THE 14TH OVERTURE WORKSHOP: **TOWARDS ANALYTICAL TOOL CHAINS**

Electrical and Computer Engineering
Technical Report ECE-TR-28

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

DATA SHEET

Title: The 14th Overture Workshop: Towards Analytical Tool Chains
Subtitle: Electrical and Computer Engineering
Series title and no.: Technical report ECE-TR-28

Authors: Peter Gorm Larsen, Nico Plat and Nick Battle (Eds)
Department of Engineering – Electrical and Computer Engineering, Aarhus University

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

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

Year of publication: 2016 Pages: 136
Editing completed: November 2016

Abstract: This report contains the proceedings from the 14th Overture workshop organized in connection with the Formal Methods 2016 symposium. This includes nine papers describing different technological progress in relation to the Overture/VDM tool support and its connection with other tools such as Crescendo, Symphony, INTO-CPS, TASTE and ViennaTalk.

Keywords: electronics, power electronics, optics and photonics, biomedical devices and applications, communication systems, digital signal processing, embedded systems, software engineering and systems

Please cite as: Peter Gorm Larsen, Nico Plat and Nick Battle (Eds), 2016. The 14th Overture Workshop. Department of Engineering, Aarhus University. Denmark. 136 pp. - Technical report ECE-TR-28

Cover layout: Peter Gorm Larsen

ISSN: 2245-2087

THE 14TH OVERTURE WORKSHOP: TOWARDS ANALYTICAL TOOL CHAINS

Peter Gorm Larsen, Nico Plat and Nick Battle (Eds)
Aarhus University, Department of Engineering

Abstract

This report contains the proceedings from the 14th Overture workshop organized in connection with the Formal Methods 2016 symposium. This includes nine papers describing different technological progress in relation to the Overture/VDM tool support and its connection with other tools such as Crescendo, Symphony, INTO-CPS, TASTE and ViennaTalk.

Preface to the 14th Overture Workshop

Nick Battle, Peter Gorm Larsen and Nico Plat

The 14th in the "Overture" series of workshops on the Vienna Development Method (VDM), its associated tools and applications, was held in association with the FM2016 conference in Limassol, Cyprus on the 7th of November 2016. The workshop aimed to identify and encourage new collaborative research, and to foster current strands of work towards new projects and publications.

Although VDM is one of the oldest formal methods to have enjoyed a level of industry use, it nevertheless has a lively and youthful research community, which has grown up around the development of the Overture open tools platform. The Crescendo and Symphony tools have been built on top of Overture by the DESTECS and COMPASS projects, respectively, and the tools are being further extended by the INTO-CPS project as well as by the TEMPO experiment, supported by the CPSE-Labs project. The platform provides a vehicle for activity in modelling and analysis technology, including static analysis, interpreters, test generation, execution support and model checking. The growth of this community has been greatly assisted by the Overture workshop series.

Research in VDM and Overture is driven by the need to develop industry practice as well as fundamental theory. Consequently, the provision of tool support has been a priority for many years. The community-based Overture initiative is developing industry-strength tools on an open platform that has been successfully adapted to form platforms for co-modelling and co-simulation in embedded systems design, and latterly SoS modelling, verification and testing.

The 14th workshop reflected the breadth and depth of work in VDM. Contributions covered topics as diverse as cyber-physical systems, automated code generation, and tool chain integration. In this technical report we include three technical sessions with papers. The theme of the first session is Cyber-physical Systems (CPSs) which start out by considering the resilience of Model-Based Design of CPSs [4]. This is followed by a paper examining different abstraction levels in the development of a UAV control system [8]. The theme of the second session considers different kinds of tool enhancements. Here the first paper deals with the automatic generation of C# and .NET code contracts from models written in VDM-SL [2]. This is followed by a paper looking into code generation for Smalltalk [6]. Afterwards, we have a paper about a tool chain for the INTO-CPS project including Overture [5]. Finally, this session is completed with a paper looking at test automation in a code generation setting [9]. The theme of the last session is Using and Extending Overture in different ways. This is started by a paper about integrating Overture into the TASTE tool chain [3]. Afterwards we have a paper using Overture to model collaboration between traffic management systems [7]. Finally, the last paper demonstrates how Overture has been extended with an ability to monitor variables at run-time in an implicit fashion [1]. The slides of the paper presentations can be found on the Overture Wiki.

References

1. Couto, L.D., Lausdahl, K., Plat, N., Larsen, P.G., Pierce, K.: Decoupling validation UIs using Publish-Subscribe binding of instance variables in Overture. In: Larsen, P.G., Plat, N., Battle, N. (eds.) The 14th Overture Workshop: Towards Analytical Tool Chains. pp. 123–136. Aarhus University, Department of Engineering, Cyprus, Greece (November 2016), ECE-TR-28
2. Diswal, S.P., Tran-Jørgensen, P.W., Larsen, P.G.: Automated Generation of C# and .NET Code Contracts from VDM-SL Models. In: Larsen, P.G., Plat, N., Battle, N. (eds.) The 14th Overture Workshop: Towards Analytical Tool Chains. pp. 32–47. Aarhus University, Department of Engineering, Cyprus (November 2016), ECE-TR-28
3. Fabbri, T., Verhoef, M., Bandur, V., Perrotin, M., Tsiodras, T., Larsen, P.G.: Towards integration of Overture into TASTE. In: Larsen, P.G., Plat, N., Battle, N. (eds.) The 14th Overture Workshop: Towards Analytical Tool Chains. pp. 94–107. Aarhus University, Department of Engineering, Cyprus, Greece (November 2016), ECE-TR-28
4. Jackson, M., Fitzgerald, J.: Resilience Profiling in the Model-Based Design of Cyber-Physical Systems. In: Larsen, P.G., Plat, N., Battle, N. (eds.) The 14th Overture Workshop: Towards Analytical Tool Chains. pp. 1–15. Aarhus University, Department of Engineering, Cyprus, Greece (November 2016), ECE-TR-28
5. Larsen, P.G., Thule, C., Lausdahl, K., Bardur, V., Gamble, C., Brosse, E., Sadovykh, A., Bagnato, A., Couto, L.D.: Integrated Tool Chain for Model-Based Design of Cyber-Physical Systems. In: Larsen, P.G., Plat, N., Battle, N. (eds.) The 14th Overture Workshop: Towards Analytical Tool Chains. pp. 63–78. Aarhus University, Department of Engineering, Cyprus (November 2016), ECE-TR-28
6. Oda, T., Araki, K., Larsen, P.G.: Automated VDM-SL to Smalltalk Code Generators for Exploratory Modeling. In: Larsen, P.G., Plat, N., Battle, N. (eds.) The 14th Overture Workshop: Towards Analytical Tool Chains. pp. 48–62. Aarhus University, Department of Engineering, Aarhus University, Department of Engineering, Cyprus (November 2016), ECE-TR-28
7. Plat, N., Larsen, P.G., Pierce, K.: Modelling Collaborative Systems and Automated Negotiations. In: Larsen, P.G., Plat, N., Battle, N. (eds.) The 14th Overture Workshop: Towards Analytical Tool Chains. pp. 108–122. Aarhus University, Department of Engineering, Cyprus, Greece (November 2016), ECE-TR-28
8. Thule, C., Nilsson, R.: Considering Abstraction Levels on a Case Study. In: Larsen, P.G., Plat, N., Battle, N. (eds.) The 14th Overture Workshop: Towards Analytical Tool Chains. pp. 16–31. Aarhus University, Department of Engineering, Cyprus, Greece (November 2016), ECE-TR-28
9. Tran-Jørgensen, P.W., Larsen, P.G., Battle, N.: Using JML-based Code Generation to Enhance the Test Automation for VDM Models. In: Larsen, P.G., Plat, N., Battle, N. (eds.) The 14th Overture Workshop: Towards Analytical Tool Chains. pp. 79–93. Aarhus University, Department of Engineering, Cyprus (November 2016), ECE-TR-28

Table of Contents

Resilience Profiling in the Model-Based Design of Cyber-Physical Systems

Mark Jackson and John S. Fitzgerald

Newcastle University, UK
{M.Jackson3, John.Fitzgerald}@newcastle.ac.uk

Abstract. We consider the potential to use co-modelling and co-simulation in the design of dependably resilient Cyber-Physical Systems (CPSs). Resilience is widely discussed in the public discourse on CPSs, but has many definitions. We propose the description of system resilience in terms of a multi-attribute profile which may be used as a basis for assessment and trade-off analysis in CPSs. Our profile has a particular focus on description of system recovery behaviour. As a first evaluation of the concept, we present a case study based on a VDM and 20-sim co-model of a small smart grid illustrating causal chains that cross the cyber-physical boundary. An evaluation of the study leads to suggestions for further proof-of-concept studies that experiment with increasingly challenging CPS architectures.

1 Introduction

> In complex environments, resilience often spells success, while even the most brilliantly engineered fixed solutions are often insufficient or counterproductive.
>
> *McChrystal et al. [1]*

Cyber-Physical Systems (CPSs) formed from the integration of computational and physical processes [2] are a natural evolution of embedded devices in networked environments [3]. Examples range from medical devices and automotive control systems, to "smart" infrastructures in areas such as road traffic management and energy grids [4,5]. In many cases, existing physical infrastructure is overlaid with a computer network that – in principle – offers more efficient and reactive control with greater autonomy than is delivered by individual embedded or centralised architectures. However, computer networks are complex (in the sense that small changes can have remote and large-scale effects), and are vulnerable to failure and attack modes that can be difficult to predict and test. Adding cyber networks to physical infrastructure is therefore a risky business. Controlling this risk requires both design methods that promote detection and avoidance of vulnerabilities, and building-in the capacity to recover from unanticipated faults or attacks.

Much of the public discourse on infrastructure is concerned with *resilience*. Although widely used, the term appears to have a range of meanings in different sectors. The United Kingdom's National Resilience Capabilities programme sees it as "the

ability of assets and networks to anticipate, absorb, adapt to and recover from disruption" [6]. It is open to debate whether this might be defined in terms of properties such as fault avoidance, detection, tolerance and recovery [7], but the term is often used in a broader sense to include adaptive capacity [8]. Given the range of facets of resilience that are important in different application sectors, it is apparent that a nuanced characterisation of resilience is needed to facilitate disciplined engineering.

Engineering dependably resilient CPSs is a demanding goal [9], and many CPSs emerge or are developed without resilience in mind at all. Model-based systems engineering methods offer considerable promise, but are challenged by the independence and heterogeneity of CPS constituents. Several of these challenges are addressed by co-modelling technology such as that explored in INTO-CPS[1]. Predecessor projects such as DESTECS[2] and COMPASS[3] demonstrated model-based engineering for fault tolerance in these settings. It is therefore legitimate to ask whether co-modelling can help to deliver dependably resilient CPSs. This is the subject of our current work.

We discuss background and related work on resilience in Section 2. Given the need for a nuanced characterisation of resilience, we discuss *resilience profiling* (Section 3) and consider how this might be realised in heterogeneous co-models of CPSs. An example based on a simple smart grid is presented (Section 4). Our work is at an early stage; Section 5 describes future directions.

2 Background and Related Work

We aim to provide usable methods and tools for engineering dependably resilient CPSs. In this section, we describe the scope and background to our work. We briefly indicate what we mean by a CPS, explain why we focus on model-based techniques and introduce our baseline tools. We then examine in more detail the existing work on resilience profiling in CPS-related contexts.

Multidisciplinary Model-Based Design for CPSs

A CPS integrates computational and physical processes [2]. CPS engineering should therefore address the integration of methods and tools from different (discrete and continuous) domains and disciplines [10]. The focus of much current work, including our own, is on systems of networked computing elements, including "smart" devices, that together deliver emergent properties on which reliance is placed. This adds to the mix important systems-of-systems (SoS) aspects, including the need to integrate independently owned and managed systems, the ability to reason about the composition of the contractual interfaces between them, and the ability to deal with dynamically evolving structures over the life of the CPS [11, 12].

Collaborative and multi-paradigm Model-Based Design (MBD) techniques have been proposed as a means of evaluating alternative architectures and functionality, and providing early identification of defects in CPSs [13]. Realising the value of such approaches requires a semantic basis for linking models given in diverse notations, the

[1] http://into-cps.au.dk/
[2] http://www.destecs.org/
[3] http://www.compass-research.eu/

ability to compose abstract descriptions of interfaces between system elements, and the ability to describe architectures explicitly.

Much research builds on hybrid systems as a common semantic framework for CPSs [14]. Rather than work with a single formalism, we aim at an extensible framework able to integrate the diverse formalisms used in practice. In the work discussed here, we take Crescendo[4] as a baseline technology. In Crescendo, a model of a CPS is actually a *co-model* with discrete-event (DE) and continuous time (CT) models (in VDM/Overture and 20-sim respectively) as its constituents. Crucially for our work, the approach allows the direct modelling of causal chains across the cyber-physical boundary. A bespoke co-simulation harness implements an operational semantics that manages time and communication between the separate DE and CT models running in their own simulators. The emerging INTO-CPS tool chain promises to extend this to an n-ary multi-modelling approach, allowing co-simulation of executables derived from multiple modelling tools [15]. It leverages Unifying Theories of Programming (UTP) to permit extensible and reusable semantics [16]. At the time of writing, the INTO-CPS framework is not quite able to handle multi-models of the type needed for our smart grid applications, and we remain with Crescendo for the moment.

Resilience

Resilience is important in many fields [17, 18]. In materials science it is "the ability of a material to absorb energy when deformed elastically and to return it when unloaded" [19]. In IT and organisational contexts, a resilient control system has been characterised as "one that maintains state awareness and an accepted level of operational normalcy in response to disturbances, including threats of an unexpected and malicious nature" [20]. In the context of power systems, it is seen as the ability of a system to degrade gracefully under extreme perturbations, and recover quickly after the events have ceased [21]. In socioecological systems Carpenter et al. argue that an assessment of system resilience must be qualified by specifying which system configuration and disturbances are of interest (resilience 'of what, to what, and under what conditions') [22]. Recent European research calls on crisis management see resilience as the ability to reduce the impact of disruptive events and the recovery time [23].

Together, the approaches in the literature reflect the idea that resilience as a composite property: a system cannot simply be said to either be resilient or not, but may be said to show some characteristics of resilience in response to a certain set of faults or attacks under certain circumstances. There is a also a trade space here: for example, a system may show resilience to a certain set of attacks, but at the expense of becoming less resilient to others, or at the price of slower recovery. Again, being able to trace cause and effect as they go across the cyber-physical boundary is critical to effective model-based engineering of resilience.

Among the few current research projects directly addressing CPS resilience are ADREAM[5], FORCES[6], and SURE[7]. ADREAM aims to investigate and develop core

[4] `http://crescendotool.org/`
[5] `https://www.laas.fr/public/en/adream-project`
[6] `https://www.cps-forces.org/`
[7] `http://cps-vo.org/group/sos/sure/`

technologies, methodologies and components that will enable the successful design of dependable CPSs. FORCES aims to increase the resilience of large-scale networked CPSs in the key areas of energy delivery, transportation, and energy management in buildings. SURE will develop foundations and tools for designing, building, and assuring CPSs that can maintain essential system properties in the presence of adversaries. Although few outputs are available from these projects at this time, there is an emerging body of work aiming to address CPS resilience.

Resilience Profiling

To analyse resilience in model-based CPS design, we need a working intuitive characterisation of resilience. We adopt some of the terminology of faults, errors and failures [7] in that we regard a failure as the deviation of a delivered service from correct service. An error occurs when the state of the system deviates from those required to deliver a correct service. A fault is the adjudged or hypothesized cause of an error. An error does not necessarily cause a failure, but it is possible one or more errors may. Throughout this report we describe a specific fault–error–failure casual chain as a *resilience scenario*.

Rather than identifying a single resilience metric, we treat it as a multi-attribute property defined in what we will call a *resilience profile*. The idea of a multifaceted definition of resilience is not new. Jackson [24] proposes a representation of resilience composed of four attributes:

Capacity: the ability of a system to absorb or adapt to a disturbance without a total loss of performance or structure.
Tolerance: the exhibition of graceful degradation near the boundary of a system's performance.
Flexibility: the systems ability to restructure itself in response to disruptions.
Inter-element Collaboration: collaborations, or communication and cooperation between human elements of a system.

A resilience scenario is divided into three aspects:

Avoidance: the preventive aspects of system resilience in response to a disruption, either internal or external.
Survival: implies that the system has not been destroyed or totally incapacitated and continues to function when experiencing a disturbance.
Recovery: the capability of surviving a major disturbance with reduced performance. This capability is a focus of system resilience.

Pflanz [25] extends Jackson's characterisation, applying it to command and control systems. Although Pflanz does not consider inter-element collaboration, he subdivides the first three of Jackson's attributes into the constituent facets listed in Figure 1. Jackson's resilience scenarios were then implemented by Pflanz as temporal phases, as shown in Figure 2.

Pflanz's work focuses on the survival phase, where capacity, tolerance and flexibility provide means of analysing resilience in this phase alone. However, there is only limited further discussion of ways in which to characterise recovery.

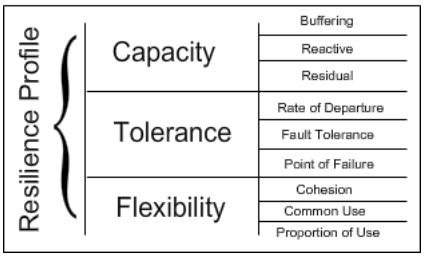


Fig. 1. Outline of Pflanz's Resilience Profile.

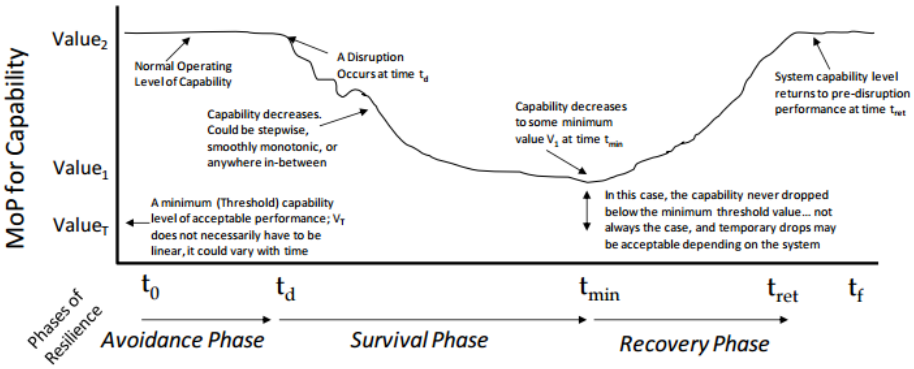


Fig. 2. Avoidance, Survival and Recovery as temporal phases, from [25].

Research Landscape

Although increasing importance is attached to the resilience of the CPSs on which we depend, there is no widely accepted definition of the concept, still less methods and tools for the lack of a coherent definition of resilience in the field, and a lack of methods for analysing resilience as a profile, especially in the recovery phase. In our current work, we therefore have two main goals. First, to deliver a resilience profile that describes resilience in the context of CPS, specifically characterising the recovery phase. Secondly we will provide methods for analysing co-models in the MBD of CPS.

3 Resilience Profiling

In this section we describe our approach to delivering the two goals mentioned at the end of Section 2. We extend the resilience profile described with new attributes which elaborate the **recovery phase**. We initially present one new attribute 'Recovery' with three facets described below and indicated in Figure 3:

Rate of Recovery: the rate at which system performance returns to an acceptable level.

Available Recovery Capacity: the available performance margin from the current operating levels to the expected recovery operating level.

Actual Recovery Capacity: the actual performance margin from the current operating levels to the recovery operating levels.

