. The CT simulator performs its step, then passes control and state back to the co-simulation engine in step 4. Finally, the co-simulation engine merges that state and the cycle repeats.

In this description of normal operation we have started by having the DE perform the first step. As it turns out this is a necessary thing: the DE simulator generates a time bound for the CT simulator. If we were to try to have the CT simulator make the first step, we would lack any bound for its run. Even were we to generate some arbitrary (but reasonable) initial bound and first perform a CT step, we are open to the possibility that there could have been state changes from the DE that would affect the CT. Hence, the DE makes the first step and the CT catches up.

The semantics of the entire co-simulation system can ultimately be given in two concise rules, plus a third for initialization (not given here). First, however, we will define the mathematical objects that describe the abstract structure of the co-simulator.

The overall co-simulation system configuration is given as

\[ \text{Config} = \text{DE} \times \text{CT} \times \Sigma_0 \times \text{Time} \times \text{Time} \times \text{Event-set} \times \text{SysTag} \]

where

- \( \text{DE} \) is the type of representations of the discrete-event simulator, covering all of the possible states that it may reach;
- \( \text{CT} \) is the type of representations of the continuous-time simulator, also covering all of the states that it may reach;
Fig. 1: Abstract representation of a DE/CT co-simulation system with indexes representing the sequencing of events.

- $\Sigma_o$ is the representation of shared variables, a mapping of variable names to values, where each value is tagged with an “owner” indicating which simulator may write to the value;
- the first $Time$ type represents the current time of the overall co-simulation system;
- the second $Time$ type represents a time bound that any CT simulator step must respect and terminate no later than;
- $Event$-set which records the events generated by the CT simulator for the DE simulator to react to; and,
- $SysTag$ which consists of two tokens representing the two simulators so that the system may record which simulator took the last step.

Events generated by the CT simulator are handled immediately by the next step of the DE simulation, and it is expected that the CT simulator stops immediately if an event is generated. In the unlikely case that multiple events are generated simultaneously there will be no ordering between them, and so a set (rather than a sequence) is used to record this.

We will cover the internal structure of the $DE$ element in Section III, and the internal structure of $CT$ is not relevant to the semantics given here. We define the type of the shared variables as

$$\Sigma_o = Id_o \rightarrow (SharedValue \times SysTag)$$

where $Id_o$ is an inexhaustible set of identifier names for variables, $SharedValue$ is the subset of possible values that may be shared across the simulators, and $SysTag$ is a symbol—either (DE) or (CT)—indicating which simulator is allowed to change the value of that variable.

With the structure defined so far we can continue on and define the main transition relation for the co-simulation engine, $\xrightarrow{cs}$.

$$\xrightarrow{cs}: Config \times Config$$

We give two inference rules to define the behaviour of the co-simulation engine; however, we will make the assumption that the system configuration is already in an appropriate state and not deal with initialization here. In Figure 2 the first rule, Co-Sim DE Step, gives the behaviour of the co-simulation engine when it is performing a step in the discrete-event simulator; the second rule, Co-Sim CT Step, handles the corresponding case when the co-simulation engine performs a step in the continuous-time simulator.

Starting with the Co-Sim DE Step rule, consider first the $Config$ tuple on the left of the $\xrightarrow{cs}$ relation. This tuple is given in terms of its components to allow us to name the components for use in the rule. Moving our focus to the first hypothesis of the rule, we see some of the components of the $Config$ tuple being used in a new transition relation, $\xrightarrow{de}$. This transition relation is defined as

$$\xrightarrow{de}: (DE \times \Sigma_o \times Time \times Event-set) \times (DE \times \Sigma_s \times Time)$$

and it represents a single simulation step of only the discrete-event simulator. We also introduce a new type

$$\Sigma_s = Id_v \rightarrow SharedValue$$

that is a simple map of variable identifiers to shared values, but without the ownership tag that $\Sigma_o$ has. Informally, the left side of the $\xrightarrow{de}$ transition is a tuple of a DE simulator state, shared variable state, an update time, and a set of events that happened between the last point at which the DE ran and the new time; the right side is a tuple of the new DE simulator state, the new values of shared variables, and a new time bound.