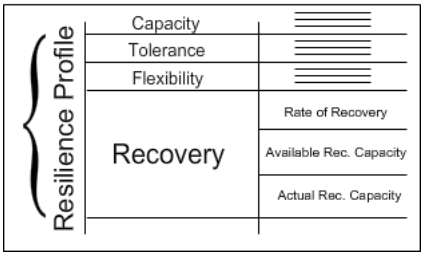


Fig. 3. Extending the resilience profile to characterise recovery.

As shown in Figure 4, the (average) Rate of Recovery (in blue) is measured by the Measure of Performance Capability (y-axis) divided by time (x-axis). The Actual Recovery Capacity (in red) is measured from the point in which the performance value settles within a range of acceptable levels, however never reaches the Available Recovery Capacity (purple) which we assume is higher. Further analysis of resilience lies in the ability to compare our attributes in our profile (as shown in Section 4).

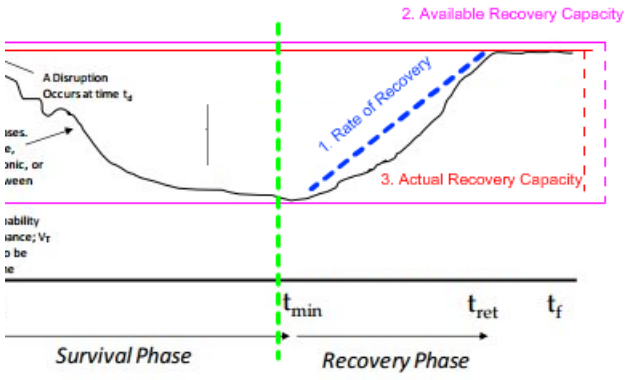


Fig. 4. Recovery Attributes: Rate of Recovery, Available Recovery Capacity and Actual Recovery Capacity.

Given a co-model that can co-simulate a resilience scenario, how can we assess the resilience of the system of interest according to our extended profile? First, we must

ensure that the co-model is *competent* in the sense that it incorporates sufficient features to allow resilience attributes to be assessed; this amount to ensuring we can observe the properties needed for the y-axis of the graphs shown in Figures 2 and 4. This could potentially involve instrumenting the constituent models by adding, e.g. methods in Overture or 20-sim. Second, we can visualise or post-process co-simulation output data to generate the elements needed for analysis against our resilience profile. In either case, the key question lies with the CPS design engineer on exactly what properties need to be measured. In some cases the y-axis measure may be composed of properties both cyber and physical in nature.

4 Example

In this section we describe an example co-model from the smart grids domain. We first explain why we have chosen this application area to evaluate our extended resilience profile. Second we demonstrate how we can use our co-model to produce sufficient data so that we are be able to analyse the resilience of an example grid.

Smart Grids

In seeking to extend the resilience profile so as to better incorporate recovery, we need to validate the approach with example studies. In order to be credible, these should be in a well established and accepted CPS domain [4, 5]. In order to test the capabilities of formalisms, they should be capable of incorporating a series of increasingly demanding architectures, including centralised, distributed, and modular control. Finally, the resilience of the systems of interest should be both desirable to have, and challenging to deliver.

Smart power grids present one such area. Such a grid is a complex ecosystem of heterogeneous (co-operating) entities that interact in order to deliver specific functionality related to the generation, transmission, storage and consumption of (usually electrical) energy [4]. It is an archetypal example of a CPS, in which the power network is overlaid with a computing and communication network, and the two are coupled together into what is generally perceived as a single system of interest [5,26]. CPS technology is seen as an integral part of the smart grid concept and resilience in smart grids is identified as a key challenge [9, 27]. There is interest in increasing local control and autonomy to better match supply and demand in smart grids, and the idea of a local microgrid provides a modular control concept. Finally, significant reliance is placed on the energy supply, even in the face of natural or human-made disruptions. Together these factors make smart grids a suitable example domain for our work.

Co-model Example

To demonstrate our approach to analyse our resilience profile, we have created an example co-model of a Smart Grid with a centralised controller in Crescendo. In our example, we model the physical environment in 20-sim which includes a physical description of

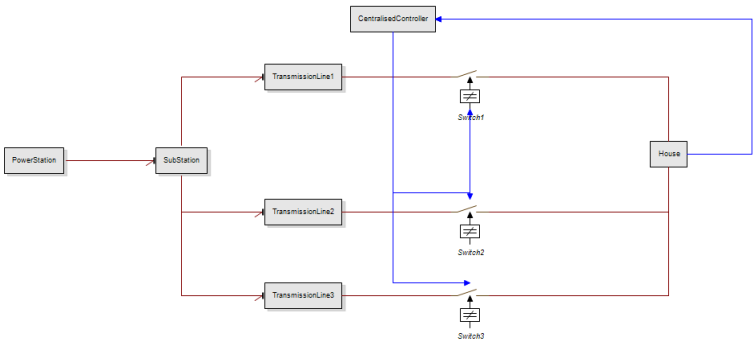


Fig. 5. 20-sim CT model: An example smart grid.

the grid's components ranging from power stations and transmission lines, to sensors and actuators.

Figure 5 is a representation of the CT model in our co-model example in 20-sim. This gives us an overview of our physical system, in which we include 1 PowerStation, 1 SubStation, 3 TransmissionLines, 3 Switches and 1 House. To give context, Figure 6 shows us the CT logic for Switch1. This is an example of the types of CT equations present in our physical model in 20-sim.

```
parameters
   real Ron = 0.00001 {ohm};     // Resistance when switched on
   real Roff = 100000.0 {ohm};      // Resistance when switched off
   real vt = 0.5; //threshold value, switch = on when abs(input)>vt
variables
   real R {ohm};
equations
   R = if input > vt then Roff else Ron end;
   p.u = R * p.i;
```

Fig. 6. A code snippet from within Switch1.

Our DE model is written using VDM-RT. This is where we write our controller logic and any other cyber functionality present in our CPS. Figure 7 shows the controller logic used in our co-model when deciding to switch transmission lines. The logic changes switch state if the voltage level falls below a 'minLevel' threshold (minLevel is a shared parameter between CT and DE models).

Cross-Domain Resilience Scenarios

Cross-domain resilience scenarios are those in which the causal chain transits the boundary between cyber and physical elements. We look at how cyber faults can lead to physical failures. An example of a cyber fault is a digital controller that fails to execute its

```
private controlLoop : () ==> ()
controlLoop() ==
(
  cycles(2)
  (
        -- retrieve the level values from Co-sim
        dcl level1 : real := levelSensor1.getLevel();
        dcl level2 : real := levelSensor2.getLevel();
        dcl level3 : real := levelSensor3.getLevel();
                if level1 >1 and level1 < minLevel then
                (
                        switch1.setClosed();
                        switch2.setOpen();
                );
  );
);
```

Fig. 7. A code snippet of controller logic

control logic even though it is receiving signal data. We model this in Overture by creating a subclass of our centralised controller class. Before we co-simulate, we initialise a faulty version (subclass) of our controller class. This is in line with the fault modelling mechanisms demonstrated in [28]. The fault would cause the switches in our Smart Grid to remain in their initial state. Our controller would never switch to a different transmission line and thus may lead to the propagation of physical failures within the system.

We can also characterise physical faults leading to cyber failures. For example, we may consider a faulty voltage sensor in the house. This sends only the first voltage value. We can model this in 20-sim by creating a second (faulty) implementation of the equation model of our house. Before we co-simulate we can switch to the faulty implementation. The value sent to the centralised controller would be an outdated value, and so the controller cannot switch transmission lines if the voltage level falls below the minimum threshold (shown in Figure 7).

Resilience Profiling

To allow for the analysis of resilience, we must be able to evaluate our resilience profile against data produced from our co-models. To test our scenario, we run our co-model through a co-simulation in Crescendo. The PowerStation in our study provides power to the SubStation. The SubStation uses a step-down transformer to reduce the voltage to 240Volts(V). This power is then split across 3 transmission lines. The first source is sent across TransmissionLine1, where it passes an open switch to get to the house. The second and third sources of power are sent across TransmissionLine2 and Trans-

missionLine3 respectively, where they encounter closed switches and do not reach the house. The power from TransmissionLine2 and TransmissionLine3 are considered as contingencies in the case TransmissionLine1 fails to provide adequate service.

In our scenario, TransmissionLine1 is faulty, which has lead to a steady decrease in voltage output. Our centralised controller checks the voltage level the house receives. If this level falls below a threshold, the centralised controller closes the switch to TransmissionLine1 and opens the switch to TransmissionLine2.

From Figure 8 we see a significant voltage drop from TransmissionLine1 (blue) until around 1 second into the simulation. This is when the controller has closed the switch from TransmissionLine1.

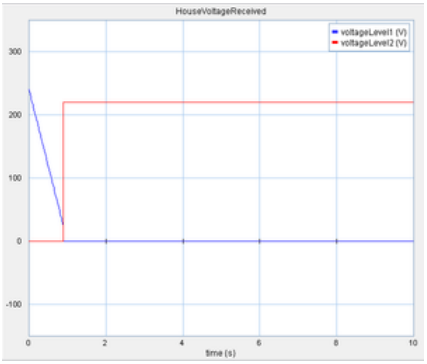


Fig. 8. A comparison of the output voltage from TransmissionLine1 and TranmissionLine2.

Figure 8 shows us the comparison between the voltage received by the house from TransmissionLine1 (blue) and TransmissionLine2 (red). The cyber controller in our DE model recognises when the voltage from TransmissionLine1 falls below our threshold and switches to TransmissionLine2. The house now receives power once again - although at a reduced voltage (220V).

With this data we demonstrate how we can analyse the resilience of our example using our resilience profile. As shown in Figure 9, from our co-model output we can analyse the three facets of our recovery phase.

1. **Rate of Recovery** - In Figure 9 we can see the gradient of the voltage increase is our Rate of Recovery (blue). In our example the switch in our transmission lines happens almost instantaneously which results in a vertical line on our graph. In this case we would have an undefined gradient. From our analysis perspective in the recovery phase this is the optimal rate at which our line can reach its Available and Actual Recovery Capacity.
2. **Available Recovery Capacity** - The Available Recovery Capacity is shown in purple in Figure 9. This is the maximum potential in which our system may recover to. In our case TransmissionLine1's voltage dropped to 0, and the maximum difference the house can receive is the limit of our initial transmission line which is 240V. This follows that our Available Recovery Capacity is 240V.

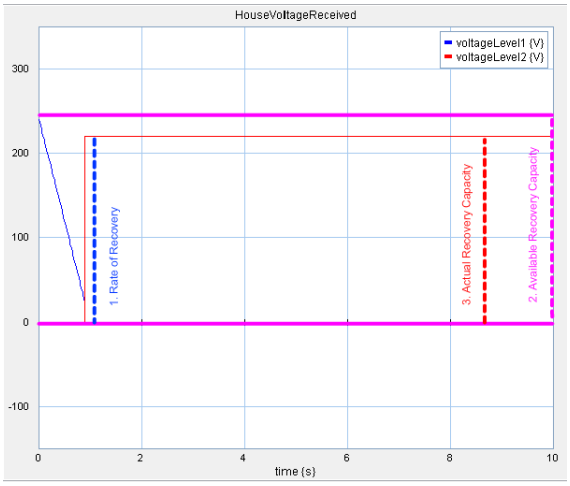


Fig. 9. An analysis of the recovery phase from a Smart Grid co-model with a centralised controller.

3. **Actual Recovery Capacity** - The Actual Recovery Capacity is shown in red. In this case TransmissionLine2 is a line operating at 220V as opposed to 240V. Therefore the Actual Recovery Capacity is 220V. This is not always the case, as performance may be recovered fully, to the Available Recovery Capacity.

Resilience Trade-Off

Crescendo allows us to model resilience scenarios across both cyber and physical domains. We present a resilience scenario in which we can perform resilience profiling and trade-off analysis of our Smart Grid co-model. To perform trade-off analysis we consider two resilience scenarios:

1. **TransmissionLine1 - TransmissionLine2** - TransmissionLine2 produces power almost instantly but at a reduced voltage (220V).
2. **TransmissionLine1 - TransmissionLine3** - TransmissionLine3 produces a higher voltage (240V), but at the expense of time.

In our example we have modelled each scenario, and analysed the results. Figure 10 shows us a comparison between scenario 1 (left) and scenario 2 (right). In scenario 1 we can clearly see that TransmissionLine2 switches almost instantly and provides power to the house at a voltage of 220V. In scenario 2 the house receives power from TransmissionLine3 at around 5 seconds, this is approximately a 4 second delay from when TransmissionLine1 is switched off. It is here a CPS design engineer must consider which is more important, the Actual Recovery Capacity, or the Rate of Recovery.

Figure 11 shows the analysis of the Recovery attribute in our resilience profile. It is shown that the Available Recovery Capacity (purple) is the same in both scenarios (240V), as this is the maximum potential voltage our house can receive under normal

operation. The Actual Recovery Capacity (red) for scenario 1 is 220V, whereas in scenario 2 it is 240V. The Rate of Recovery in scenario 1 is a vertical line and so we have an undefined gradient as the switch is almost instantaneous. However due to the delay of switching in scenario 2, our (average) Rate of Recovery becomes 240V divided by 4 seconds, this gives us a Recovery Rate of 60V/s. With this data a CPS design engineer can assess the trade-off between the Actual Recovery Capacity and the Rate of Recovery facets, in the Recovery Attribute of our resilience profile.

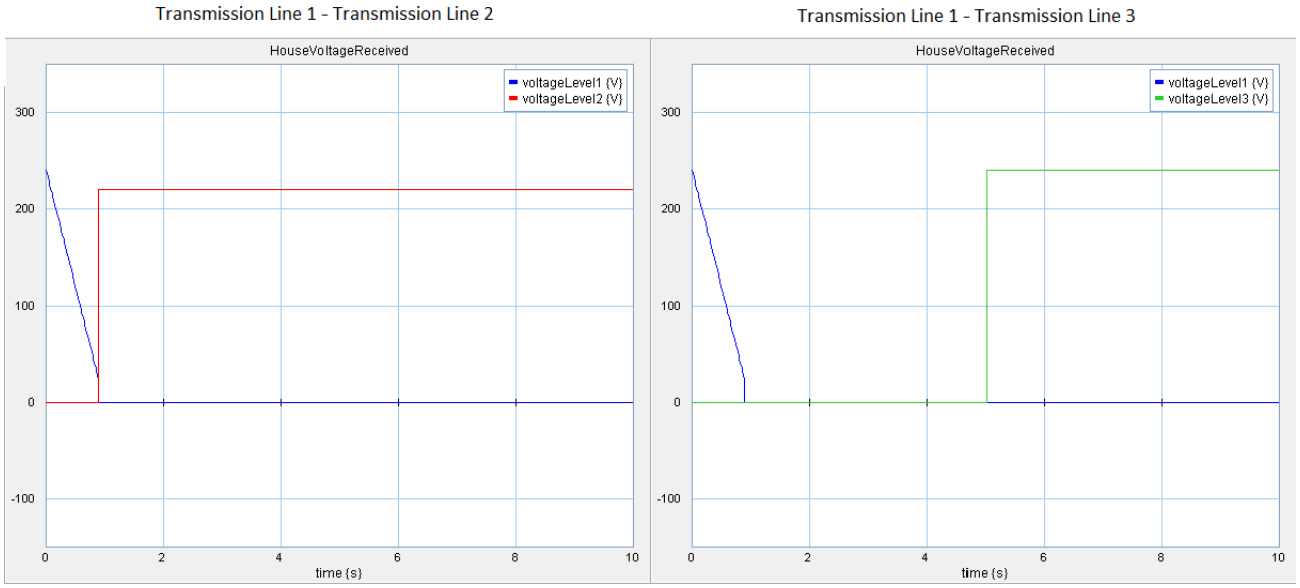


Fig. 10. A comparison of resilience scenarios.

An important note to make about our trade-off analysis example as seen in Figure 11, resides in the fact that our Rate of Recovery facet (blue) is comprised of our voltage level and time. The Rate of Recovery facet will always rely on the relation and importance of the y-axis and the x-axis on our graph. We calculated the (average) Rate of Recovery in Figure 11 for scenario 2 as 60V/s, however we can see from the graph that there is no voltage supplied at all until 5 seconds. This is due to the discrete nature of switching transmission lines in our model. This information may be of interest to a CPS design engineer. For other resilience scenarios, the engineer may wish to have some values of performance available leading up to the Actual Recovery Capacity, as opposed to our co-model example, in which the transmission line is either switched on or off. In this case we can analyse our recovery phase further at more time steps. It is for reasons like this that we seek to extend our resilience profile to a more refined and sophisticated version. Although this is somewhat a trivial example, it describes the basis of our resilience profile and demonstrates how we can perform trade-off analysis between attributes and facets.

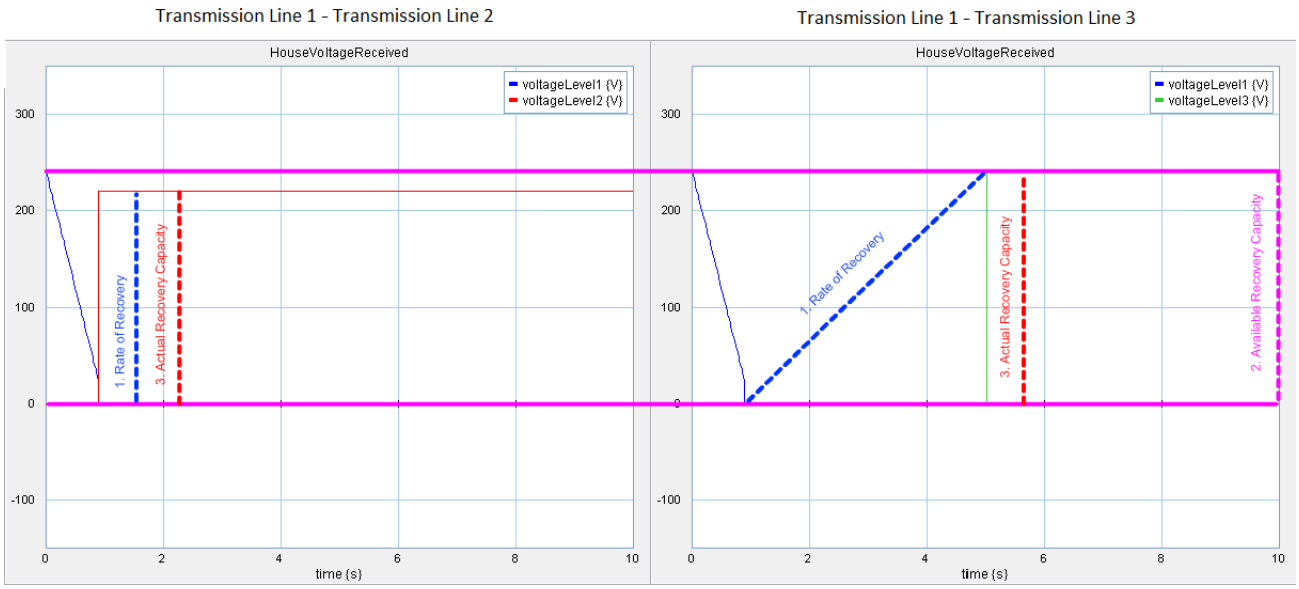


Fig. 11. An analysis of trade-off.

5 Evaluation and Further Work

We have provided motivation for being interested in resilience as a property of cyber-physical systems. Recognising that it is a multi-faceted property, we have extended an existing resilience profile with additional features that characterise recovery phases, and demonstrated that it is possible to assess this aspect of resilience on CPS co-models using Crescendo by means of a simple example from the domain of smart grids. We have illustrated resilience scenarios that cross the cyber-physical boundary and have demonstrated the potential to assess trade-offs.

Our work is at a preliminary stage, but we believe that there is potential to build useful methods for CPS resilience engineering on top of co-modelling technology of the kind pioneered by Overture, 20-sim, Crescendo and INTO-CPS. We naturally expect to progress from the Crescendo framework to INTO-CPS, exploiting the nascent SysML architectural modelling methods developed in that context as well as the improving performance of tools. We hope to take advantage of order of magnitude improvements in co-simulation and design space exploration performance that will allow us to evaluate our resilience profile on system architectures that reflect the potential complexity of emerging CPSs. These will include decentralisation of control, and eventually increased localised responsibility and autonomy in smart grids. Modular architectures such as microgrids (in which local smart grids negotiate with one another to trade energy) may serve to improve or impede facets of resilience, for example. We will assess the extent to which the abstractions currently available in VDM-RT help or hinder the modelling of such structures.

As discussed at the end of Section 4, we seek to extend our profile in order to better encapsulate the interesting design decisions that a CPS design engineer may face when considering the resilience of a system. In the future we look towards generating data using the INTO-CPS tool chain described in Section 1. This would allow us to profile the resilience of models of CPSs that have been generated with different semantics across a variety of simulation tools.

Finally, we note that there are – at least at a certain abstraction level – significant similarities between CPSs in different infrastructure domains. We might expect that some modelling patterns might be shared between, say, negotiating microgrids, and negotiating traffic flow systems. There is therefore significant potential in identifying and exploiting those patterns as a means of sharing experience between otherwise quite separate application domains.

As societal and business dependence on cyber-physical systems grows, we believe that the need to take a systematic view of resilience – and in particular recovery – will only grow. Through their support for varied levels of abstraction, and their capacity to integrate with hitherto isolated tools, Overture and VDM have a vital role to play in addressing this requirement in the future.

Acknowledgments We would like to thank the anonymous referees for valuable input on this work. The work presented here is partially supported by the INTO-CPS project funded by the European Commission's Horizon 2020 programme under grant agreement number 664047.

References

1. S. McChrystal, T. Collins, D. Silverman, and C. Fussell, *Team of Teams: New Rules of Engagement for a Complex World*. Portfolio Penguin, October 2015.
2. E. A. Lee, "CPS foundations," in *Proceedings of the 47th Design Automation Conference*, DAC '10, (New York, NY, USA), pp. 737–742, ACM, 2010.
3. M. Broy, "Engineering cyber-physical systems: Challenges and foundations," pp. 1–13, 2013.
4. S. Karnouskos, "Cyber-physical systems in the smartgrid," in *Industrial Informatics (INDIN), 2011 9th IEEE International Conference on*, pp. 20–23, July 2011.
5. J. Taneja, R. Katz, and D. Culler, "Defining cps challenges in a sustainable electricity grid," in *Cyber-Physical Systems (ICCPS), 2012 IEEE/ACM Third International Conference on*, pp. 119–128, April 2012.
6. "Summary of the 2015-16 sector resilience plans." United Kingdom Cabinet Office, April 2016.
7. A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, pp. 11–33, Jan 2004.
8. "Keeping the country running; natural hazards and infrastructure." United Kingdom Cabinet Office, October 2011.
9. Q. Zhu and T. Basar, "Robust and resilient control design for cyber-physical systems with an application to power systems," in *Decision and Control and European Control Conference (CDC-ECC), 2011 50th IEEE Conference on*, pp. 4066–4071, Dec 2011.

10. M. Broy, "Engineering cyber-physical systems: Challenges and foundations," pp. 1–13, 2013.
11. L. Zhang, "Modeling large scale complex cyber physical control systems based on system of systems engineering approach," in *Automation and Computing (ICAC), 2014 20th International Conference on*, pp. 55–60, Sept 2014.
12. A. Hellinger and S. Heinrich, "Cyber-physical systems driving force for innovation in mobility, health , energy and production," tech. rep., acatech - National Academy of Science and Engineering, 2011.
13. C. Brooks, C. P. Cheng, T. H. Feng, E. A. Lee, and R. Von Hanxleden, "Model engineering using multimodeling," tech. rep., DTIC Document, 2008.
14. R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine, "The algorithmic analysis of hybrid systems," *Theoretical computer science*, vol. 138, no. 1, pp. 3–34, 1995.
15. P. G. Larsen, J. Fitzgerald, J. Woodcock, P. Fritzson, J. Brauer, C. Kleijn, T. Lecomte, M. Pfeil, O. Green, S. Basagiannis, and A. Sadovykh, "Integrated tool chain for model-based design of cyber-physical systems: The into-cps project," in *2016 2nd International Workshop on Modelling, Analysis, and Control of Complex CPS (CPS Data)*, pp. 1–6, April 2016.
16. P. G. Larsen, J. Fitzgerald, J. Woodcock, R. Nilsson, C. Gamble, and S. Foster, "Towards semantically integrated models and tools for cyber-physical systems design," in *Proc. 7TH Intl. Conf. on Leveraging Applications of Formal Methods, Verlification and Validation (ISoLA 2016)*, Springer, in press, 2016.
17. E. Hollnagel, D. D. Woods, and N. Leveson, *Resilience engineering: Concepts and precepts*. Ashgate Publishing, Ltd., 2007.
18. E. Hollnagel, J. Paries, D. W. David, and J. Wreathall, *Resilience engineering in practice: A guidebook*. Ashgate Publishing, 2010.
19. S. M. Mitchell, *Resilient engineered systems: the development of an inherent system property*. PhD thesis, Texas A&M University, 2007.
20. C. G. Rieger, D. I. Gertman, and M. A. McQueen, "Resilient control systems: next generation design research," in *Human System Interactions, 2009. HSI'09. 2nd Conference on*, pp. 632–636, IEEE, 2009.
21. L. Mili and N. V. Center, "Taxonomy of the characteristics of power system operating states," in *Taxonomy of the characteristics of power system operating states*, 2011.
22. S. Carpenter, B. Walker, J. Anderies, and N. Abel, "From metaphor to measurement: Resilience of what to what?," *Ecosystems*, vol. 4, no. 8, pp. 765–781, 2001.
23. C. of the European Communities, "Disaster resilience: Safeguarding and securing society, including adapting to climate change."
24. S. Jackson, *Architecting resilient systems: Accident avoidance and survival and recovery from disruptions*, vol. 66. John Wiley & Sons, 2009.
25. M. Pflanz, *On the Resilience of Command and Control Architectures*. PhD thesis, George Mason University, 2011.
26. O. Yagan, D. Qian, J. Zhang, and D. Cochran, "Optimal allocation of interconnecting links in cyber-physical systems: Interdependence, cascading failures, and robustness," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 23, no. 9, pp. 1708–1720, 2012.
27. P. Smith and A. Schaeffer-Filho, "Management patterns for smart grid resilience," in *Service Oriented System Engineering (SOSE), 2014 IEEE 8th International Symposium on*, pp. 415–416, IEEE, 2014.
28. J. Fitzgerald, K. Pierce, B. Bos, and C. Gamble, "Co-modelling of faults and fault tolerance mechanisms," in *Collaborative Design for Embedded Systems*, ch. 9, pp. 185–198, Springer, 2014.

Considering Abstraction Levels on a Case Study

Casper Thule and René Nilsson

Aarhus University, Department of Engineering
Finlandsgade 22, 8200 Aarhus N, Denmark
{casper.thule,rn}@eng.au.dk

Abstract. Cyber-physical systems consist of cyber parts controlling physical entities and this close interaction can be challenging. To manage the complexity of CPSs it can be useful to create models of the constituent components and simulate these in what is called a co-simulation. This can help in building prototypes and discovering undesired behaviour. When creating such models it is important to choose the right abstraction level to enable prototyping of various parts of a system. For this purpose it can be advantageous to create models at different levels of abstraction. This paper describes a case study based on a continuous time model and a discrete event model of a quadrotor unmanned aerial vehicle. Abstractions are considered a tool to gain insights and manage the complexity of a given system, and therefore the discrete event model has been abstracted to allow focusing on high-level waypoint behaviour. The results show that the abstracted models resemble the original model with respect to the goals of the abstraction.

Keywords: Cyber-Physical Systems, Co-Simulation, Model-Based Design, Crescendo, Overture, VDM-RT, DESTECS, INTO-CPS, FMI

1 Introduction

Cyber-Physical Systems (CPSs) incorporate sensing, actuating, computing, and communication from a cyber perspective [14]. Such systems are becoming a vital part of our society, which relies on them in cars, trains, medical devices, and so forth. The development of CPSs poses a challenge because of the close interaction between cyber parts and physical entities. This challenge is also due to the necessity of taking the complexity of the physical domain into account when developing control applications [12].

To support the development of CPSs it can be useful to create models of the constituent components that jointly form a given system. A model is an abstract description of a component, where irrelevant details are abstracted away [10,16]. In case these components are expressed in an executable subset of a notation, a collaborative simulation (a co-simulation) can be employed, which can couple models created in different formalisms. One way of performing such co-simulations is presented in the DESTECS[1] project, where Continuous Time

[1] http://www.destecs.org/, visited on 28 Oct. 2016.

(CT) and Discrete Event (DE) models can be co-simulated. The approach of using models is captured in "model-based design", a methodology that describes how models can be used when developing new systems [9]. In this paper the physical system dynamics are modeled in the CT modeling domain and the digital control in the DE modeling domain [5]. The model-based design approach excels in managing complexity [4], which is an important attribute as it is necessary to allow for increasing complexity of CPSs [8].

This paper concerns reflecting on the chosen level of abstraction used in modeling the DE side of a quadrotor Unmanned Aerial Vehicle (UAV), shown in Fig. 1, in a case study [6]. Furthermore, it demonstrates how two abstractions of the DE model can serve for prototyping purposes in a learning process. The collaborative model of the UAV consists of a CT model representing the physical system dynamics and a DE model representing the control logic. The CT model captures the system properties that are continuous in time, e.g. physical phenomena, whereas the DE modeling domain captures system functionality that is executed at discrete time intervals, e.g. control algorithms.

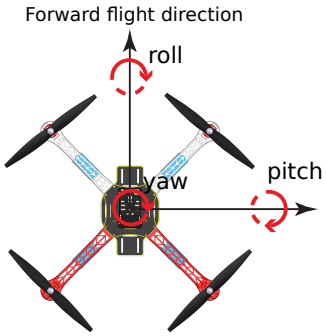


Fig. 1. The model of the UAV. Yaw, pitch, and roll are Euler angles, which are described in Sect. 3.

The Crescendo technology developed in the DESTECS project enables the possibility of collaboratively modeling and simulating the UAV. The CT parts of the system are modeled in 20-sim using differential equations, block diagrams, and bond graphs [2]. Discrete events are modeled in Overture using VDM-RT [11,13]. The coupling is achieved through the Crescendo tool[2], which functions as a co-simulation engine for the CT model and the DE model. The DE and CT model for the UAV are jointly referred to as co-model. The Crescendo technology is limited in the sense that it supports a maximum of one CT and one DE model, which have been designed specifically for the Crescendo technology with proprietary interfaces.

[2] http://crescendotool.org/, visited on 28 Oct. 2016.

Using the tools developed in the INTO-CPS project [3] it is possible to perform co-simulations with models adhering to the the standardized Functional Mock-up Interface (FMI). FMI describes an interface in the C language and a model/component implementing this along with providing a model description is called a Functional Mock-up Unit (FMU). The INTO-CPS project concerns development of CPSs from requirements through design, down to realisation in hardware and software. It encompasses the tools 20-sim and Overture along with functionality for e.g. performing Design Space Exploration (DSE), verification, and co-simulation with hardware in the loop [7]. Therefore, the CT and DE models mentioned above can be represented as FMUs and co-simulated using INTO-CPS technology.

After this introduction Sect. 2 presents the development history of both the CT and DE models of the UAV model and the motivation for the work presented in this paper. Sect. 3 will then give an overview of the CT model. Next, Sect. 4 describes the DE model, which is based on an open source quadrotor control application called APM:Copter[3]. To improve the DE model and gain more insight into the development of control logic for a UAV Sect. 5 presents a case study and reflects on abstraction levels. Furthermore, a proposal for a new DE model is presented. Afterwards, Sect. 6 describes the transition from the Crescendo technology to the INTO-CPS technology. Finally, Sect. 7 presents concluding remarks on this paper and Sect. 8 outlines the future work on the co-model.

2 History

The work presented in this paper builds upon an earlier project in which a detailed co-model of a UAV was developed with the Crescendo technology [6]. The project used a CT-first approach [5], where initial work was commenced on the CT model, since the developers had little or no prior knowledge of UAV dynamics. Development of a detailed CT model was carried out using a refinement process, resulting in three CT models at different abstraction levels:

Conceptual model: A pure CT model of a "Flying box". This model was used to give a thorough understanding of forces and moments and coordinate systems in a UAV setting.

Generic component model: This model is a refinement of the conceptual model. Rather than a flying box, this model captures all components of a quadrotor UAV and provides valuable information about components and their interaction. Additionally, it identifies the interface to the DE model and enables co-simulations.

Specific device model: The specific device model is refined by measuring real system dynamics, such as motor/rotor characteristics, of a specific UAV and applying these measurements in the CT model.

Development of the DE model did not follow the same refinement process. Instead a detailed DE model was developed by reverse engineering parts of the existing

[3] https://www.dronecode.org/dronecode-software-platform, visited on 28 Oct. 2016.

open source framework APM:Copter. This lead to a comprehensive DE model with parts that were not fully understood, due to undocumented values and code in the framework. This DE model will henceforth be referred to as the original DE model.

The motivation for this paper is to demonstrate how an abstract version of the original DE model can resemble the functionality and therefore be used to gain insights concerning the control logic of a UAV. Additionally, the purpose of this paper is also to begin the development of a new DE model because of the undocumented implementation of the APM:Copter framework. This is necessary in order to thoroughly understand the parts that make up the control logic for a UAV and to create an architecture that improves the possibility of prototyping.

3 The Continuous Time Model

The CT model describes the physical dynamics of a DJI F450 Flamewheel quadrotor UAV, illustrated in Fig. 1. The UAV has a cross airframe and is propelled by four rotors rotated by four motors driven by four Electronic Speed Controllers (ESCs). Despite of four rotors, the system is inherently unstable, causing a need for a controller [1]. The controller reads inputs from various sensors and pilot guidance through a telemetry system. Using complex control algorithms, the controller is capable of keeping the UAV stable, as well as steering the UAV, by adjusting the four motor speeds individually. The control algorithms are realized in the DE side and described further in Sect. 4.

When developing UAV models, both in the DE and CT side, it is important to have a common notion of terms and their meaning concerning UAV position and orientation. These are most often described using two coordinate systems and a rotation convention. For simplicity the mathematical basis is not described here, but the main terms are described as follows:

Position: The position of a UAV refers to the spatial location. This is often expressed with latitude, longitude, and altitude.

Attitude: The attitude of a UAV refers to the orientation (also referred to as angular position). This can be expressed using Euler angles roll, pitch, and yaw. Each Euler angle is a measure of how much the UAV is rotated around one of its axes, illustrated in Fig. 1.

An overview of the components and the interface to the CT model are presented in Fig. 2.

The CT model receives motor setpoints for all motors from the control algorithm at the DE side through an interface defined in a contract. These setpoints drive the motor controllers (ESCs), which in turn drives the motors and rotors. The airframe of the UAV is represented as a rigid body that can move and rotate in space, when affected by forces from the rotors. The rigid body is described with ideal differential equations, from which both spatial and angular accelerations, velocities, and positions are extracted. These extracted values serve as input to various sensors, which can simulate noise, inaccuracies,

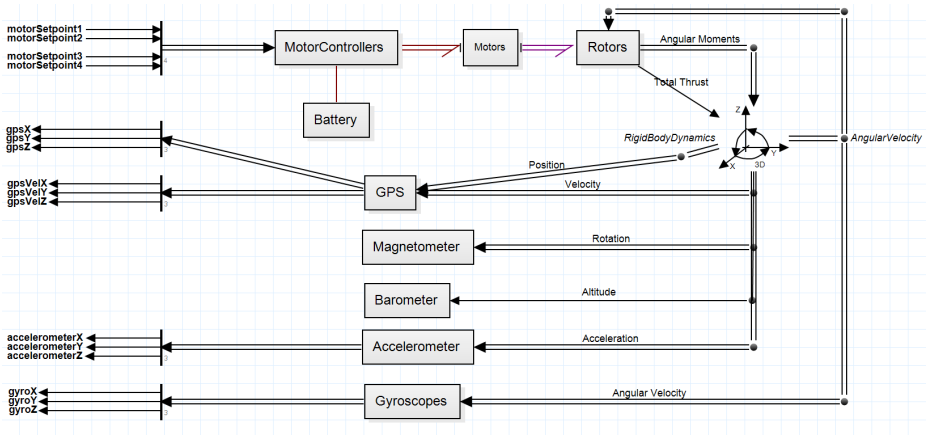


Fig. 2. CT model overview

and rounding errors. Sensor outputs are made available to the control algorithm in the DE side through the contract interface.

4 The Discrete Event Model

The original DE model was developed by reverse engineering parts of the open source framework APM:Copter, which is a control platform for multi-rotor UAVs. The architecture in the model resembles the APM:Copter project, but all irrelevant functionality is abstracted away. This includes all low level software such as operating system, drivers, and wireless communication (telemetry). Additionally, sensor fusion has been overly simplified, since ideal sensor data is available from the CT model, thus removing the sensor fusion requirement[4].

An overview of the main components of the DE model is shown in Fig. 3 whereas the components of the flight control algorithm are shown in Fig. 4.

The main class of the model `Arducopter` starts a `Scheduler` and a `Flight controller`. To improve model fidelity, sensor values are updated periodically by the scheduler to emulate the real update frequencies of the various sensors. The flight controller takes input from a pilot and from sensor data, on which sensor fusion has been performed, and is responsible for calculating desired accelerations for the UAV. These accelerations are translated, by the `Motors` class, into motor setpoints for each motor. The translation involves a complex tradeoff between roll, pitch, and yaw accelerations and total thrust, as well as motor clipping and voltage scaling. The `MotorsQuad` class is shown to illustrate that the `Motors` class can be extended to support any number or configuration of motors, thus increasing complexity even further.

[4] Note that sensor fusion is a very important aspect when considering robustness against noise and uncertainties in sensors.

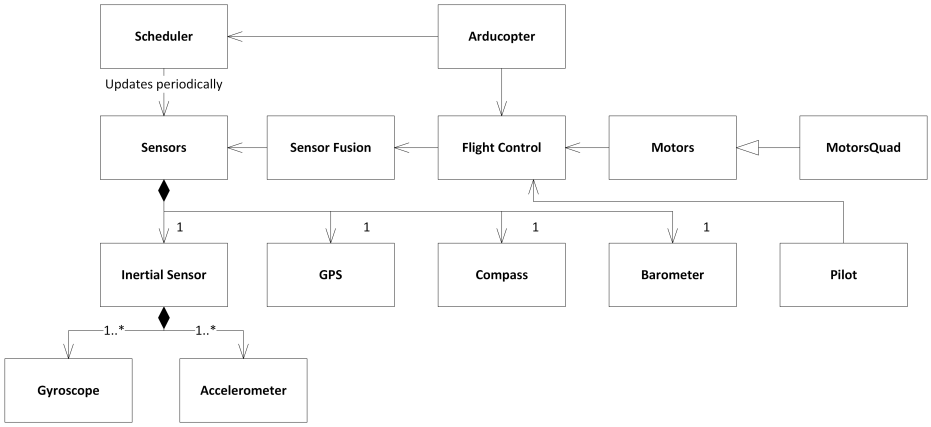


Fig. 3. Original DE model overview.