The second hypothesis has the effect of merging the new values of the shared variables into the overall shared state. And, finally, on the right side of the $\xrightarrow{ct}$ transition is the resulting tuple giving the system state, using elements from the hypothesis to produce the changed tuple.

So, the Co-Sim DE Step rule defines the behaviour of the co-simulation engine when executing a DE step directly in terms of the behaviour of the DE simulation step itself, with a little bit of extra mechanism to ensure that the overall shared state is updated.

The second rule in Figure 2, Co-Sim CT Step, is structured in a similar manner to the first, but uses the $\xrightarrow{ct}$ transition relation to model the behaviour of the CT simulator as it performs a single simulation step. This transition relation is defined as

$$\xrightarrow{ct}: (CT \times \Sigma_o \times Time) \times (CT \times \Sigma_s \times Time \times Event-set)$$

where, informally, the left side of the transition relation is the CT simulator state, the shared variable state, and a time bound, and the right side of the transition relation is the new CT simulator state, the modified shared variables, the time up to which the CT simulator actually ran, and a (possibly empty) set of events generated during that step.

The primary advantage of this semantic model for the co-simulation engine is that it isolates the two simulators as much as possible, allowing the semantics for each simulator to be defined independently.

III. VDM-RT SEMANTICS

A. Overview of VDM-RT Structure & Entities

We will first give an overview of the static structure of the VDM-RT semantics, then describe the behaviour of the system.
Co-Sim DE Step
\[
\begin{align*}
\sigma'_o &= \sigma_o \upharpoonright \{id \mapsto \sigma_s(id) \mid id \in \text{dom } \sigma_s \land \sigma_o(id) = (\ldots) \} \\
\text{(de, } \sigma_o, \tau, \text{ events}) &\xrightarrow{\text{de}} (\text{de'}, \sigma_s, \tau', \text{ events'})
\end{align*}
\]

Co-Sim CT Step
\[
\begin{align*}
\sigma'_o &= \sigma_o \upharpoonright \{id \mapsto \sigma_s(id) \mid id \in \text{dom } \sigma_s \land \sigma_o(id) = (\ldots) \} \\
\text{(de, } \sigma_o, \tau, \text{ events, } (\text{DE})) &\xrightarrow{\text{ct}} (\text{de'}, \sigma'_o, \tau', \text{ events'}, (\text{DE}))
\end{align*}
\]

Fig. 2: Inference rules for the behaviour of the co-simulation engine.

The entities used to describe the VDM-RT semantics form a hierarchy starting with the \textit{DE} record. The \textit{cpus} field records the CPUs in a model by mapping of identifiers to CPUs. Similarly, the \textit{busses} field represents the busses in the model, recording the connections between CPUs and a queue of messages for each connection. The \textit{sharemap} field associates the variables shared between simulators with the full path (CPU identifier, Object identifier and variable identifier) to the model’s variable. We record the current time of the model in the \textit{τ} field and, finally, the \textit{classes} field is a map from class identifier to a class in the model.

\[
\begin{align*}
\text{DE} &::= \text{cpus} : \text{Id}_c \xrightarrow{m} \text{CPU} \\
&\quad \text{busses} : \text{Id}_b \xrightarrow{m} \text{Bus} \\
&\quad \text{sharemap} : \text{Id}_o \xrightarrow{m} (\text{Id}_c \times \text{Id}_o \times \text{Id}_o) \\
&\quad \text{eventmap} : \text{Id}_e \xrightarrow{m} (\text{Id}_c \times \text{Id}_o \times \text{Id}_op) \\
&\quad \tau : \text{Time} \\
&\quad \text{classes} : \text{Id}_cl \xrightarrow{m} \text{Class}
\end{align*}
\]

The \textit{Class} entity is a simple structure, with a map of the variables to their types, and the operations and functions defined for objects of that class. There is also an \textit{initialThreadBody} that defines the type of initial behaviour that an object of that class exhibits; either a sequence of duration statements executed at instantiation, or a periodic thread definition that should be run regularly. Operations are defined as either synchronous or asynchronous within the \textit{Op} type, the definition of which is omitted here. The call of a synchronous operation forces the calling thread to wait until the called operation has finished; asynchronous operations are run in a separate thread and do not cause the caller to wait.