The `Flight Control` class contains the core functionality of the model and is probably also the most complex part (see Fig. 4 for an overview of the components). It can operate in multiple flight modes, which make use of either an attitude controller or both an attitude and a position controller. The attitude controller is capable of obtaining and maintaining any given attitude, whereas the modeled position controller is only capable of controlling the altitude and not the spatial location (latitude/longitude). In practice this means that it is not possible to make the UAV fly to a given location, but it is however possible to let the UAV move in any direction, simply in an uncontrolled manner. The attitude and position controllers depend on a number of low level controllers, such as Proportional-Integral-Derivative (PID) controllers, the simpler PI or P controllers, and a somewhat similar square root controller. To complicate things even further, most of these controllers are cascaded and added a feed-forward term in order to improve control capabilities. Additionally, a number of different filters are used to remove unwanted noise and vibrations caused by the fast spinning rotors.

5 Abstracting the Original Discrete Event Model

This section will investigate whether an abstraction to the original model described in Sect. 4 can be used for implementing vertical waypoint behaviour. The vertical waypoint behaviour was chosen, because the original DE model only supports vertical movement. In order to create a proper abstraction it is necessary to understand the task at hand. First, we consider the basic hardware of a UAV. This is shown in Fig. 5, which contains the following sensors and actuators:

Barometer: A barometer is used to measure atmospheric pressure and can be used to measure altitude.

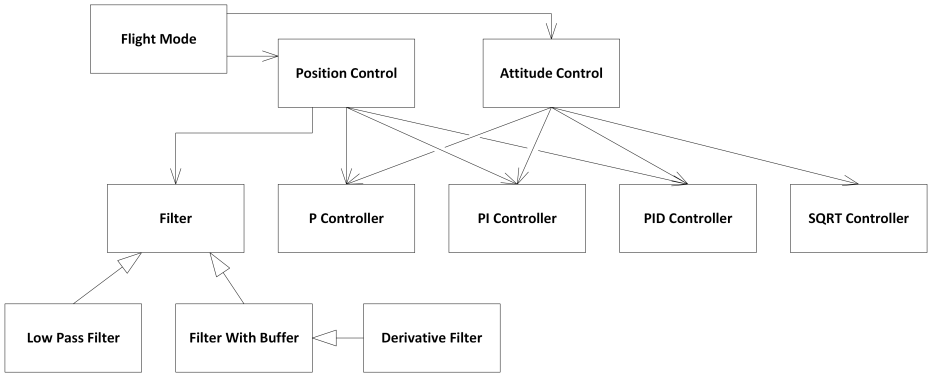


Fig. 4. Flight control components

Accelerometer: An accelerometer measures acceleration forces, and can be used to get the orientation of the UAV relative to earth's surface.
Gyroscope: A gyroscope measures angular velocity, e.g. yaw, pitch, and roll velocity and can therefore be used along with the accelerometer to get the orientation of the UAV.
Magnetometer: A magnetometer measures magnetic fields and can be used to detect earth's magnetic field, thereby providing a magnetic north.
GPS: A GPS can be used to measure altitude, latitude, and longitude and thereby provide a position in space.
Motor(s): The motor(s) are used for the rotor(s) on the UAV. One or more motors can be used.

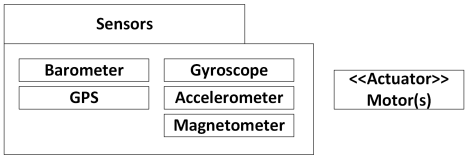


Fig. 5. Basic sensors and actuators of a UAV

Next, we consider the basic control flow of a UAV as shown in Fig. 6. This consists of the following components:

Sensor Input: Input from the sensors listed above.
Sensor Fusion: In order to gain useful information the sensor data must be fused together. For example, the accelerometer, gyroscope, and magnetometer is commonly referred to as an Inertial Measurement Unit (IMU). The information obtained from these three sensors is used to get the best possible position information, as they each have their strengths and weaknesses. The result of the sensor fusion is position and velocity.

Flight Mode: This construct is the control logic and it depends on the flight mode. A waypoint is a position in space, where the drone should fly to, and it can be followed by other waypoints. Pilot means that a pilot gives commands to the drone, e.g. throttle and yaw rotation. The output from the flight mode is roll, pitch, and yaw angles along with a desired velocity.

Motor(s): The throttle value is sent to one or more motors.

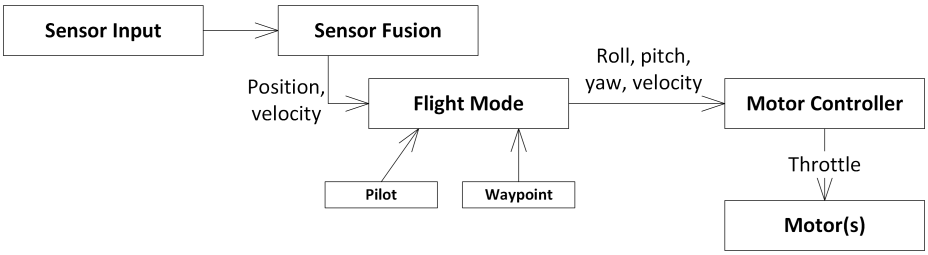


Fig. 6. Basic control flow of a UAV

The next step is to consider which parts to include in the abstraction in order to model the high-level waypoint behaviour for the UAV. High-level behaviour in the context of this paper is used to describe functionality closely related to the goal of a feature. For example, the goal of the waypoint behaviour is for the UAV to fly to a waypoint, and once this waypoint is reached it should fly to the next waypoint. Low level details are considered as the constructs that enables the high-level functionality, e.g. reading sensor data. The waypoint behaviour is based on the position of the drone, and once it coincides with the position of a given waypoint, the UAV has visited the waypoint. It only depends on the position and velocity of the UAV and not how they are retrieved. Thus the sensor fusion component can be abstracted away when developing high-level waypoint behaviour as GPS coordinates is enough to provide a position and calculate a velocity.

The abstraction introduced above is realised in an implementation that consists of two DE models that only differ in the feedback system. One of the models, henceforth referred to as the P model, uses a proportional gain controller. The other model, called the PID model, uses a PID controller instead of a P controller. Both models only make use of the GPS sensor component and the motor actuators. The P and PID controller is based on altitude and used to provide input to the throttle value. The model is shown in Fig. 7 and resembles the basic flow depicted in Fig. 6. The `UAV` class initiates and starts up the system. The GPS data is retrieved through the `GPS` class, and the `SF_GPS` class is created to provide an abstraction to the low level `GPS`, as sensor fusion is to be implemented later on. `PositionControl` contains the functionality to retrieve the GPS data, calculate a throttle value by converting the control value from `P/PID`, and to check whether the current position is within the bounds of

the current waypoint. The `FlightController` is responsible for the high-level functionality for waypoint behaviour. This means it uses `PositionControl` to calculate a throttle value and checking whether the current waypoint is within bounds along with setting a new waypoint once a waypoint has been reached. Finally, `MotorController` converts a single throttle value to a throttle value for each motor, represented by the class `Motor`.

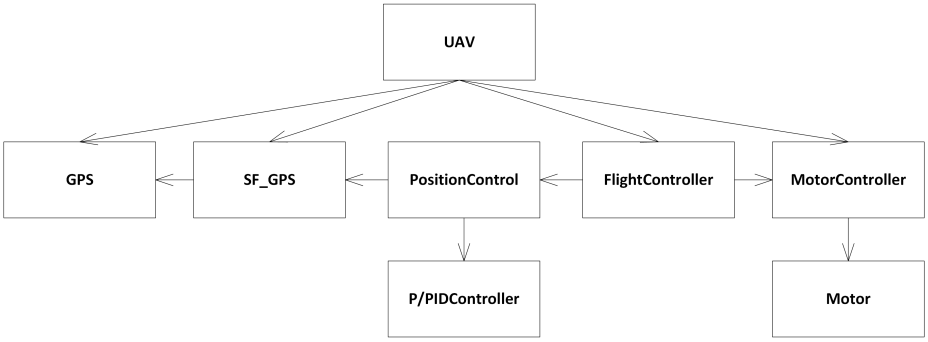


Fig. 7. The P and PID models.

A waypoint is considered reached with an approximation when the drone is vertically within 50 centimeters of the waypoint. The waypoints are located in an altitude of: three meters, two meters, and four meters. This combination forces the drone to respectively increase, decrease, and increase velocity. This is to ensure that the drone does not reach a subsequential waypoint because it overshoots the target. Overshooting means that the drone goes beyond its current target because of excessive velocity. The vertical waypoint behaviours of the three DE models is shown in Fig. 8. The figure shows that the drone reaches the waypoints in all three DE models with different levels of overshooting as shown in Table 1. The original model shows the best performance in terms of how fast it reaches the three waypoints. However, in general the PID model has the lowest total error with ~3.34 meters, where the P model has ~6.78 meters and the original model has ~3.89 meters. Because the P and PID model roughly resemble the original model they are useful for prototyping. Deciding whether a model is useful or not depends on the particular case. As co-simulation deals with approximations of real systems in order to achieve a reasonable simulation speed, it can be useful to define tolerances. A tolerance can be used to define whether a simulation result is "close enough" to the real system and therefore considered useful [15].

Another interesting measure is how "simple" these abstractions are compared to the original model. To compare the simplicity Lines Of Code (LOC) is used as measurement unit. The original model consists of 42 files with 2270 LOC along with an additional Java library, the P model consists of 12 files with 307 LOC,

and the PID model consists of 12 files with 333 LOC[5]. Thus the original model is roughly seven times larger than the P and PID model.

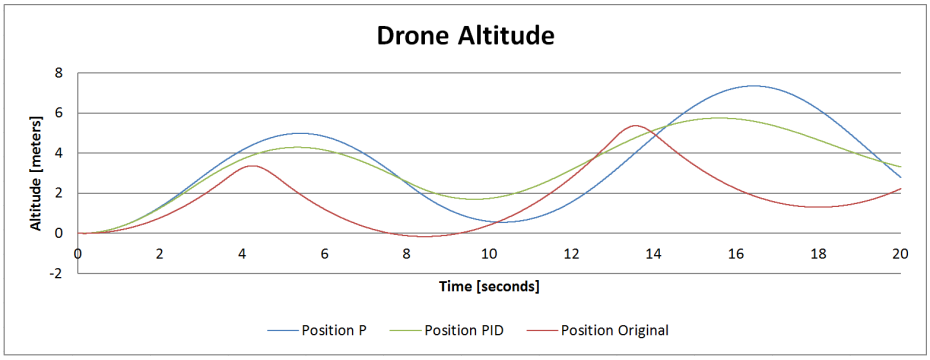


Fig. 8. The altitude of the drone following the waypoints. The waypoint functionality was implemented based on different abstractions.

Table 1. Overshooting for the three DE models.

Altitude Target	Model	Actual	Overshoot
3	P	4.983961	1.983960533
	PID	4.287995836	1.287995836
	Original	3.367945477	0.367945477
2	P	0.552304337	1.447695663
	PID	1.696441193	0.303558807
	Original	-0.161162787	2.161162787
4	P	7.345629313	3.345629313
	PID	5.745529086	1.745529086
	Original	5.363993294	1.363993294

The P and PID models were created by a software engineer within six hours without any prior knowledge of UAVs. This illustrates that using the model and prototype approach it is possible for a person without knowledge of mechanics to develop the high-level waypoint behaviour, and leave the development of low level details to experts within the given discipline. Implementing the waypoint behaviour in this fashion is an example of focusing on what should be accomplished rather than how. This leads to well defined interfaces, because the high-level behaviour, the "what", is implemented based on what the lower level details, the

[5] The original model does contain more functionality than the P and PID model, however the difference is significantly larger than what the additional functionality accounts for.

"how", should offer. It is illustrated in Fig. 9, where the interface to the sensor fusion component is already in place. The sensor fusion component can thus be refined without affecting the high-level waypoint behaviour[6]. The prototype models can also be used as a reference to the development of the low level functionality. In this case study it can be seen that the PID model has a smaller overshoot than the original model when moving from a high altitude to a lower altitude. Was this known before implementing the original model, then it might have affected the resources spent on tuning the variables of the used feedback systems.

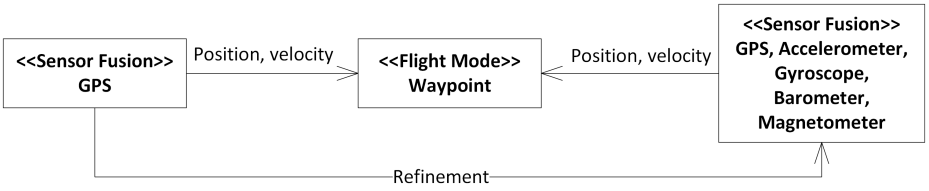


Fig. 9. Refinement of the `Sensor Fusion` component, without altering the interface to the high-level waypoint behavior.

It is important to consider the abstraction level when prototyping new features. Choosing the right abstraction level is an engineering skill that depends on the purpose of the abstraction. If the purpose is to prototype high-level behaviour, then choosing a too low abstraction level makes it a cumbersome task to prototype new features, as it is necessary to take low level details into consideration. However, if the purpose is to prototype low level details, then it is necessary to make these easily accessible and possibly abstract high-level code. Another important thing is not choosing a too high abstraction level if the goal is to code generate working code. In this case it can become difficult to unify the abstractions with the fully working implementation. The idea of separating abstractions into this sort of levels is described by Wing [16], who uses the term "layers". She states that we are working with at least two layers of abstraction simultaneously: the layer of interest and the layer below; or the layer of interest and the layer above.

Having a focus on abstractions and prototyping promotes a breadth approach rather than a depth approach as shown in Fig. 10. The breadth approach means that some can work on high-level behaviour using simple models of the low level details, while others work on the low level details. The depth approach means that part of the functionality is fully implemented before moving on. This requires low level details to be implemented, as they are the building blocks for the high level functionality. Therefore, the breadth approach can allow more people to work on different levels, however functionality might not be fully implemented as fast as when using the depth approach. Using the depth approach, some functionality

[6] It is not always the case that the same high-level component can be reused when other models have been refined.

will be fully implemented, whereas development of other functionality will not have begun. Thus, there are advantages and disadvantages to both approaches, and which approach to use depends on the given project. Another important aspect to consider is whether the effort spent on creating models and prototyping provide sufficient insight for the task to balance or exceed the resources spent [3].

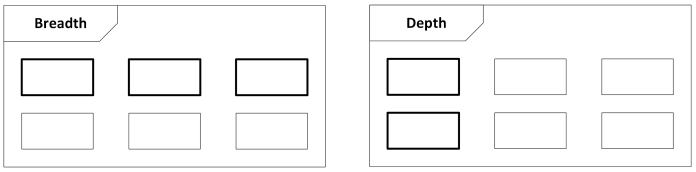


Fig. 10. Breadth and depth approach. The thick boxes represent fully implemented functionality, and the normal boxes represent shallow implementations, e.g. models or implementation to be done.

6 Transitioning to INTO-CPS Technology

The INTO-CPS toolchain offers the possibility of performing co-simulations with any number of FMUs. Because FMI is standardized the execution of co-simulations using INTO-CPS technology is not limited to models expressed in VDM-RT using Overture and 20-sim. Thereby any tool capable of generating an FMU can be used in the development process. This allows for choosing the tool best suited for a given task provided that the tool can generate an FMU. Furthermore, the INTO-CPS toolchain is being actively developed and supports verification, DSE, traceability, and other features. Therefore it is of interest to transition to INTO-CPS technology for the further development of the UAV, such that these features can be utilized. Furthermore, tools using FMUs can also be used in the development process e.g. MathWorks Simulink for low level control testing.

The transition from Crescendo to the INTO-CPS technology is aided by the tools 20-sim and Overture through support of FMU generation. The necessary modifications to the DE and CT model are limited to the contract or interface between the two models. When using the Crescendo tool, shared variables between the two models are declared explicitly in 20-sim as `imported` or `exported` `external` signals. In Overture such shared variables are declared implicitly as instance variables and linked in a separate link file. With the INTO-CPS tool chain and the use of FMUs, these interface declarations must be changed. In 20-sim the explicit declarations are no longer needed. To ease the transition between the two technologies, the model can be split into two connected 20-sim blocks; one with the interface declaration and one with the model functionality. By doing so, the model can still be used with the Crescendo technology and the

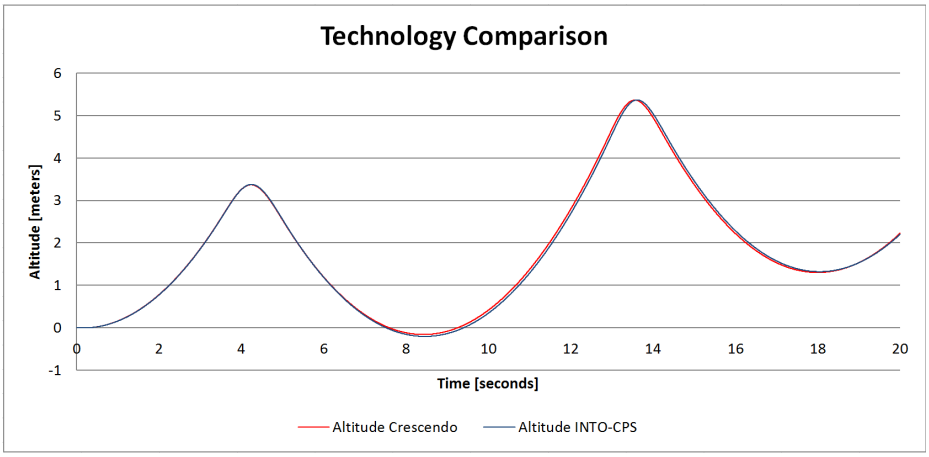


Fig. 11. Comparison of simulation results from Crescendo and the INTO-CPS tool chain

block containing the model functionality can be exported as an FMU and used with the INTO-CPS technology. In Overture shared variables must be declared using the construct `FMI Port`, which contains an instance variable. Therefore shared variables are now declared explicitely and they are accesable via get and set functions.

Once FMUs of both models have been created, they can be imported into the INTO-CPS app, where they can be connected, validated, and executed as a co-simulation. A comparison of the results from the original DE model run with Crescendo and the INTO-CPS app is shown in Figure 11. The comparison shows only minor differences between the two simulation technologies. This can be due to several reasons, e.g. varying step sizes in the two technologies. In the INTO-CPS app, the step size is determined entirely by the Overture FMU, whereas in Crescendo the 20-sim model generates additional intermediate steps, which seems to be related to the step size of the integrator in 20-sim. The data shown in Fig. 11 consist of ~8000 samples from the INTO-CPS app, and ~32000 samples from Crescendo. The FMU for the DE model is a tool wrapper for Overture and therefore the same code is executed but with an FMI layer on top, which can also make a difference. Additionally, it can also be due to the rounding performed by the co-simulation engine used in INTO-CPS.

7 Concluding Remarks

In this paper a full model of a DJI F450 Flameweel quadrotor UAV has been presented. This included the description of a CT model created with the tool 20-sim and a detailed DE model, called the original DE model, created with the open-source tool Overture. The CT model in 20-sim and DE model in Overture was coupled and co-simulated using the Crescendo tool. The CT model contains

all relevant electronics, mechanics, and sensors of the UAV. The original DE model is a reverse engineering of the APM:Copter project with irrelevant functionality abstracted away. It contains functionality to hold an attitude, hold an altitude, and respond to pilot instructions.

This paper also describes a case study, where the original DE model has been abstracted into two simplified DE models called the P model and the PID model. These models are simplified in the sense that they only use the GPS sensor and other low level details are abstracted away such as a low pass filter. The P and PID models only differ in their use of a P-controller and PID controller respectively. They both contain functionality to hold a given altitude and fly between vertical waypoints.

It has been shown that the abstracted models resemble the original DE model close enough for them to be used for prototyping. This was demonstrated by implementing vertical waypoint behaviour in all models and comparing their behaviour when flying to the three waypoints. Furthermore, the LOC required for all three implementations has been calculated to show the difference in implementation size. The results demonstrate that abstract models avoiding low level details can be used for prototyping purposes with reasonable results. This is interesting because it allows people with different expertises to work jointly on a project on different levels of abstraction, e.g. a software engineer working on high-level behaviour using models while a mechanical engineer works on low level constructs. The case study in this paper serves as an example of this, as it was implemented by a software engineer within six hours without particular knowledge of mechanics or electronics.

Finally the paper describes the effort of converting the CT and DE models to FMI compliant FMUs in order to be used for co-simulation in the INTO-CPS tool-chain along with a comparison of the results obtained with the two different technologies Crescendo and INTO-CPS.

8 Future work

The motivation of creating new DE models is to gain insight into developing control logic for a UAV. This insight will be made visible by documented models and data. The data will be gathered by utilizing DSE throughout the development process to various components and thereby choose optimal solutions. Part of this development will also result in building blocks in the shape of libraries for Overture, e.g. a vector library, and various generic components that can be used for other projects. An example of a generic component can be a feedback system such as a PID controller. During the development it will also be explored how to create the appropriate abstractions for testing and optimizing the constituent components. An example of creating the appropriate abstraction is related to a planned case study, which will investigate how to recover from battery failure. In this case low level details are important, whereas behaviour such as waypoint navigation is irrelevant.

The tools from the INTO-CPS project are expected to aid in the process of achieving optimal CT and DE models and ultimately to create a DE model detailed enough to be code generated and used to control a UAV. In this process alternative implementations will inevitably arise, and the choice of implementation will depend on results from experiments and case studies, e.g. using DSE. Thus it will contain functionality similar to the APM:Copter framework but with the difference that it will be documented. It is also the goal to enhance the CT model to a higher level of fidelity and thereby improve simulations.

Acknowledgments We would like to thank the anonymous referees for valuable input on this work. The work presented here is partially supported by the INTO-CPS project funded by the European Commission's Horizon 2020 programme under grant agreement number 664047.

References

1. Bouabdallah, S., Murrieri, P., Siegwart, R.: Design and control of an indoor micro quadrotor. In: Proceedings of the 2004 IEEE International Conference on Robotics & Automation. pp. 4393–4398. IEEE, New Orleans, LA (April 2004)
2. Broenink, J.F.: Modelling, Simulation and Analysis with 20-Sim. Journal A Special Issue CACSD 38(3), 22–25 (1997)
3. Fitzgerald, J.S., Larsen, P.G.: Balancing Insight and Effort: the Industrial Uptake of Formal Methods. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) Formal Methods and Hybrid Real-Time Systems, Essays in Honour of Dines Bjørner and Chaochen Zhou on the Occasion of Their 70th Birthdays. pp. 237–254. Springer, Lecture Notes in Computer Science, Volume 4700 (September 2007), iSBN 978-3-540-75220-2
4. Fitzgerald, J., Larsen, P.G.: Modelling Systems – Practical Tools and Techniques in Software Development. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, Second edn. (2009), ISBN 0-521-62348-0
5. Fitzgerald, J., Larsen, P.G., Verhoef, M. (eds.): Collaborative Design for Embedded Systems – Co-modelling and Co-simulation. Springer (2014), http://link.springer.com/book/10.1007/978-3-642-54118-6
6. Grujic, I., Nilsson, R.: Model-based development and evaluation of control for complex multi-domain systems: Attitude control for a quadrotor uav. Tech. Rep. 23, Department of Engineering, Aarhus University (January 2016)
7. Isasa, J.A.E., Jørgensen, P.W., Larsen, P.G.: Hardware In the Loop for VDM-Real Time Modelling of Embedded Systems. In: MODELSWARD 2014, Second International Conference on Model-Driven Engineering and Software Development (January 2014)
8. Jensen, J., Chang, D., Lee, E.: A Model-Based Design Methodology for Cyber-Physical Systems. In: Wireless Communications and Mobile Computing Conference (IWCMC), 2011 7th International. pp. 1666–1671 (2011)
9. Karsai, G., Sztipanovits, J., Ledeczi, A., Bapty, T.: Model-Integrated Development of Embedded Software. In: Proceedings of the IEEE. vol. 91, pp. 145–164 (2003)
10. Kramer, J.: Is Abstraction the Key to Computing? Communications of the ACM 50(4), 37–42 (2007)
11. Lausdahl, K., Coleman, J.W., Larsen, P.G.: Semantics of the VDM Real-Time Dialect. Tech. Rep. ECE-TR-13, Aarhus University (April 2013)

12. Lee, E.A.: Cyber Physical Systems: Design Challenges. Tech. Rep. UCB/EECS-2008-8, EECS Department, University of California, Berkeley (Jan 2008), http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-8.html
13. Overture-Core-Team: Overture Web site. http://www.overturetool.org (2007)
14. Thompson, H. (ed.): Cyber-Physical Systems: Uplifting Europe's Innovation Capacity. European Commission Unit A3 - DG CONNECT (December 2013)
15. Thule, C.: Verifying the Co-Simulation Orchestration Engine for INTO-CPS. In: Doctoral Symposium FM 2016. Limassol, Cyprus (November 2016)
16. Wing, J.M.: Computational thinking and thinking about computing. Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences 366(1881), 3717–3725 (2008)

Automated Generation of C# and .NET Code Contracts from VDM-SL Models

Steffen P. Diswal, Peter W. V. Tran-Jørgensen and Peter Gorm Larsen

Department of Engineering, Aarhus University, Finlandsgade 22, 8200 Aarhus N, Denmark

Abstract. Automatic code generation gives software engineers a convenient way to realise a VDM-SL specification in a programming language in order to achieve interoperability with standard libraries and external software modules. This paper presents a set of rules for translating VDM-SL to C#. The rules have been implemented in a proof-of-concept transcompiler as an extension to the Overture tool. The performances of the .NET Code Contracts library and the OpenJML tool are measured and compared, revealing a significant computational overhead in the latter.

Keywords: VDM-SL, C#, JML, .NET Code Contracts, code generation, Design-by-Contract

1 Introduction

Being a formal modelling language, VDM-SL differs significantly from low-level programming languages by focusing on high-level abstractions rather than pointers and details of the underlying hardware platform [8]. Combined with the Design-by-Contract (DbC) elements in VDM-SL, which enable automatic software verification, these language traits support the early analysis of software systems. However, eventually a VDM-SL model needs to be realised in a programming language. It is therefore of interest to investigate to what extent it is possible to automate the efforts needed to translate a VDM-SL specification into contract-aware code.

This paper is based on the master's thesis by the first author [7], which presents a translation of VDM-SL to C# programs with .NET Code Contracts. The translation covers a substantial part of the executable subset of VDM-SL, and the core of that is presented in this paper.

In [19], Jørgensen et al. present an approach to translating VDM-SL specifications to Java Modelling Language (JML) [4] annotated Java programs. Their translation is presented as rules that are implemented in Overture's Java code generator in order to make the approach fully automated. Similar to our work, the JML generator is developed using Overture's code generation platform [9]. However, our work is different in that we target different implementation technologies, i.e. C# and .NET Code Contracts. In particular, there are differences between JML and .NET Code Contracts which do not allow us to directly use the rules presented in [19]. The reason is that the DbC elements supported by JML and .NET Code Contracts differ. For example, in .NET Code Contracts, invariants are only checked at the end of public methods, whereas in JML, they

must hold in the pre- and post-states of every method, unless the method is declared as a JML helper. Furthermore, JML uses ghost variables to specify abstract state of a class – this construct does not exist in .NET Code Contracts.

Other noteworthy attempts have been made to represent VDM-SL's DbC constructs using contract-aware code. In [21], Visser et. al. present an approach that uses Haskell's monad construct to offer several modes of runtime evaluation. To name a few examples, the *error* mode raises exceptions when contracts are violated, whereas the *free fall* mode performs no checking of contracts at all. Free fall mode may be desired for production code when system performance is of high importance. One advantage of using monads to represent contracts is that function definitions do not need to be augmented with extra contract checks as this is being handled by monads.

In terms of modelling technologies other than VDM, attempts have been made to translate Event-B models [1] into contract-aware code. In [16], Rivera et. al. present the EventB2Java code generator, which translates both abstract and refinement Event-B models into JML annotated Java programs. Similar to our work, EventB2Java is fully automated and does not require any user intervention. In [6], Dalvandi et al. present an approach to the translation of Event-B specifications into Dafny code contracts [11]. While their code generator leaves the implementation of the Dafny classes to the user, our code generator also translates the explicit parts of function and operations (the bodies) into code.

The remaining part of this paper is structured as follows: Section 2 introduces the parts of C# and the .NET Code Contracts library necessary to understand the translation. Section 3 presents an extract of the rules used to generate C#/.NET Code Contracts from VDM-SL. Section 4 compares the performance of the .NET Code Contracts library to the OpenJML runtime-assertion checker, and finally, Section 5 concludes this paper and outlines future work.

2 C# and .NET Code Contracts

.NET is a software platform invented by Microsoft in 2001[1]. The core of .NET is the ISO- and Ecma-standardised Common Language Infrastructure (CLI) specification [15] (see Fig. 1). It defines a platform-independent assembly language, Common Intermediate Language (CIL), as well as the architecture of the corresponding runtime environment, Virtual Execution System (VES). Additionally, it defines a set of standard libraries that can be used by all CLI-compliant programs. The primary implementation of the CLI is the *.NET Framework*, which targets the Windows operating system family. In this context, the Common Language Runtime (CLR) is a virtual machine that implements VES.

Alongside the development of .NET, a development team at Microsoft lead by Anders Hejlsberg conceived the C# programming language [12]. Primarily influenced by C++ and Java, it is an object-oriented language that targets the .NET platform by compiling to CIL. Its syntax resembles that of Java to a large extent.

[1] See https://dot.net.

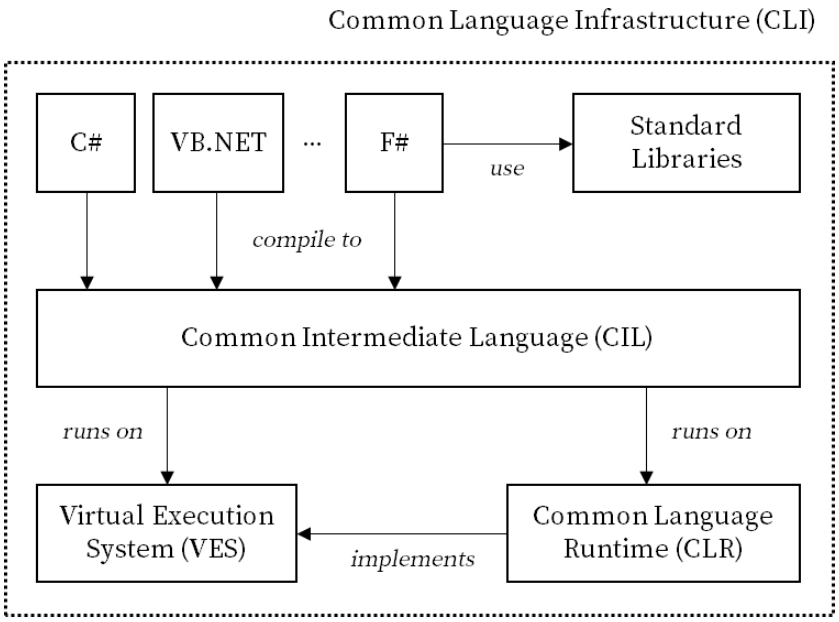


Fig. 1. The architecture of the Common Language Infrastructure in .NET.

3 Example Rules for Translation

3.1 Design-by-Contract

C# supports DbC through the .NET Code Contracts library [18, 17], which is a spin-off from the Spec# research project [2]. The .NET Code Contracts library resides under the `System.Diagnostics.Contracts` namespace and supports preconditions, postconditions, invariants as well as explicit assertions. When contracts are violated the library throws an exception. Contract enforcement is enabled by defining a symbol named `CONTRACTS_FULL` via the **`#define`** directive in C#.

Pre- and postconditions In .NET Code Contracts, preconditions are specified by calling the `Contract.Requires` method at the beginning of the method being guarded. Similarly, postconditions are specified by calling `Contract.Ensures`. They take a logical predicate as input and throw an instance of `ContractException` if the predicate evaluates to false when the C# method is entered and exited, respectively. The predicates must be referentially transparent and may only call methods tagged with the `PureAttribute` type – *pure methods* in .NET terminology – and read variables. Because VDM-SL treats the logical predicates in pre- and postconditions as self-contained Boolean functions [10], they are equivalent to pure **`bool`** methods in C# and may be called within contract predicates.

Within the predicate of a postcondition, it is also possible to refer to the guarded method's return value by calling `Contract.Result<TReturn>`, where `TReturn`

is the method's return type. Furthermore it is also possible to refer to the method's pre-state by calling `Contract.OldValue`, which takes an expression as input and returns the value of this expression at the time of entry into the method.

Translating preconditions for functions

Let f : X * Y * Z -> R be a VDM-SL function guarded by a precondition p, let e_{p} be the logical predicate of p, and let f_{pre} : X * Y * Z -> **bool** be the self-contained function for p in VDM-SL. Then f and f_{pre} become pure methods f' and f'_{pre} in C#. f' calls `Contract.Requires(`f'_{pre}`(`x`, `y`, `z`))` for parameters x, y and z. f'_{pre} evaluates and returns e_{p}.

Translating postconditions for functions

Let f : X * Y * Z -> R be a VDM-SL function guarded by a postcondition q, let e_{q} be the logical predicate of q, and let f_{post} : X * Y * Z * R -> **bool** be the self-contained function for q in VDM-SL. Then f and f_{post} become pure methods f' and f'_{post} in C#. f' calls `Contract.Ensures(`f'_{post}`(`x`, `y`, `z`, Contract.Result<`R'`>()))` for parameters x, y and z and return type R'. f'_{post} evaluates and returns e_{q}.

The VDM-SL specification of an Automated Teller Machine (ATM) has been used to demonstrate Overture's JML translator [19]. The same example is used here with focus on DbC elements to compare the translations to JML and C#. A VDM-SL operation such as:

```
operations
    AddCard: Card ==> ()
    AddCard(c) == validCards := validCards union {c}
    pre c not in set validCards
    post c in set validCards;
```

is in a C#/.NET context translated to:

```
public static void AddCard(Card c) {
    Contract.Requires(c != null);
    Contract.Requires(PreAddCard(c, State));
    Contract.Ensures(
        PostAddCard(c, Contract.OldValue(State), State));
    State.ValidCards.Add(c);
}

[Pure]
public static bool PreAddCard(Card c, St st) {
    Contract.Requires(c != null);
    Contract.Requires(st != null);
    return !st.ValidCards.Contains(c);
```

```
}

[Pure]
public static bool PostAddCard(Card c, St oldSt, St st) {
    Contract.Requires(c != null);
    Contract.Requires(oldSt != null);
    Contract.Requires(st != null);
    return st.ValidCards.Contains(c);
}
```

Invariants Invariants in .NET Code Contracts are specified by calling `Contract.Invariant` [13]. Since invariants protect the state of the program rather than the input and output of methods, they are treated a bit differently than pre- and postconditions in .NET Code Contracts. All calls to `Contract.Invariant` must reside in a special helper method that takes no parameters, returns **`void`** and is tagged with the `ContractInvariantMethodAttribute` type, which is an attribute provided by .NET Code Contracts. The helper method is specific to the C# class and is not defined in VDM-SL. Traditionally, it is named `ObjectInvariant`.

Invariants in .NET Code Contracts are only checked upon leaving any public method in contrast to VDM-SL where they must hold at all times. An invariant that has been satisfied so far can only be violated by manipulating the program state it is guarding. Therefore, in practice, it is only necessary to check invariants after program state mutations. By encapsulating all program state in C# properties, .NET Code Contracts will check the invariants after invoking a public property setter, as this is equivalent to calling a public setter method. Upon class instantiation, the class invariants are also checked after calling the public constructor.

Translating invariants

Let i be an invariant for type T, let e_{i} be the logical predicate of i, and let T_{inv} **:** T `->` **`bool`** be the self-contained function for i in VDM-SL. Then T becomes an appropriate type T' in C# and T_{inv} becomes a member of T' as the pure method T'_{inv}. The special `ObjectInvariant` helper method of T' calls `Contract.Invariant(`T'_{inv}`(this))`. T'_{inv} evaluates and returns e_{i}.

Type invariants Type invariants must be enforced in five places: parameters to functions and operations, result values from functions and operations, variable initialisers, assignments and value definitions [19].

Type invariants for method parameters are enforced as preconditions, that is, by calling `Contract.Requires` and checking that the type invariant predicate is satisfied. Type invariants for return values are enforced similarly as postconditions via `Contract.Ensures`.

All persistent state is stored in properties which are subject to invariant enforcement through the calls to `Contract.Invariant` carried out by the nested class

declarations. Temporary state in VDM-SL can be defined through the use of variables initialised in let-expressions and define-expressions. They are equivalent to local variable declarations in C#. Potential type invariants are enforced by calling the `Contract.Assert` method in .NET Code Contracts. It takes a predicate as input and evaluates it immediately.

VDM-SL values are translated to static read-only properties in C# that are initialised upon declaration. Therefore, type invariants on values only need to be enforced once. This is done by calling `Contract.Assert` from a static constructor in the class that contains the property. The property initialiser is always executed just before the static constructor [12]. So for a small VDM-SL extract that uses a type invariant such as:

```
types
    Pin = nat
    inv p == 0 <= p and p <= 9999;
```

the following C# code is produced:

```
public sealed class Pin : ICopyable<Pin>, IEquatable<Pin> {
    public int Value { get; }

    public Pin(int @value) { Value = @value; }

    [ContractInvariantMethod]
    private void ObjectInvariant() {
        Contract.Invariant(Value >= 0);
        Contract.Invariant(InvPin(Value));
    }

    [Pure]
    public static bool InvPin(int p) {
        Contract.Requires(p >= 0);
        return 0 <= p && p <= 9999;
    }
    // Equals, GetHashCode etc. have been omitted.
}
```

3.2 Collections

The .NET Standard Library provides the `ISet<T>`, `IList<T>` and `IDictionary<TDomain, TRange>` collection interfaces that correspond to sets, sequences and maps in VDM-SL [14]. Furthermore, the LINQ library defines extension methods to manipulate generic collections of type `IEnumerable<T>` (the base type of all collection types in .NET) which enable straightforward translation of most of VDM-SL's collection operations. For example, small VDM-SL expressions such as:

```
dunion m
iota i in set m & p
tl m
inverse m
```

are translated to the following LINQ queries in C#:

```
m.Cast<IEnumerable<T>>().Aggregate(Enumerable.Union).ToHashSet()
m.Single(i => p)
m.Skip(1).ToList()
m.ToDictionary(_ => _.Value, _ => _.Key)
```

Set comprehensions While C# does not offer a dedicated set comprehension construct like VDM-SL, it provides built-in language support for LINQ queries. A LINQ query is initiated with a **from**...**in** clause that specifies the collection to use as a starting point. The **from**...**in** clause is followed by an optional **where** clause, which filters the elements in the source collection, and a mandatory **select** clause, which projects the filtered elements into a new collection instance. Compound **from**...**in** clauses are useful for manipulating multiple collections in a single query. Hence, LINQ queries in C# emulate set comprehensions in VDM-SL.