\[
\begin{align*}
\text{Class} &::= \text{variables} : \text{Id}_v \xrightarrow{m} \text{Type} \\
&\quad \text{ops} : \text{Id}_op \xrightarrow{m} \text{Op} \\
&\quad \text{funs} : \text{Id}_f \xrightarrow{m} \text{Fun} \\
\text{initialThreadBody} &::= \text{Duration}^* | \text{Periodic}
\end{align*}
\]

The \textit{Object} entity represents instantiated objects and contains a pointer to defining class, the state of the object’s variables, and mapping containing the \textit{threads} running in the context of the object. There is also a \textit{periodicCountdown} field which tracks the time remaining until the next instantiation of a periodic thread (if the object is defined to have a periodic thread). The value of the \textit{periodicCountdown} field is pre-calculated, including jitter, by the \textit{createPeriodicAndEventThreads()} function, given in part in Section III-B. Although we do not cover how periodic threads are handled by the \textit{createPeriodicAndEventThreads()} function, the informal idea is that the appropriate time for the next instantiation of a periodic thread is computed when the \textit{periodicCountdown} field reaches zero. This new time value is placed in the \textit{periodicCountdown} field as the present periodic thread is instantiated.

\[
\begin{align*}
\text{Object} &::= \text{class} : \text{Id}_ct \\
&\quad \text{state} : \Sigma \\
&\quad \text{threads} : \text{Id}_t \xrightarrow{m} \text{Thread} \\
&\quad \text{periodicCountdown} : [\text{Time}]
\end{align*}
\]

The \textit{Thread} entity records a thread’s status, the values of variables pending commit, and the remaining duration statements to be executed. When the value of a variable has been changed by a thread but not yet committed, the new value is kept aside in the thread’s \textit{pending} field until a certain amount of time has passed; this behaviour is described in Section III-C.

\[
\begin{align*}
\text{Thread} &::= \text{status} : \text{RUNNING} | \text{RUNNABLE} | \text{COMPLETE} \\
&\quad \text{pending} : \Sigma \\
&\quad \text{body} : \text{Duration}^*
\end{align*}
\]

The body of a \textit{Thread} is a sequence of \textit{Duration} statements; these statements are used to indicate the execution time of the block of statements contained in the \textit{Duration}. A \textit{Duration} statement is composed of a \textit{time} field that represents the estimated execution time bound, and the \textit{body} field containing the sequence of statements to be executed.

The atomicity of the outermost \textit{Durations} have the property of being all-or-nothing so long as no operations are invoked on remote CPUs. If an operation is invoked on a remote CPU then the data will be sent outside of the scope of the \textit{Duration}; this ‘leaks’ the data and destroys the all-or-nothing property of the \textit{Duration} block. Atomicity in the sense of instantaneous execution is possible by using 0 in the \textit{time} field of a \textit{Duration}.

\[
\begin{align*}
\text{Duration} &::= \text{time} : \text{Time} \\
&\quad \text{body} : \text{Stmt}^*
\end{align*}
\]

Note that a \textit{Duration} is just a type and, thus, can contain durations. The behaviour of nested \textit{Durations}, and \textit{Durations} in general, is described in III-D.

This is sufficient structure to move on to the behaviour of the DE simulator. The \textit{DE} top level rule, \textit{DE big step} in Figure 3, consists of six hypotheses. The first updates the \textit{DE} state from the shared variables \(\sigma\). The second hypothesis deals with committing pending state changes from threads and updating
the DE time. Then new periodic threads are created based on the updated time and new event threads are created followed by thread switching for threads that needs to be switched in or out. After context switching the deexec rule is recursively used to execute the body of all threads until all threads has an empty body or a duration statement is present as the head of the body. When execution through the deexec rule is completed then all shared variables are extracted and the shortest time until the next action –commit or thread creation– is calculated.