Translating set comprehensions

Let C be a set comprehension with bindings $b_1, ..., b_n$, predicate expression p and projection expression e. Let the binding b_i be on the form s_i **in set** S_i in VDM-SL. Then C becomes a LINQ query C' in C# constituted by `(`**from** s_1 **in** S_1 `...` **from** s_n **in** S_n **where** p **select** e`).ToHashSet()`. If the set comprehension leaves out the predicate p, the **where** clause is omitted in the LINQ query.

A set comprehension that quantifies over multiple variables such as:

```
{mk_(x, y, z) | x, y, z in set {1, ..., 5} & x + y = z}
```

leads to the following rather compact LINQ query in C#:

```
(from x in Enumerable.Range(1, 5)
 from y in Enumerable.Range(1, 5)
 from z in Enumerable.Range(1, 5)
 where x + y == z
 select Tuple.Create(x, y, z)).ToHashSet()
```

Universal quantifications In addition to occurring in set comprehensions, set bindings also occur in quantified expressions such as universal quantifications, which check

whether a predicate holds for every element in a set. The translation of bindings in universal quantifications is similar to the one for set comprehensions: a **`from`**...**`in`** clause, compound if necessary. The **`select`** clause projects the elements of the bindings into Boolean values computed from the predicate. LINQ provides the `All` method for testing whether a predicate is satisfied for all elements in a single collection. In this case, it simply tests that all Boolean values are true.

Translating universal quantifications
Let Q be a universally quantified expression with bindings $b_1, ..., b_n$ and predicate expression p. Let the binding b_i be on the form s_i **`in set`** S_i in VDM-SL. Then Q becomes a LINQ query Q' in C# constituted by `(`**`from`** s_1 **`in`** S_1 **`...`** **`from`** s_n **`in`** S_n **`select`** p`).All(x => x)`, where `x => x` is a C# lambda expression for the Boolean identity function.

So for a small quantified expression in VDM-SL such as:

```
forall n in set s & n <= r
```

one gets the following LINQ query in C#:

```
(from n in s select n <= r).All(x => x)
```

3.3 Type aliases

A type alias that defines a type invariant is realised in C# by a wrapper class that contains a single property that holds the value of the underlying type subject to aliasing. By default, classes in C# exhibit referential equality and pass-by-reference semantics [12] so that it is just the object references, not the objects themselves, that are copied when they are passed around. This behaviour can be changed by overriding the `Equals` and `GetHashCode` methods to provide structural equality and implementing a `Copy` method to enable pass-by-value semantics like VDM-SL. The solutions proposed by Joshua Bloch [3] for overriding the `equals` and `hashCode` methods in corresponding Java classes have been adapted to C#.

Different wrapper classes of the same type are not interchangeable in C#, although this feature is achievable through type casting. In C#, both the explicit and implicit type cast operators can be overloaded to increase the number of valid type casts [12]. However, the class needs to overload the type cast operators for every other type alias that has been defined for the same underlying type in order to make them interchangeable. This leads to a combinatorial explosion of overloads, which is not desirable. It suffices to overload the type cast operators to and from the underlying type so that literals, for example integers, are automatically cast to instances of the class. By leaving the property public, it is always possible to retrieve the aliased value.

Translating type aliases

Let U be an alias for type T. Then T becomes an appropriate type T' in C# and U becomes a class U' that contains a single `Value` property of type T'. `Value` is initialised from a public constructor. U' overrides the `Equals` and `GetHashCode` methods to provide structural equality on the `Value` property. It also implements a `Copy` method that creates a deep copy of an instance of U'. Finally, it overloads the type cast operators from T' to U' and vice versa – the former by instantiating U' from an instance of T'; the latter by retrieving the wrapped instance of T' stored in the `Value` property in U'.

As an example of how to translate a VDM-SL type alias consider the example below:

```
types
    Pin = nat
    inv p == 0 <= p and p <= 9999;
```

For this example the following C# code is produced:

```
public sealed class Pin : ICopyable<Pin>, IEquatable<Pin> {
    public int Value { get; }

    public Pin(int @value) { Value = @value; }

    public static implicit operator Pin(int that)
        => new Pin(that);

    public static implicit operator int(Pin that)
        => that.Value;

    public Pin Copy() => new Pin(Value);

    public bool Equals(Pin that) {
        if (ReferenceEquals(null, that)) return false;
        if (ReferenceEquals(this, that)) return true;
        return Value == that.Value;
    }

    public override bool Equals(object that) {
        if (ReferenceEquals(this, that)) return true;
        return that is Pin && Equals((Pin) that);
    }

    public override int GetHashCode() {
        var result = 17;
        result = 31 * result + Value;
        return result;
    }
```

```
    // ObjectInvariant and InvPin have been omitted.
}
```

4 Comparison of .NET Code Contracts and OpenJML

This section introduces a VDM-SL specification that is used to analyse the performance of .NET Code Contracts and OpenJML. The VDM-SL specification models an algorithm that is used to obfuscate financial accounting district (FAD) codes – a six-digit number that identifies a retail branch. The FAD code example was originally used in [20] to analyse the performance of code generated traces executed using OpenJML. The obfuscation algorithm defined in the convert function, shown in Listing 1.1, takes a mapping of digits as input, and computes a permutation of all FAD codes so that no FAD code is mapped to itself.

```
values
    SIZE = 6;
    MAX = 10 ** SIZE - 1;
    DM1 : DigitMap =
        {1 |-> 9, 2 |-> 8, 3 |-> 7, 4 |-> 6, 5 |-> 0,
         6 |-> 4, 7 |-> 3, 8 |-> 2, 9 |-> 1, 0 |-> 5};
types
    DigitMap = inmap nat to nat
    inv m ==
        let digits = {0, ..., 9} in
            dom m = digits and rng m = digits
            and forall c in set dom m & m(c) <> c;

    FAD = nat
    inv f == f <= MAX
functions
    convert: FAD * DigitMap -> FAD
    convert(fad, dm) ==
        let digits = digitsOf(fad) in
            valOf([dm(digits(i)) | i in set inds digits])
    post RESULT <> fad;
```

Listing 1.1. The FAD model

The corresponding C# code is:

```
public static int Size { get; } = 6;

public static int Max { get; } = 10.IntPower(Size) - 1;

public static DigitMap Dm1 { get; } = new Dictionary<int, int>
{
    [1] = 9, [2] = 8, [3] = 7, [4] = 6, [5] = 0,
```

```
    [6] = 4, [7] = 3, [8] = 2, [9] = 1, [0] = 5
};

[Pure]
public static Fad Convert(Fad fad, DigitMap dm) {
    Contract.Requires(fad != null);
    Contract.Requires(dm != null);
    Contract.Ensures(Contract.Result<Fad>() != null);
    Contract.Ensures(
        PostConvert(fad, dm, Contract.Result<Fad>()));

    return Let(() => {
        var digits = DigitsOf(fad);
        return ValOf(
            (from i in Enumerable.Range(1, digits.Count)
                                 .ToHashSet()
             select dm.Value[digits[i - 1]]).ToList());
    });
}

[Pure]
public static bool PostConvert(Fad fad, DigitMap dm,
                               Fad result) {
    Contract.Requires(fad != null);
    Contract.Requires(dm != null);
    Contract.Requires(result != null);
    return !Equals(result, fad);
}

public sealed class DigitMap : ICopyable<DigitMap>,
                               IEquatable<DigitMap> {
    public DigitMap(Dictionary<int, int> @value) {
        Value = @value;
    }

    public Dictionary<int, int> Value { get; }

    [ContractInvariantMethod]
    private void ObjectInvariant() {
        Contract.Invariant(Value != null
            && Value.IsInjective()
            && Contract.ForAll(Value.Keys, _ => _ >= 0)
            && Contract.ForAll(Value.Values, _ => _ >= 0));
        Contract.Invariant(InvDigitMap(Value));
    }

    [Pure]
    public static bool InvDigitMap(Dictionary<int, int> m) {
        Contract.Requires(m != null && m.IsInjective()
            && Contract.ForAll(m.Keys, _ => _ >= 0)
```

```
            && Contract.ForAll(m.Values, _ => _ >= 0));

        return Let(() => {
            var digits = Enumerable.Range(0, 10).ToHashSet();
            return m.Keys.ToHashSet().SetEquals(digits)
                && m.Values.ToHashSet().SetEquals(digits)
                && (from c in m.Keys.ToHashSet()
                    select m[c] != c).All(_ => _);
        });
    }
    // Equals, GetHashCode etc. have been omitted.
}

public sealed class Fad : ICopyable<Fad>, IEquatable<Fad> {
    public Fad(int @value) { Value = @value; }

    public int Value { get; }

    [ContractInvariantMethod]
    private void ObjectInvariant() {
        Contract.Invariant(Value >= 0);
        Contract.Invariant(InvFad(Value));
    }

    [Pure]
    public static bool InvFad(int f) {
        Contract.Requires(f >= 0);
        return f <= Max;
    }
    // Equals, GetHashCode etc. have been omitted.
}
```

An exhaustive test checks that the `convert` function maps every FAD code to another FAD code. When the value of `SIZE` is 6, there are one million FAD codes to check.

This section carries out the exhaustive test in order to measure the computational performance of both the output from the C# and Java code generators. In C#, DbC elements are implemented and checked by .NET Code Contracts [17], whereas they are represented by JML annotations in Java, which can be checked using the OpenJML runtime-assertion checker [5].

The benchmark program runs the exhaustive test eight times and reports the average time spent per iteration. The first iteration is a warm-up iteration for the virtual machines – .NET CLR and Java HotSpot VM, respectively – and is not included in the result. The return value of `convert` is stored in a field variable whose final value is printed to the console in order to avoid the virtual machine performing dead code elimination[2]. Three experiments are carried out for each platform:

[2] See http://www.oracle.com/technetwork/java/hotspotfaq-138619.html.

Experiment I No contracts are checked. This corresponds to removing all DbC elements from the VDM-SL specification and executing the `convert` function. In C#, the definition of the `CONTRACTS_FULL` symbol is omitted to disregard all contracts. In Java, the program is compiled with the normal Java compiler in OpenJDK 7, which disregards all JML annotation comments. Thus, no contracts are emitted in the bytecode.

Experiment II Contracts are specified, but not checked during execution. This is the default mode of operation for production-ready software. In .NET Code Contracts, the contracts can be disabled via the Visual Studio extension. In Java, they can be disabled by omitting the '-ea' (enable assertions) command line argument. The program is compiled with the OpenJML compiler.

Experiment III Contracts are specified and checked during execution. This is the approach employed for software under development.

4.1 Results

Table 4.1 shows the results of the benchmarking experiments. The numbers denote the time spent executing a particular experiment[3]. They have been performed with `SIZE` values of 1 to 6, cf. Listing 1.1, yielding 10^{SIZE} iterations in an exhaustive test.

Size	.NET I [ms]	.NET II [ms]	.NET III [ms]	Java I [ms]	Java II [ms]	Java III [ms]
1	1	1	1	1	2	2
2	1	1	1	2	20	22
3	1	1	1	4	245	254
4	15	15	23	22	3,103	3,212
5	190	189	295	212	37,626	38,401
6	2,273	2,279	3,610	2,498	440,716	443,523

Table 1. Benchmark results.

Experiment I and II in .NET show that there is no notable difference between omitting the `CONTRACTS_FULL` symbol and configuring the Visual Studio extension to disable contracts at runtime. In fact, when contracts are disabled in Visual Studio, they are erased from the bytecode, just like omitting the `CONTRACTS_FULL` symbol would do. They complete the exhaustive test of a million iterations in about 2,300 milliseconds. In Experiment III, the contracts are emitted in the CIL bytecode and checked during

[3] Benchmarking has been performed on an HP desktop computer that runs a 64-bit edition of Windows 10 and is equipped with a 3.4 GHz Intel i7 Skylake processor and 16 GB RAM. The C# benchmarks have been run on .NET CLR 4.0, whereas the JML benchmarks have been run on Java HotSpot VM 1.8 after being built on OpenJDK 7 via an Ubuntu 14 virtual machine.

execution, which results in a performance overhead. It completes the exhaustive test in about 3,600 milliseconds, implying an overhead from code contracts of approximately 60%.

In Java, Experiment I performs about as well as .NET, completing the exhaustive test in about 2,500 milliseconds. However, Experiment II and III, which are compiled with OpenJML, suffer from a very large overhead. They complete the exhaustive test in about 440,700 and 443,500 milliseconds, respectively, making them approximately 175 times slower than Experiment I. This overhead has to be caused by the OpenJML compiler, otherwise the performance of Experiment II would be comparable to Experiment I, which it is clearly not. When assertions are disabled in Java, they should have no impact on the execution. This confirms the OpenJML performance results reported in [20].

As shown in Experiment III, when code contracts enabled, .NET Code Contracts is about 120 times faster than OpenJML. Fig. 4.1 shows the results in a logarithmically scaled graph.

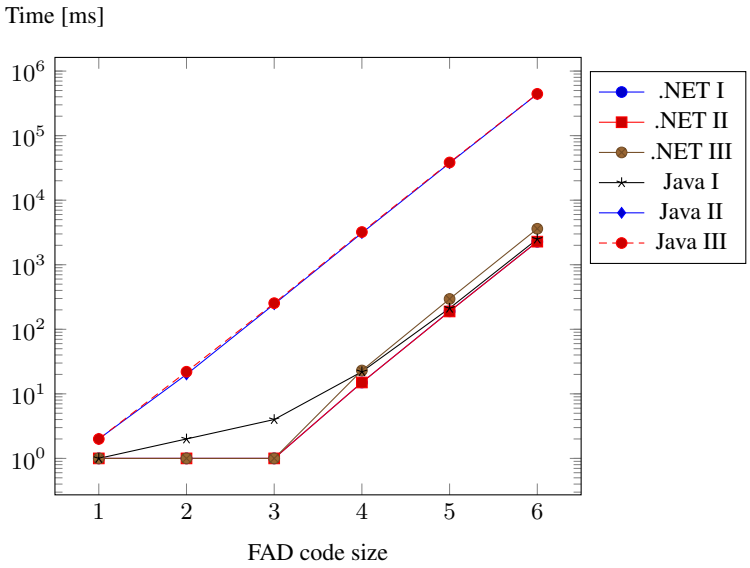


Fig. 2. Benchmark results visualised in a logarithmic data plot.

4.2 Validity of results

When inspecting the benchmark results, it should be taken into consideration that .NET CLR and Java HotSpot VM are two different environments and the Java and C# code generators use different approaches in some situations. For example, the C# code generator favours native language constructs and standard library methods over manually implemented utility methods like those shipped with the Java code generator runtime.

They also differ in how DbC elements are enforced: Type invariants such as bounds- and null-checking are implemented in C# as pre- and postconditions by .NET Code Contracts when they occur in method parameters and return values. The Java code generator, on the other hand, enforces the type invariants through JML assertions. Furthermore, type aliases are inlined by the Java code generator, but defined as separate wrapper classes by the C# code generator. Still, the difference in performance between .NET Code Contracts and OpenJML is so significant that it cannot be attributed to platform differences alone.

5 Conclusion and Future Work

In this paper, we have presented a handful of rules for translating core VDM-SL concepts to C# by utilising .NET Code Contracts and LINQ. They have been implemented in a transcompiler prototype for the Overture tool.

.NET Code Contracts performs significantly better than the OpenJML tool in the FAD code obfuscation model, which covers all kinds of DbC elements in VDM-SL. The benchmark results reveal that .NET Code Contracts is about 120 times faster than OpenJML, thus confirming the results of Jørgensen et al. [20] on the large overhead caused by OpenJML. Its impact is further demonstrated by the significant time gap between Java Experiment I and II.

The translation rules and transcompiler prototype targets only a subset of VDM-SL, leaving plenty of room for enhancements. For example, pattern matching and decomposition of tuples, records, sets etc. in VDM-SL are yet to be translated to C#. Basic object decomposition is to be introduced as a native feature of C# 7.0 [4].

Another extension is to support the object-oriented VDM++ dialect [10] and handle the translation of object aliasing, method overloading, multiple class inheritance and concurrency.

Acknowledgements We would like to thank the anonymous referees for valuable input on this work. The authors would also like to thank Aslan Askarov and Erik Ernst for input on this work.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, New York, NY, USA, 1st edn. (2010)
2. Barnett, M., Leino, R.M., Schulte, W.: The Spec# Programming System: An Overview. In: CASSIS Proceedings (October 2004)
3. Bloch, J.: Effective Java. Addison-Wesley, second edn. (2008)
4. Burdy, L., Cheon, Y., Cok, D., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML Tools and Applications. Intl. Journal of Software Tools for Technology Transfer 7, 212–232 (2005)

[4] See https://blogs.msdn.microsoft.com/dotnet/2016/08/24/whats-new-in-csharp-7-0/.

5. Cok, D.R.: OpenJML: JML for Java 7 by Extending OpenJDK. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods, Lecture Notes in Computer Science, vol. 6617, pp. 472–479. Springer Berlin Heidelberg (2011), http://dx.doi.org/10.1007/978-3-642-20398-5_35
6. Dalvandi, M., Butler, M., Rezazadeh, A.: From Event-B Models to Dafny Code Contracts, pp. 308–315. Springer International Publishing, Cham (2015)
7. Diswal, S.P.: Transcompilation of VDM-SL to C#. Master's thesis, Aarhus University, Department of Computer Science (June 2016)
8. Fitzgerald, J., Larsen, P.G.: Modelling Systems – Practical Tools and Techniques in Software Development. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, Second edn. (2009), ISBN 0-521-62348-0
9. Jørgensen, P.W.V., Larsen, M., Couto, L.D.: A Code Generation Platform for VDM. In: Battle, N., Fitzgerald, J. (eds.) Proceedings of the 12th Overture Workshop. School of Computing Science, Newcastle University, UK, Technical Report CS-TR-1446 (January 2015)
10. Larsen, P.G., Lausdahl, K., Battle, N., Fitzgerald, J., Wolff, S., Sahara, S., Verhoef, M., Tran-Jørgensen, P.W.V., Oda, T.: VDM-10 Language Manual. Tech. Rep. TR-001, The Overture Initiative, www.overturetool.org (April 2013)
11. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: International Conference on Logic for Programming Artificial Intelligence and Reasoning. pp. 348–370. Springer (2010)
12. Microsoft: C# Language Specification 5.0 (2013)
13. Microsoft: Code Contracts User Manual. https://www.microsoft.com/en-us/research/project/code-contracts/ (2013), accessed Sep 1st 2016
14. Microsoft: Language-Integrated Query (LINQ) (C#). http://msdn.microsoft.com/da-dk/library/mt693024.aspx (2015)
15. Petzold, C.: .NET Book Zero (2007)
16. Rivera, V., Cataño, N., Wahls, T., Rueda, C.: Code generation for Event-B. International Journal on Software Tools for Technology Transfer pp. 1–22 (2015)
17. Skeet, J.: C# in Depth, Second Edition. Manning Publications (2010)
18. Strauss, D.: C# Code Contracts Succinctly. Syncfusion Inc. (2016)
19. Tran-Jørgensen, P.W.V., Larsen, P.G., Leavens, G.T.: Automated translation of VDM to JML annotated Java (January 2016 Submitted to the International Journal on Software Tools for Technology Transfer (STTT))
20. Tran-Jørgensen, P.W., Larsen, P.G., Battle, N.: Using JML-based Code Generation to Enhance the Test Automation for VDM Models. In: The 14th Overture Workshop: Towards Analytical Tool Chains. pp. 79–93. Aarhus University, Department of Engineering, Cyprus (November 2016), ECE-TR-28
21. Visser, J., Oliveira, J.N.F., Barbosa, L., Ferreira, J.F., Mendes, A.: CAMILA revival: VDM meets Haskell. In: 1st Overture Workshop. University of Newcastle TR series (2005)

Automated VDM-SL to Smalltalk Code Generators for Exploratory Modeling

Tomohiro Oda[1,2], Keijiro Araki[2], and Peter Gorm Larsen[3]

[1] Software Research Associates, Inc. (tomohiro@sra.co.jp)
[2] Kyushu University (araki@ait.kyushu-u.ac.jp)
[3] Aarhus University, Department of Engineering, (pgl@eng.au.dk)

Abstract. Different dialects in the VDM-family have executable subsets. Simulated execution of a specification, either by an interpreter or a code generator, is an effective technique not only to produce production source code but also to validate the specification. This paper describes requirements on code generators for earlier stages of formal specification, a phase called exploratory modeling, and introduces an implementation in ViennaTalk. Performance, readability and liveness of the generated code are also evaluated.