B. Event Handling

One of the phases in a step of the DE simulator involves handling the events generated by the previous CT simulator step. This appears in the DE big step rule as the de3 hypothesis, where de3 represents the state of the DE simulator after the new threads for both new events and periodic threads in an object have been created.

createPeriodicAndEventThreads: DE × Event-set → DE
createPeriodicAndEventThreads(de, events)de′ ==
post
∀e ∈ events.
∃id ∈ Id_e, id_o ∈ Id_o, id_op ∈ Id_op, id_id ∈ Id_id.
(id_e, id_o, id_op) = de.eventmap(e) ∧
id_id = de.cpus(id_e).objects(id_o).class ∧
∃id ∈ Id_e.
de′.cpus(id_e).objects(id_o).threads(id_e) = mk-Thread(RUNNABLE, {}),
dc.classes(id_id).ops(id_op).body

Fig. 4: Definition of createPeriodicAndEventThreads()

In Figure 4 we define this function in terms of two other functions: createPeriodicThreads(), which handles the periodic creation of threads in objects that use the periodic feature; and createEventThreads(), which deals only with creating new threads to handle events. We will not cover createPeriodicThreads() in this paper.

Events are just identifiers, i.e. Event = Id_e

and are implemented in the tool as strings. There are no parameters attached to the occurrence of an event, just the identity of the event that happened. This allows for a straightforward mapping of events to their handling operations in statically declared objects.

A new thread is created for each event, with the object and operation determined by the eventmap field in the DE record. The new thread is created with RUNNABLE status, but it does not become active until it is selected by a context switch. Note that event threads do not have priority over other threads; once created they are indistinguishable from a thread created for any other reason. The present semantics definition does not include thread priorities, though this will be addressed in future work.

The new thread will run in a state where all pending commits up to the time immediately prior to the event have completed, and changes made by the event will not be visible to other threads until its pending commits complete. This means that any other threads that start before the event handling thread completes may not necessarily see the outcome of handling that event.

This behaviour for event handling is given in the createEventThreads() function in the semantics, shown in Figure 5. In the post condition of the function we require that, for every event, there is exactly one thread that is ready to run and contains the body of the operation defined as the event handler.

C. Committing Pending Values

During execution, the way threads change the values of variables in objects is modelled using the deexec transition. Such changes are not committed to the object state immediately; instead they are cached in the pending field of the Thread, hiding the values from other threads, until time progresses sufficiently to fill the time required by the active duration of the Thread.

The resolution of pending values and durations are handled in the DE big step rule by the de2 = commitPendingValuesAndUpdateTime(de′, τ)
hypothesis together with update of the DE time from the CT simulator. The de2 object represents the state of the DE simulator after the simulator time is set to τ and all pending values are committed for threads currently active –i.e. with RUNNING status– and all active duration blocks in every RUNNING thread body have their time value decremented by the amount of time passed since last time update.

The behaviour of value commit and time update is encapsulated in the semantic function commitPendingValuesAndUpdateTime(), shown in Figure 6. In the post condition of this function we require that the DE time is updated to the new time, τ, and every thread that was running has its duration decremented by the difference between the previous time and the new time, i.e. τ-DE.τ.

D. Dealing with Durations and Context Switching

The execution cycle of the VDM-RT semantics is centred around Duration statements that are used to indicate execution
commitPendingValuesAndUpdateTime:  
\[ DE \times Time \rightarrow DE \]
commitPendingValuesAndUpdateTime\((de, \tau)de' = \]
pre \( \tau \geq de.\tau \)
\( \forall \text{id}_o \in \text{dom } de.\text{cpus}(\text{id}_o)\text{.objects} \.)
\( \forall \text{id}_t \in \text{dom } de.\text{cpus}(\text{id}_o)\text{.objects}(\text{id}_o)\text{.threads} \.)
\( \exists \text{thr}, \text{thr}' \in \text{Thread}, \text{dt} \in \text{Time} \)
\( \text{thr} = \text{de}.'\text{cpus}(\text{id}_o)\text{.objects}(\text{id}_o)\text{.threads}(\text{id}_t) \land \)
\( \text{thr}' = \text{de}.'\text{cpus}(\text{id}_o)\text{.objects}(\text{id}_o)\text{.threads}(\text{id}_t) \land \)
\( (\text{thr}.'\text{body} = [\text{t}] \Rightarrow \text{thr}'\text{.status} = \text{COMPLETED}) \land \)
\( (\text{thr}.'\text{status} = \text{RUNNING} \Rightarrow \)
\( \exists \text{dt}, \text{step} \in \text{Time} \).
\( \text{step} = \tau - de.\tau \land \text{dt} = (\text{hd thr}.'\text{body})\text{.time} \land \)
\( (\text{dt} = \text{step} \Rightarrow \text{thr}'\text{.pending} = [\text{t}] \land \)
\( \text{de}.'\text{cpus}(\text{id}_o)\text{.objects}(\text{id}_o)\text{.state} = \)
\( \text{de}.'\text{cpus}(\text{id}_o)\text{.objects}(\text{id}_o)\text{.state}+\text{thr}.'\text{.pending} \land \)
\( (\text{dt} = \text{step} \Rightarrow \)
\( (\text{hd thr}.'\text{body})\text{.time} = \text{dt} - \text{step}) \))

Fig. 6: Definition of commitPendingValuesAndUpdateTime() times for blocks of expressions. Any changes made to the containing object’s state are hidden until the time value of the Duration reaches zero, at which point the changes become visible. These Duration statements, when incomplete, also have the effect of blocking other threads from executing on that CPU.

It is important to remember that the time value in a Duration statement represents information from the user about the temporal characteristics of the eventual implementation — Duration time values are not intended to be calculated in any but the most trivial way; i.e. reduced in value as time passes in a simulation. So, the time value of a duration can be interpreted as a strict deadline on the execution of the contained statements. This constraint can always be achieved as the semantics ignores nested Duration statements.

The semantics deals with execution the statements contained in duration blocks in the DE big step rule. The time-related function of duration blocks is handled in the commitPendingValuesAndUpdateTime() function, where the time of any active durations on a given CPU is decreased, and for duration blocks that have reached their bound, the pending values are committed to the object state. A duration completes its execution when the time field is decremented to 0 and at that point enables a context switch where the active thread of a CPU may be swapped to another thread. A context switch is handled in the DE big step rule by the hypothesis

\[ de_4 = \text{doContextSwitches}(de_3) \]

where \( de_4 \) calculates the DE state after a context switch. A context switch allows any thread of a CPU to be swapped out if it previously completed a duration statement and allows any thread to be swapped in if the thread state is RUNNABLE. Of course, given that description, doContextSwitches() is specified in a non-deterministic manner.

The behaviour for context switching is given in the doContextSwitches() function in the semantics, as shown in Figure 7. The post condition of the function must be read in four parts — each part is enclosed in square brackets for easy identification— that jointly constrain the behaviour to what we require. The first part of the post condition simply ensures that all threads that were RUNNING and had finished the Duration in its body had deleted that Duration in the post-state. The second part ensures that all threads with empty bodies have the COMPLETED status. The third part guarantees that if there are any threads that are in a RUNNABLE status then this function will make sure that there is exactly one thread with RUNNING status. Finally, the fourth part checks the relationship between the two RUNNING threads chosen before and after this function to make sure that: a) if the RUNNING thread before this function is in the middle of executing a duration then this function did not swap it out, i.e. the same thread is still RUNNING; and b) if a different thread is RUNNING after the context switch then the time value in its Duration is incremented by the amount of time it takes to perform a context switch, i.e. switchDelta.

IV. PROPOSED OPTIMIZATIONS

The semantics presented here was developed using the behaviour of the current implementation of the DESTecs tool as a primary guide; this was done with the aim of using this semantics to justify that performance optimizations to the tool are sound and correct. Here we present one possible optimization to the implementation which will present the same observable behaviour to the tool user. The conservative approach involves synchronizing the shared state between the DE and CT simulators at every point where the DE needs to commit a pending value. The
implementation will do this regardless of whether or not those pending values affect the CT simulator in any way, and regardless of whether or not the computation following the commit depends on any values provided by the CT simulator.