1 Introduction

Formal methods are mostly used to precisely state the desired properties of a system to be engineered. In general, such formal specifications are not necessarily executable [6]. However, in order to communicate the insight of a formal specification it may make sense to express it in an executable subset in order to be able to animate its conceptual behaviour to stakeholders with little understanding of the formal notation that has been used [5, 1]. Interpretation of subsets of the different VDM dialects have been enabled for many years [12, 13]. Different tools exist for the different VDM dialects and here the most widely used ones are Overture [11] and VDMTools [10]. However, there is a balance between the time spent on making the specification executable and the insight gained so the time used to enable exploratory modelling needs to be properly balanced [3]. Different means of enabling executability exist so in this paper pros and cons for alternative solutions are considered. The primary focus in early exploration involves interpretation of executable subsets of VDM models but this also involves integration with legacy or graphical user interface code [4, 14]. Another alternative here is using code generation [7]. Target languages of code generators include those compiled into native binary code, those compiled into bytecode that runs on virtual machines and those executed by interpreters.

The rest of this paper is organised as follows: Section 2 provides a brief overview of exploratory modelling in general and ViennaTalk specifically. Afterwards, Section 3 provides a brief introduction to the code generation functionality from both Overture and VDMTools, followed by general requirements for code generation functionality seen from an explorative modelling perspective. Then Section 4 presents the core results in this paper indicating how the code generation features in the ViennaTalk solution can support the code generation requirements for exploratory modelling. Section 5 then

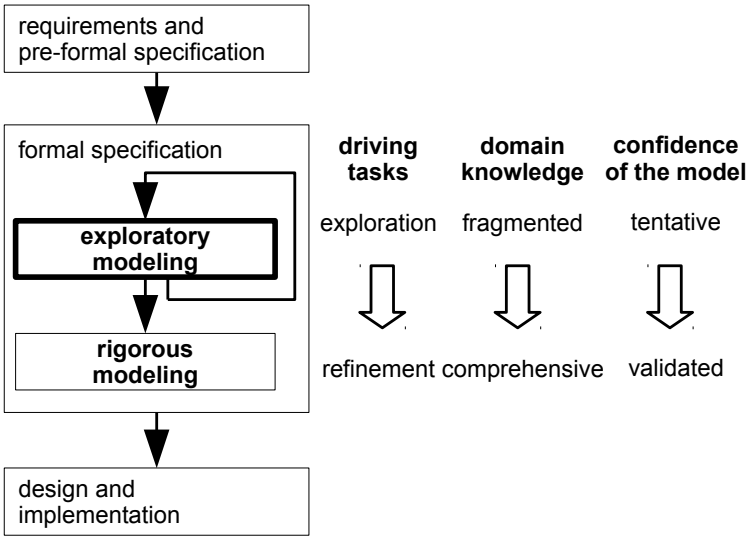


Fig. 1. Exploratory modeling in the different development phases

evaluates and compares the different code generators and their performance. Finally, Section 6 concludes the paper and points out future work possibilities.

2 Exploratory Modeling and ViennaTalk

There is no complete formal specification at the beginning of any development. Formal methods engineers often have limited and fragmented domain knowledge at the earlier stages of the specification phase. The formal methods engineers learn the nature of the target domain by writing tentative models of the system to be developed. The formal methods engineers' confidence in the model increases as the model is validated by stakeholders. The model gains in maturity and rigour when it is passed to the design and implementation phase. We call the earlier stages of the formal modeling "exploratory modeling"[15, 17, 16] (see Figure 1). The exploratory modeling involves learning efforts and validation by domain experts to find an appropriate abstraction of the problem domain. The exploratory modeling is followed by the rigorous modeling to make the model precise, concise and assured.

ViennaTalk [16] is a meta-IDE to develop tools for exploratory modeling in VDM-SL built on top of Pharo Smalltalk [2]. Figure 2 shows the configuration of ViennaTalk and external libraries of the Pharo system.

ViennaTalk is equipped with packages for interpreter wrappers called ViennaTalk-Engine, runtime support packages for VDM-SL such as ViennaTalk-Type and ViennaTalk-Value, parsers and accompanying source code analysis tools packaged as ViennaTalk-Parsers and the animation manager called VDMC. Three prototyping tools are built and bundled in ViennaTalk.

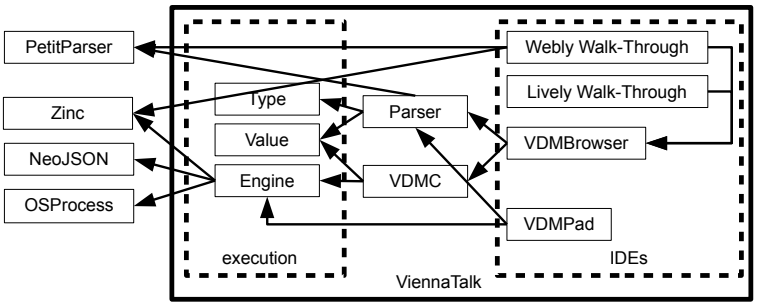


Fig. 2. Configuration of ViennaTalk

One of these is VDMBrowser [17], which has syntax highlighting, pretty printer and a live animation interface named Workspace. VDMBrowser can be used as a development tool, and also can be embedded into other development tools. The second is a UI prototyping environment named Lively Walk-Through [17], which enables VDM-SL engineers and UI designers to build a prototype with a VDM-SL executable specification and a prototypical user interface. VDM-SL engineers can confirm that the VDM-SL specification provides enough functionality to drive the user interface. UI designers can confirm that the UI is designed with appropriate assumptions on the system's functional model. The VDM-SL engineers and the UI designers can let domain experts, who may not have an engineering background, "test-drive" the UI prototype in order to validate the VDM-SL specification and the UI design. The third package is a Web API prototyping environment called Webly Walk-Through [17]. Operations and functions exported in a VDM-SL specification can be published as a Web API so that client programs can be accessed via HTTP. The fourth is VDMPad, a lightweight WebIDE for VDM-SL.

These tools are designed to support exploratory modeling in VDM-SL. A common feature among those tools is live animation that gives flexibility required by exploratory modeling. Liveness of a programming language is an ability to modify a running program without aborting the program and to continue the execution of the program with the modified code. Live animation on ViennaTalk enables the user to evaluate expressions as well as to modify the specification while the animation is running, so that the user can more deeply understand what's going on in the animation and try more alternative definitions. Flexibility is the most important aspect of support tools for exploratory modeling.

3 Requirements on Code Generators for Exploratory Modeling

VDMTools is equipped with C++ and Java source code generators. The code generators of VDMTools are reliable and widely applied in industry. The Overture tool also provides a Java generator [8] and also recently, a C generator.

Those generators are developed for the final stage of the formal specification phase: to generate production code. Java, C and C++ require development tools independent

of the VDM development environment. Use of those external tools may disturb the user because the user has to come in and out of the Integrated Development Environment (IDE) although IDEs are designed to host all activities in an integrated manner. Code generators for exploratory modeling should address those issues in order to support the user whose mental focus is not on the production of code, but on the modeling task.

In this section, requirements on code generators for exploratory modeling are listed. The requirements R1-a through R1-d are requirements specific to code generators. The requirements R1, R2 and R3 are general requirements on support tools for exploratory modeling.

R1 Direct interaction with smaller models.

Interaction between the user and the model at hand plays an important role in exploratory modeling. Direct interaction is required to support tools for exploratory modeling in general. In case of code generators, this general requirement can be broken down to specific requirements R1-a through R1-d.

R1-a Automatic compilation and execution of the generated source code.

Because the user's focus is on modeling, requiring the user to modify the generated source code should be avoided.

R1-b Compilation and execution of the generated source code must take place in the same IDE as the exploratory modeling.

IDEs are designed to support the user in an integrated manner. Switching between IDEs may diminish benefits of IDEs.

R1-c Few limitations on the specification language for automated code generation.

Exploratory modeling is carried out at earlier stages of the specification phase. The model is therefore expected to be abstract. The code generator for exploratory modeling should be able to generate executable code from as abstract a specification as possible.

R1-d Debug capability must be enabled.

Because the source model is still tentative and not sufficiently rigorous, it may be error prone. In order to debug the generated code, it is desirable that the generated code can be debugged in the way that the hand-written source code in the target language is usually debugged. The generated code should be human readable and can be easily modified.

R2 Understandable by stakeholders with no formal methods background.

In exploratory modeling, it is useful to ask domain experts and get feedback from the model. Code generators also should be suitable for communicating with stakeholders with no programming background. It is therefore desirable that GUI construction tools or visualisation techniques are available in the target language.

R3 Permissive checking by choice.

Exploratory modeling handles tentative, immature, often inaccurate models. Rejecting all suspicious models is not always productive. It is desirable that a code generator provides options to turn on or off static type checking, runtime type checking and runtime assertion testing.

R4 Continuous analysis.

Generated code can be unit tested efficiently. It is desirable that the generated code can be continuously tested by test frameworks. This requirement mainly affects choices of target languages.

4 Code Generators in ViennaTalk

ViennaTalk can automatically generate 3 kinds of Smalltalk programs, namely classes, objects of anonymous classes and scripts. The user of ViennaTalk can use 3 source code generators in three different ways: in the VDMBrowser, in the Live translator and by embedding VDM-SL expressions/statements inside Smalltalk code.

4.1 Design Rationale

We listed the design requirements based on the requirements on code generators for exploratory modeling **R1-a** through **R4**. Using the Smalltalk environment automatically satisfies requirements **R1-a** and **R1-b**. Requirement **R2** and **R4** can not be realised by code generators alone, but depend on other development tools. However, it should be noted that the generated code should be modifiable to combine with such development tools. The resulting design rationales are listed below.

- The code generator should cover as large a subset of VDM-SL as possible. This is requirement **R1-c**.
- The code generator should be able to turn on and off the runtime checking of types and assertions. This is requirement **R3**.
- The generated code should be as much like hand-written source code as possible. This is for requirements **R1-d**, **R2** and **R4**.
- The generated code should be traceable to the original VDM-SL source. This is for requirements **R1-d**, **R2** and **R4**.
- The generated code uses the standard Smalltalk library and extends its classes if necessary. New classes to implement VDM values should be minimised. This comes from requirement **R1-d**.

There are essential differences in the type systems between VDM-SL and Smalltalk and this makes naive mappings from VDM-SL types to Smalltalk classes infeasible. In VDM-SL, one value may belong to many types, i.e. 1 is a value of `nat`, `nat1`, `int`, `int inv` x == x < 10, `[nat]`, `nat | char` and so on. In Smalltalk, one object must have only one class. It is therefore not possible to define a one-to-one mapping between VDM-SL types and Smalltalk classes. A more flexible mapping between type objects and value objects is needed. Thus we made the following design decision.

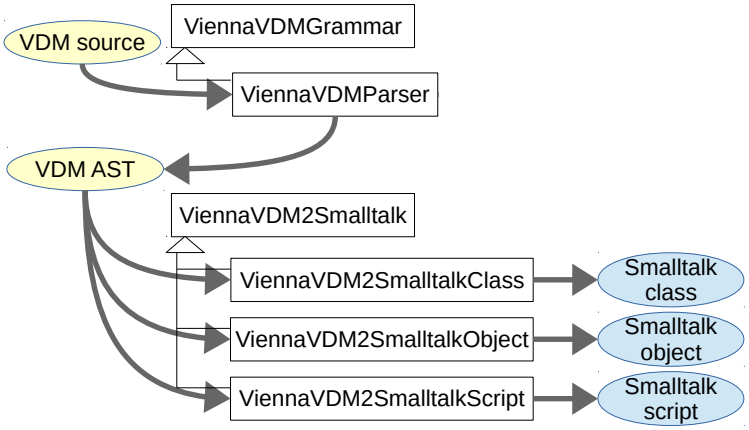


Fig. 3. Parsers and AST visitors to generate Smalltalk artefacts from VDM-SL source

– Types are not classes. A VDM type is an instance independent of classes of its values.

4.2 Implementation

Figure 3 illustrates the classes associated with parsing and code generation. ViennaTalk uses PetitParser[9] to implement a parser for VDM-SL. PetitParser is a combinatory parser library where a parser object can be synthesised from one or more parser objects. The framework provided by PetitParser allows to separate grammar definitions and corresponding outputs. ViennaVDMGrammar defines the grammar of VDM-SL which only accepts valid VDM-SL sources but does not produce any output. Subclasses of ViennaVDMGrammar defines outputs corresponding to each grammatical construct defined by the ViennaVDMGrammar class. ViennaVDMGrammar has a subclass named VDMParser which outputs an abstract syntax tree from the given VDM-SL source. The ViennaVDM2Smalltalk class is an abstract class that traverses over a given abstract syntax tree and produces Smalltalk code in the form that each of its three concrete classes defines.

The ViennaVDM2SmalltalkClass class generates one document class that represents the whole specification and module classes each of which implements the corresponding VDM-SL module. If the specification is flat, only one class for the specification will be generated. The users of ViennaTalk does not have to operate any external development tools. The generated classes can be browsed using Smalltalk's conventional class browsers, and are already compiled. The ViennaVDM2SmalltalkClass class is designed to be used in the transition from a VDM-SL specification to a Smalltalk implementation.

The ViennaVDM2SmalltalkObject class generates the same classes as those generated by the ViennaVDM2SmalltalkClasses class except that the anonymous classes do

not have global names. Because these anonymous classes do not appear in the global name space, the ViennaVDM2SmalltalkObject class leaves less footprints than the ViennaVDM2SmalltalkClass. The ViennaVDM2SmalltalkObject is designed to test specifications faster than using interpreters, either in automated test frameworks or in exploratory testing by domain experts.

The ViennaVDM2SmalltalkScript class generates a piece of Smalltalk script for a flat specification or one with a single module. Multi-moduled specifications are not supported. State invariants are not automatically checked in the generated program code. The ViennaVDM2SmalltalkScript is designed to test specifications by evaluating expressions on it or to embed a VDM expression in Smalltalk program code.

4.3 Runtime library

To generate Smalltalk code, runtime support for the generated code is required to fill the difference between VDM-SL and Smalltalk. One difference to fill is the type system. Smalltalk is a class-based and dynamically typed OO programming language and VDM-SL is a statically typed language without the OO features. One language construct in Smalltalk that possibly corresponds to the type in VDM-SL is the class. However, a naive mapping between types in VDM-SL and classes in Smalltalk is not feasible because every Smalltalk object belongs to a class while a value in VDM-SL may belong to multiple types.

ViennaTalk provides type *objects* that represent VDM-SL's types independent of classes of objects for VDM-SL values. As the types are objects in ViennaTalk, it is possible to send a message to those objects, to compose new type objects from existing type objects, and query membership of a value object. The summary of mapping of VDM-SL types to the type object in Smalltalk and the class of its value objects is shown in Table 1. The composite type and the token type have no corresponding classes in the Smalltalk standard library, and therefore ViennaComposite and ViennaToken are defined accordingly.

Some operations and language features of VDM-SL are not present in Smalltalk. The ViennaComposition class supports function compositions and the ViennaIteration implements iteration of the function composition. Pattern matching is not supported in the standard Smalltalk language. ViennaRuntimeUtils provides functionalities to simulate the pattern matching mechanisms.

4.4 Pattern matching

Pattern matching is a powerful language construct of VDM-SL which many programming languages do not support. Although pattern matching can be replaced with `if` expressions, it is not desirable to sacrifice abstraction in the exploratory phase of the specification. Supporting pattern matching is strongly required to satisfy the requirement **R1-c**. The Smalltalk language does not employ the pattern matching mechanism by itself, but ViennaTalk implements, as a library, pattern matching mechanisms for various types of patterns that appear in VDM-SL.

The ViennaRuntimeUtils class defines methods for all kinds of patterns in VDM-SL, each of which returns all possible bindings of pattern identifiers as a dictionary

Table 1. Mappings among types in VDM-SL, type objects and classes of value objects

VDM-SL type	**type object**	**the class of value objects**
nat	ViennaType nat	Integer
nat1	ViennaType nat1	Integer
int	ViennaType int	Integer
real	ViennaType real	Float
bool	ViennaType bool	Boolean
<quote>	ViennaType quote: #*quote*	Symbol
[*t*]	*t* optional	*t*'s class or UndefinedObject
t1 * *t2*	*t1* * *t2*	Array
t1 | *t2*	*t1* | *t2*	*t1*'s or *t2*'s class
set of *t*	*t* set	Set
set1 of *t*	*t* set1	Set
seq of *t*	*t* seq	OrderedCollection
seq1 of *t*	*t* seq1	OrderedCollection
map *t1* to *t2*	*t1* mapTo: *t2*	Dictionary
inmap *t1* to *t2*	*t1* inmapTo: *t2*	Dictionary
t1 -> *t2*	*t1* -> *t2*	BlockClosure
t1 +> *t2*	*t1* +> *t2*	BlockClosure
token	ViennaType token	ViennaToken
compose *t* of *f1* : *t1* *f2* :- *t2* *t3* end	ViennaType compose: '*t*' of: {*f1* . false . *t1*. {*f2* . true . *t2*}. {nil . false . *t3*}}	ViennaComposite
t inv *pattern* ==*expr*	*t* inv: [:*v* | *expr*]	*t*'s class

object. The code generator emits a piece of Smalltalk code that first gets the possible bindings by the ViennaRuntimeUtils class and then assigns the corresponding value to each pattern variable.

4.5 Preconditions and postconditions

In VDM-SL, preconditions and/or postconditions can be attached to explicit definitions of functions and operations. In exploratory modeling, runtime checking of preconditions and postconditions should be possible by the user's choice because the specification at hand may be error prone and needs runtime checking to confirm that the generated code is running as expected. Quoting preconditions and postconditions is necessary because evaluating a quoted precondition or postcondition with various kinds of parameters is an effective approach to understanding specification.

ViennaTalk's code generators support both runtime checking of preconditions and postconditions and quoting them. ViennaTalk automatically produces quoted preconditions with the `pre_` prefix and quoted postconditions with the `post_` prefix. ViennaTalk generates, from a function or an operation, a method to check the precondition and the postcondition that is separate from the method that implements the function's or the

operation's body if the runtime assertion checking option is turned on. When a precondition or postcondition is violated, an exception will be signalled accordingly. This helps the user programmer to remove the precondition and postcondition check easily. It is also possible for the user programmer to add extra precondition and/or postcondition as required by implementation details such as Smalltalk's native libraries.

4.6 Type invariants

Type invariants are also important assertions in VDM-SL. It does not only cause a runtime error when a value of the type does not satisfy the invariant, but also affects the return values of the `is_` family of expressions.