The implementation presently runs the DE model to the point where every thread has either finished or reached a \textit{Duration} block. A time bound is then calculated on that basis, and the CT is triggered to run the least amount of time until either that bound is reached or an event happens (or both). We propose to do a simple analysis of the DE model to determine what the largest time bound is until the DE either commits a value that affects the CT, or the DE needs a value which may be provided by the CT. Ensuring that we do not have dependencies between the DE and CT within this time bound will allow the DE to take larger steps than it presently does.

In terms of the semantics, a naïve formalization of this only involves a change to the \textit{extractValuesAndMinDuration()} function so that we omit from consideration those pending commits where the a) the pending values do not affect the CT simulator; and b) the computation following the commit on that thread does not depend on values from the CT simulator.

V. CONCLUSIONS AND FUTURE WORK

We have presented a semantics for a co-simulation engine, in particular, the one used in the DESTECS tool. As written, the semantics allows for the possibility of using any DE or CT simulator that meets the required interface. With some extension we will also be able to add a rule to the co-simulation engine semantics that allows for testing functionality such as fault-injection.

The VDM-RT semantics, as described, deals with nested durations such that only the outermost duration is considered — this is consistent with the original intent of durations as a means of allowing the modeller to estimate the time an operation will take to execute. However, we have not considered here how durations should be handled when the body of a duration includes a synchronous call to an operation in an object on a different CPU. The key problem here is that the execution of the remote operation is subject only to constraints on the remote CPU: there is no guarantee that the caller’s duration timing will be respected. In the worst case execution of the remote operation may be delayed indefinitely as it is possible, in certain circumstances, for the thread to never be scheduled.

Though still incomplete, our efforts thus far with the VDM-RT semantics have highlighted areas where it is unclear what the behaviour should be for various language elements. The process of identifying what does happen in the implementation in these cases and determining what we need to change is enlightening. As a result of writing the semantics, we have already identified one area where we will improve the performance of the tool implementation.
Acknowledgements: The authors gratefully acknowledge funding from the European Commission through the FP7 DESTECS project (grant agreement number 248134). We would also like to thank Marcel Verhoef for feedback on draft versions of the paper.

REFERENCES


Shigeru KUSAKABE, Yoichi OMORI, and Keijiro ARAKI
Graduate School of Information Science and Electrical Engineering, Kyushu University,
744 Motooka Fukuoka, 819-0395, Japan
kusakabe@ait.kyushu-u.ac.jp

Abstract—We aim to propose effective and usable formal methods to cover the whole software lifecycle including operation and maintenance phases through architecture oriented formal approach. In an architecture-oriented approach, communicating an architecture among stakeholders is very important. Our perspective is that the family of Overture/VDM will be effective in documenting and communicating architecture, covering a wider range of software life-cycle than other formal methods. We expect stakeholders including designers and implementers of detailed components and their interfaces can efficiently understand the architecture if we use Overture/VDM. While we have many aspects to be considered, we report the results of our initial trials in evaluating the effectiveness of Overture/VDM in a development process.

I. INTRODUCTION

A growing number of software engineers seem to have interests in formal methods as a wider range of software supports important social infrastructure and problems in the software may heavily impact on our society. Formal methods for software and system development are regarded as promising approaches to efficiently and effectively realize reliable and dependable IT systems. However, in the actual development project, formal methods are introduced in very limited cases even though we have reports of succeeded projects[1] and engineers and managers are interested in formal methods. For example in Japan, several organizations such as Software Engineering Center (SEC) of Information-technology Promotion Agency (IPA) are trying to establish guidelines to introduce formal methods in real projects.

We aim to propose effective and usable formal methods to cover the whole software lifecycle including operation and maintenance phases. We employ a software architecture-oriented approach as software architecture discipline is expected to play an important role in reducing complexity in developing and managing large scale software through abstraction and separation of concerns. While we do not have "the" precise definition of software architecture, we model a complex system by considering comprehensively its development and maintenance processes, and its environment as well as operations for the system.

Keijiro Araki’s proposal based on this approach has been accepted as a five-year project of Scientific Research (S) of Grant-in-Aid for Scientific Research in Japan, with 122.2 million Yen budget and seven researchers. Fig. 1 shows the outline of the modeling view in this proposal. We model a complex system as an ensemble of components and interactions between them. We consider its development and maintenance processes, and its environment as well as operations for the system. We aim to clarify which formal method is effective in which process from various points of view during the entire lifecycle of software system based on this modeling view.

The key research topics of the project include the following:

1) Proposal of effective formal techniques applicable to model and analyze complicated IT systems and case studies of their applications,
2) Reference models of software development processes based on formal methods,
3) Architecture oriented formal approaches to treat complicated systems of systems including environment and operation phases, and
4) Development of support tools.

In this paper, we report an initial result of a trial mainly related to the second topic of this project, software development process with formal methods.

A book of software architecture[2] explains three reasons why software architecture is important as follows.

1) Architecture is the vehicle for stakeholder communication. Software architecture represents a common abstraction of a system that stakeholders can use as a basis for mutual understanding and communication.
2) Architecture manifests the earliest set of design decisions. These early decisions are the most difficult to get correct and the hardest to change later in the development process.
3) Architecture can be a transference and reusable model, abstraction of a system. While code reuse is beneficial, reuse at the architecture level provides tremendous leverage for systems with similar requirements.

Our perspective is that the family of Overture/VDM[3][4] will be effective in abstracting documenting and communicating architecture. Although there exist architecture description languages, such as SysML, architecture description languages based on UML have less formal than we expect. Among formal methods with description languages, we focus on
Overture/VDM in this study from the viewpoint of software process as Overture/VDM can cover a wider range of life-cycle than other formal methods. We expect stakeholders including designers and implementers of detailed components and interfaces can easily understand the architecture and rigorously described documents can facilitate process activities if we use Overture/VDM. While we have many aspects to be considered, we report the results of our initial trials in evaluating the effectiveness of Overture/VDM in a development process.

II. PROCESS WITH FORMAL MEEHODS IN ARCHITECTURE-ORIENTED APPROACH

Although we do not have "the" precise definition of software architecture, we expect formal specification languages play an important role in seamlessly associating the earliest design decisions with documents and activities in an architecture-oriented approach. According to the description in web pages of Software Engineering Institute (SEI) of Carnegie Mellon University (CMU),

"The software architecture of a program or computing system is a depiction of the system that aids in the understanding of how the system will behave. Software architecture serves as the blueprint for both the system and the project developing it, defining the work assignments that must be carried out by design and implementation teams."

If we use formal methods in early design decision of software architecture, we will have formally described blueprint, which will facilitate project activities including detailed design and implementation. One of the issues in an architecture oriented approach is cost-performance trade-off. We examine the benefit of using documents in formal languages by evaluating a software development process extended with formal methods.

A. PSP: Personal Software Process

We employ TSP (Team Software Process) and PSP (Personal Software Process)[5][6][7], developed and managed by CMU SEI as our baseline process frameworks, mainly because these are widely known as scalable practice providing measurable and customizable processes[8]. We evaluate the impact of formal methods on development a process according to the process data, and we customize the process depending on the relationships between formal methods and the early design decision in an architecture-oriented approach. We expect formal methods are helpful in refining the early design decision of the architecture to team level processes, and then personal level processes. We briefly introduce PSP without going into the details of PSP here, as the details of PSP can be found in literatures[7]. PSP has different levels of process practice as PSP can be used in education of process improvement as well as in actual projects. Based on a standard course of PSP, we used PSP0 focused on measuring practice, PSP1 estimate, and PSP2 quality. PSP0 and PSP1 consist of the following phases: planning, detailed design, coding, compile, test, and postmortem, while PSP2 extra two review phases: planning, detailed design, design review, coding, code review, compile, test, and postmortem.

B. Process extension with VDM++

In the first part of this trial, a graduate student developed four programs for PSP0 and PSP1 according to a standard course guideline, and we use process data for these programs as the baseline. He used UML in the detailed design phase in this baseline process. Then he introduced VDM++ in the detailed design phase and developed two programs for PSP2. In introducing VDM++ into PSP, we employ a light-weight approach, which corresponds to so-called level 0 approach.
There is now more widespread belief that it is best to use formal methods as needed, and there exist different formalization levels\cite{9}. The levels are level 0 - formal specification, level 1 - formal development/verification, and level 2 - machine-checked proofs. The activities are using formal notation to specify requirements only without analysis or proof in level 0, proving properties and applying refinement calculus in level 1, and using a theorem prover or checker to prove consistency and integrity in level 2, respectively. We expected level 0 is most cost-effective, and the student introduced based on a level 0 approach. We analyzed the process data of the baseline process based on the defect type defined in PSP.

- Documentation - Comments, messages
- Syntax - Spelling, punctuation, typos, instruction formats
- Build, Package - Change management, library, version control
- Assignment - Declaration, duplicate names, scope, limits
- Interface - Procedure calls and references, I/O, user formats
- Checking - Error messages, inadequate checks
- Data - Structure, content
- Function - Logic, pointers, loops, recursion, computation, function defects
- System - Configuration, timing, memory
- Environment - Design, compile, test, or other support system problems

According to the process data for this case, we focus on the following defect types which were frequently injected and expensive to fix. The figures in parenthesis are average fix time of defects for the type in minutes.

- Interface type defects
  - I-1 Defects due to insufficient refinement (15.8)
- Function type defects
  - F-1 Defects on looping control (10.3)
  - F-2 Defects in implementation logic (6.8)

In order to prevent injection of defects of the types described above, we extended the baseline process to include the following steps.

Step 1: Writing signature of methods in VDM++ in detailed design phase and using VDMTools for syntax and type check in order to prevent injection of defects of I-1 type.
Step 2: Describing sequence handling part in VDM++ in order to prevent injection of defects of F-1 type.
Step 3: Writing explicit VDM++ specifications for selected parts and using animation of VDMTools.

Fig. 2 shows the comparison of time ratio spent in each phase. As we can see from the figure, the developer spent more in design and design review, and spent less in coding and testing. In the baseline process, he spent 11.1% in design, 42.6% in coding, and 17.3% in testing, while he spent 38.9% in design plus design review, 23% in coding plus code review, and 14.3% in testing in the proposed process. By introducing formal methods like Overture/VDM, we expect reduction of undesirable behaviors like designing in coding, and consequently reduction of time spent in testing. In this case, the range of the program size in LOC, line of code, about more than hundred to less than three hundreds, and we do not see remarkable increase or decrease of productivity in comparing process data for the baseline and proposed processes.

Fig.3 shows the comparison of defect density, the number of defects per one thousand lines of code, between the baseline process and the extended process with VDM++. As we can see from the figure, there seems no difference in total defect density between the baseline and the extended process. The defects of language type were injected mainly because the student was unfamiliar with the programming language used in this trial. We can expect this kind of defects will be reduced if the developer get familiar with the programming language. Please note we could reduce defects of interface type, one of the focused defect types. We conclude rigorously described interfaces in formal languages such as VDM++ is effective in reducing defects, and we expect describing interfaces rigorously in an early stage is facilitated in an architecture-oriented formal approach.
III. CONCLUDING REMARKS

We reported our initial evaluation of the effect of formal methods on a personal process, expecting that we can easily produce detailed design in a formal specification language in an architecture oriented approach using formal methods. In this trial, the developer wrote the formal specification from the requirement document in a natural language. In an ideal setting, documents are already written in a formal notation before starting a development process of personal level. Some part of the design time will be shifted into more upfront activities. Rigorously described interfaces in formal languages such as VDM++ is effective in reducing defects, and we expect describing interfaces rigorously in an early stage is facilitated in an architecture-oriented formal approach. As this is a report of our initial trial, we will continue to work on this kind of effort and other topics in our architecture oriented approach.

ACKNOWLEDGMENT

This work was partly supported by MEXT/JSPS KAKENHI in Japan, Grant-in-Aid for Scientific Research (S), Grant Number 24220001, "Architecture Oriented Formal Approaches to High Quality Software Development”.

REFERENCES