In ViennaTalk, type invariants are implemented as testing block closures held by type objects. A type object responds to the `includes:` message, which is common vocabulary among Smalltalk's collection objects for the membership query. For example, `ViennaType nat inv: [:x | x < 10]` produces the type object for `nat inv x === x < 10`, and `(ViennaType nat inv: [:x | x < 10]) includes: 100` evaluates to `false`. ViennaTalk's code generators emit a `includes:` message for the `is_` family of expressions and also for the runtime type checking.

4.7 State invariants

State invariants are another language construct of VDM-SL which is not supported in many programming languages. Along with preconditions and postconditions, state invariants must be checked at runtime when animating a specification to validate it.

ViennaTalk uses the Slot [18] mechanism of Pharo to implement state invariants. A Slot is a user-defined model of variables which defines (1) what bytecode should be generated for an assignment to a variable, (2) what bytecode should be generated for a read access to the variable, (3) what should be performed when the variable is assigned to, and (4) what should be performed when the variable is read. The `ViennaStateSlot` class is a slot for instance variables with invariants. When a variable declared as a ViennaStateSlot is assigned a value, the `inv` message is sent to the object that holds the variable. The default definition of the `inv` method is to do nothing, and ViennaTalk's code generator overrides the `inv` method to check the invariant. If the invariant is not satisfied, an exception is signalled. The state invariant is also quoted as a function.

4.8 Runtime type checking

Runtime type checking brings a subtle issue to Smalltalk code generators because Smalltalk is a dynamically typed language and has no static types on variables, arguments and return values. It is not common in Smalltalk programs to test the type of a value. Generating runtime checking code may be considered against the design rationale to generate code that is as *natural* as possible. On the other hand, runtime type checking is a strong technique in animating VDM-SL specifications for validation.

ViennaTalk addresses this issue by providing an option to enable or disable generation of runtime type checking. When the runtime type checking is turned on, the code

```
isPrime := [ :x |
        [ :_forall |
                _forall allSatisfy:
                        [ :y | (y <= 1 or: [ x = y ]) or: [ x \\ y ~= 0 ] ] ]
        value: ViennaType nat1 ]
```

Fig. 4. Smalltalk code automatically generated from a specification of prime numbers with a type bind in a set comprehension

```
isPrime := [ :x |
        [ :_forall |
                _forall allSatisfy:
                        [ :y | (y <= 1 or: [ x = y ]) or: [ x \\ y ~= 0 ] ] ]
        value: (ViennaType nat1 runtimeSet: (1 to: 256)) ]
```

Fig. 5. Smalltalk code modified to use {1,...,256} as approximation of the nat1 type

generator inserts a test using the `includes:` message to check if the value object belongs to the specified type object. If the runtime type checking fails, an exception will be signalled.

4.9 Type binds

Type binds can be used in set comprehensions, sequence comprehensions or map comprehensions, but they are not always executable. To make a comprehension expression with a type bind executable, one can rewrite the comprehension to use a set bind with a proper definition of the set. The resulting specification is executable but sacrifices abstraction. It is desirable to enable executability without modifying the abstract specification to satisfy the requirement **R1-c**.

ViennaTalk provides the `runtimeSet` mechanism that enables the generated Smalltalk code to approximate a type object with infinite members by a finite collection object. For example, the code shown in Figure 4 is generated from the VDM-SL expression
`isPrime := lambda x:nat & forall y:nat1 & y <= 1 or x = y or x mod y <> 0`, but is not executable due to the set comprehension with a type bind. Figure 5 is a modified version that the nat1 type is given a runtime set `(1 to: 256)` for approximation at the last line. The `isPrime` in Figure 5 is executable, i.e. `isPrime value: 23` evaluates to `true`. This approximation on the last line is apparently dangerous in rigorous analysis, but can be useful, with caution, to animate specifications with set binds in the exploratory modeling.

5 Evaluation

5.1 Benchmark: Prime numbers

The benchmark used to evaluate code generators is to generate a list of prime generators from natural numbers from 2 up to 10000 shown in Figure 6. The algorithm is similar to

```
state Eratosthenes of
    space : seq of nat1
    primes : seq of nat1
init s == s = mk_Eratosthenes([], [])
end
operations
    setup : nat1 ==> ()
    setup(x) ==
        (space := [k | k in set {2, ..., x}];
        primes := []);
    next : () ==> [nat1]
    next() ==
        if space = [] then
          return nil
        else let x = hd space in
          (primes := primes ^ [x];
           sieve(x);
           return x);
    sieve : nat1 ==> ()
    sieve(x) ==
        space := [space(i)
                  | i in set inds space
                  & space(i) mod x <> 0];
    prime10000 : () ==> seq of nat1
    prime10000() ==
        (setup(10000);
        while next() <> nil do skip;
        return primes);
```

Fig. 6. Benchmark specification: enumerate prime numbers less than 10000

the sieve of Eratosthenes but slightly different. The original Sieve of Eratosthenes writes to the same array of numbers. The algorithm used for benchmarking, on the other hand, generates a new list of the remaining numbers so that the benchmark will reflect the processing speed of comprehensions which often appears in practical specifications. As a result, the benchmark results mainly reflect efficiency of sequence comprehension, if expression, sequence operations and integer arithmetics.

The benchmark was run by interpreters and code generators of VDMTools, the Overture tool and ViennaTalk. They were run on Ubuntu 16.04 on a virtual machine with i5-3210M CPU @ 2.50GHz x 2 cores and 4GB main memory. Table 2 summarises the result.

As expected the code generators run faster than the interpreters. Amongst the code generators, ViennaTalk's code generators took the best and the worst positions. The script generated by ViennaTalk was slow because the script has to capture the program context (stack frame) to perform the return statements. In many languages, capturing

Table 2. Benchmarking result

Tool	Interpreter or Code generator	Language	Pattern matching	Time (ms)	Time (Overture CG=1)
VDMTools	Interpreter	C++	Yes	22,044	79.6
VDMJ	Interpreter	Java	Yes	5,281	19.1
ViennaTalk	Code generator	Smalltalk(Script)	Yes	957	3.45
VDMTools	Code generator	C++	Yes	337	1.22
Overture tool	Code generator	Java	No	277	1.00
ViennaTalk	Code generator	Smalltalk(Class)	Yes	202	0.729
ViennaTalk	Code generator	Smalltalk(Object)	Yes	193	0.700

gcc version 5.4.0, Java HotSpot(TM) 64-Bit Server VM(build 25.101-b13), Pharo 4 with 32-bit Cog VM.

```
public static void sieve(final Number x) {
    VDMSeq seqCompResult_2 = SeqUtil.seq();
    VDMSet set_2 = SeqUtil.inds(Eratosthenes.space);
    for (Iterator iterator_2 = set_2.iterator(); iterator_2.hasNext();) {
        Number i = ((Number) iterator_2.next());

        if (!(Utils.equals(Utils.mod(
                ((Number) Utils.get(Eratosthenes.space, i)).longValue(),
                x.longValue()), 0L))) {
            seqCompResult_2.add(((Number) Utils.get(Eratosthenes.space, i)));
        }
    }
    Eratosthenes.space = Utils.copy(seqCompResult_2);
}
```

Fig. 7. Java code for the sieve operation generated by the Overture tool

the current programming context is not allowed to user programmers. The Smalltalk environment has a mechanism to capture the program context but it is costly. The classes and objects generated by ViennaTalk do not need to capture the program context, but can simply return from the method context. The cost to capture the program context is the only difference between script and classes/objects. The difference between the generated classes and the generated objects is that the classes of the generated objects are anonymous. The generated methods and internal structures are the same.

In general, it is known that native binary code compiled from C++ source runs faster than Java code on Java VM, and Java code on Java VM runs faster than Smalltalk code on Smalltalk VM. However, in this benchmark, the Smalltalk objects and classes generated by ViennaTalk ran faster than Java and C++ code generated by the Overture tool and VDMTools in that order. This simple benchmark is not sufficient to investigate why Smalltalk code generated by ViennaTalk outperforms Java code and C++ code generated from the same VDM-SL source. One possible reason is the container objects for VDM-SL values.

Figure 7 shows the Java source code generated by the Overture tool, and Figure 8 is the Smalltalk code generated by ViennaTalk. The Java code includes getter calls such as `longValue()` inside the iterator for loop. The Smalltalk code uses only functionalities provided by the standard Smalltalk library, and does not use any container objects for VDM-SL values. Smalltalk is a flexible language in which the programmer can de-

```
_sieve_: x
    space := ((1 to: space size)
            select: [ :i | (space value: i) \\ x ~= 0 ]
            thenCollect: [ :i | space value: i ])
        asOrderedCollection
```

Fig. 8. Smalltalk code for the sieve operation generated by ViennaTalk

```
_prime10000
    self setup: 10000.
    [ self next value ~= nil ] whileTrue: [  ].
    ^ primes

prime10000: _op
    | _oldState RESULT |
    RESULT := self _prime10000.
    (ViennaType nat1 seq includes: RESULT)
        ifFalse: [ ViennaRuntimeTypeError singal ].
    ^ RESULT
```

Fig. 9. Generated Smalltalk code with runtime type checking

fine new methods on the kernel classes such as `Object`, `Collection` and `Integer`. For example, ViennaTalk extends the `Collection` class to define a `power` method to compute the power set of finite sets instead of providing `VDMSequence` class and defining a `power` method on it. On the other hand, the code generators of VDMTools and the Overture tool use container classes for VDM values such as the `Number` class because the target language, namely C++ and Java, do not allow user programmers to modify primitive data types and built-in classes. ViennaTalk takes advantage of the flexibility of the Smalltalk language to avoid overhead of container objects.

5.2 Code Readability

Figure 9 shows the generated Smalltalk code from the prime10000 operation. Two methods are defined in the code: `_prime10000` and `prime10000:`. The operation body and runtime type checking is separated into different methods.

The generated code is readable and also traceable. The `_prime10000` method looks natural as if it is hand-written, and pretty printed according to the Smalltalk standard coding convention. We can also see a clear correspondence between the `_prime10000` method and the `prime10000` operation in Figure 6. The `prime10000:` method implements runtime type checking on the return value. The return value from `_prime10000` is assigned to the temporary variable called RESULT, and it is tested if it is a value of the sequence of `nat1`. In Smalltalk, runtime type checking is not usually performed because Smalltalk employs a dynamic type system. ViennaTalk's code generator gives options to turn the runtime type checking and runtime assertion tests. The user can simply turn off runtime type checking generation of the code generator, or let the code generator emit runtime type checking code and confirm the generated code runs correctly. It is also easy to remove the runtime type checking because the type checking code is clearly separated from the code of the operation body.

5.3 Liveness

The Overture tool and VDMTools require external compilers and build tools while ViennaTalk does not. After the Overture tool or VDMTools generates a source code, the user need to build a native binary file or a Jar archive, and run it. The execution is performed as a separate process.

This difference does not only affect the burden of the user, but also makes a big difference in liveness. The generation and compilation of the prime number example in Figure 6 do not take a second on ViennaTalk. When objects of generated classes are running, the user can overwrite the specification, and re-generate Smalltalk classes. The running objects keep running as instance objects of the newly generated classes. With the liveness of the ViennaTalk's code generator, the user has more flexibility and lower barrier against trial and error in exploration.

6 Conclusion and Future Work

The automated code generators from VDM-SL specifications to Smalltalk are implemented based on requirements on exploratory modeling and design rationale. The performance and readability of the generated code are satisfactory. However, we can still improve usability of the code generator by developing debugging tools on the generated code. It is desirable that the specification can be edited and debugged during execution of the generated Smalltalk bytecode. We believe it is possible to implement such a specification-level bytecode debugger using the Smalltalk's flexible frameworks for live programming.

Acknowledgments This work is partially supported by Grant-in-Aid for Scientific Research (S) 24220001 and Grant-in-Aid for Scientific Research (C) 26330099.

References

1. Andersen, M., Elmstrøm, R., Lassen, P.B., Larsen, P.G.: Making Specifications Executable – Using IPTES Meta-IV. Microprocessing and Microprogramming 35(1-5), 521–528 (September 1992)
2. Black, A., Ducasse, S., Nierstrasz, O., Pollet, D., Cassou, D., Denker, M.: Pharo by Example. Square Bracket Associates, Kehrsatz, Switzerland (2009), `http://pharobyexample.org`
3. Fitzgerald, J.S., Larsen, P.G.: Balancing Insight and Effort: the Industrial Uptake of Formal Methods. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) Formal Methods and Hybrid Real-Time Systems, Essays in Honour of Dines Bjørner and Chaochen Zhou on the Occasion of Their 70th Birthdays. pp. 237–254. Springer, Lecture Notes in Computer Science, Volume 4700 (September 2007), iSBN 978-3-540-75220-2
4. Fröhlich, B., Larsen, P.G.: Combining VDM-SL Specifications with C++ Code. In: Gaudel, M.C., Woodcock, J. (eds.) FME'96: Industrial Benefit and Advances in Formal Methods. pp. 179–194. Springer-Verlag (March 1996)
5. Fuchs, N.E.: Specifications are (preferably) executable. Software Engineering Journal pp. 323–334 (September 1992)

6. Hayes, I., Jones, C.: Specifications are not (Necessarily) Executable. Software Engineering Journal pp. 330–338 (November 1989), http://www.cs.man.ac.uk/csonly/cstechrep/Abstracts/UMCS-89-12-1.html
7. Jørgensen, P.W.V., Larsen, M., Couto, L.D.: A Code Generation Platform for VDM. In: Battle, N., Fitzgerald, J. (eds.) Proceedings of the 12th Overture Workshop. School of Computing Science, Newcastle University, UK, Technical Report CS-TR-1446 (January 2015)
8. Jørgensen, P.W.V., Larsen, P.G.: Towards an Overture Code Generator. In: The Overture 2013 workshop (August 2013)
9. Kurs, J., Larcheveque, G., Renggli, L., Bergel, A., Cassou, D., Ducasse, S., Laval, J.: PetitParser: Building Modular Parsers, pp. 377–412. Square Bracket Associates (September 2013)
10. Larsen, P.G.: Ten Years of Historical Development: "Bootstrapping" VDMTools. Journal of Universal Computer Science 7(8), 692–709 (2001)
11. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The Overture Initiative – Integrating Tools for VDM. SIGSOFT Softw. Eng. Notes 35(1), 1–6 (January 2010), http://doi.acm.org/10.1145/1668862.1668864
12. Larsen, P.G., Lassen, P.B.: An Executable Subset of Meta-IV with Loose Specification. In: VDM '91: Formal Software Development Methods. VDM Europe, Springer-Verlag (March 1991)
13. Lausdahl, K., Larsen, P.G., Battle, N.: A Deterministic Interpreter Simulating A Distributed real time system using VDM. In: Qin, S., Qiu, Z. (eds.) Proceedings of the 13th international conference on Formal methods and software engineering. Lecture Notes in Computer Science, vol. 6991, pp. 179–194. Springer-Verlag, Berlin, Heidelberg (October 2011), http://dl.acm.org/citation.cfm?id=2075089.2075107, ISBN 978-3-642-24558-9
14. Nielsen, C.B., Lausdahl, K., Larsen, P.G.: Combining VDM with Executable Code. In: Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) Abstract State Machines, Alloy, B, VDM, and Z. Lecture Notes in Computer Science, vol. 7316, pp. 266–279. Springer-Verlag, Berlin, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-30885-7_19, ISBN 978-3-642-30884-0
15. Oda, T., Araki, K., Larsen, P.G.: VDMPad: a Lightweight IDE for Exploratory VDM-SL Specification. In: Plat, N., Gnesi, S. (eds.) FormaliSE 2015. pp. 33–39. In connection with ICSE 2015, Florence (May 2015)
16. Oda, T., Araki, K., Larsen, P.G.: ViennaTalk and Assertch: Building Lightweight Formal Methods Environments on Pharo 4. In: Proceedings of the International Workshop on Smalltalk Technologies. pp. 4:1–4:7. Prague, Czech Republic (Aug 2016)
17. Oda, T., Yamomoto, Y., Nakakoji, K., Araki, K., Larsen, P.G.: VDM Animation for a Wider Range of Stakeholders. In: Ishikawa, F., Larsen, P.G. (eds.) Proceedings of the 13th Overture Workshop. pp. 18–32. Center for Global Research in Advanced Software Science and Engineering, National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-Ku, Tokyo, Japan (June 2015), http://grace-center.jp/wp-content/uploads/2012/05/13thOverture-Proceedings.pdf, GRACE-TR-2015-06
18. Verwaest, T., Bruni, C., Lungu, M., Nierstrasz, O.: Flexible object layouts: Enabling lightweight language extensions by intercepting slot access. In: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications. pp. 959–972. OOPSLA '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/2048066.2048138

Integrated Tool Chain for Model-Based Design of Cyber-Physical Systems

Peter Gorm-Larsen[1], Casper Thule[1], Kenneth Lausdahl[1], Victor Bandur[1], Carl Gamble[2], Etienne Brosse[3], Andrey Sadovykh[3], Alessandra Bagnato[3], and Luis Diogo Couto[4]

[1] Aarhus University, Department of Engineering, Denmark
[2] School of Computing Science, Newcastle University, UK
[3] Softeam Research & Development Division, Paris, France
[4] United Technologies Research Center, Cork, Ireland

Abstract. Having a well-founded connection between different modelling tools such that they form a chain from requirements over formal descriptions for the constituent elements towards final realisations of Cyber-Physical Systems (CPSs) is essential. In this tool paper we explain how this can be achieved with a collection of baseline tools that are adapted to fit into such an open tool chain. The semantic foundations for the different notations used for CPSs are based on different parts of mathematics and the heterogeneous nature of these gives challenges that are solved in the suggested tool chain.

Keywords: Well-founded tool chain, discrete event formalism VDM-RT, continuous-time formalism OpenModelica, co-simulation, FMI

1 Introduction

The development of Cyber-Physical Systems (CPSs) is challenging. The close interaction between a computer-controlled cyber part and a physical part makes it hard to make the kinds of abstractions normally made from a computer science perspective. The INTO-CPS project targets the production of a well-founded chain of tools for the model-based development of CPSs [8]. In this paper we present an overview of the tool chain, its semantic foundations, baseline tools that are adapted to fit this setting and its envisaged work flow.

In the INTO-CPS project this new technology is being tested with industrial case studies in four different application domains (automotive, railways, agriculture and building automation). In addition smaller academic sized pilot studies are carried out in order to easier introduce the different features of the technology. In this paper we make use of a small line-following robot pilot study that originally was introduced in one of the predecessor projects called DESTECS[5] [4].

Throughout the paper, images are used to illustrate the features being described. Where these images show model features they are taken from the models

[5] `http://destecs.org/`.

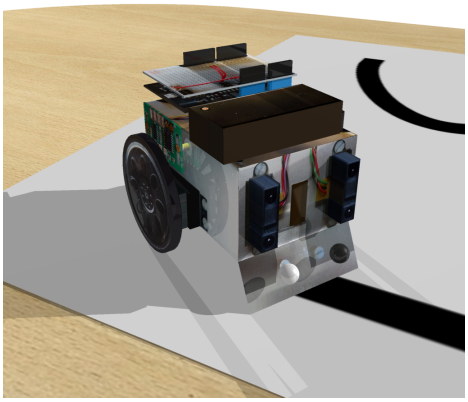


Fig. 1: The Line Follower Robot rendered from within 20-sim

of a Line Follower Robot that is being used as a pilot study in the INTO-CPS project. The robot, shown in fig. 1, comprises two infrared reflectivity sensors to detect a line, a body housing two servo motors connected to wheels for locomotion and a micro controller that reads the sensor values and produces signals that set the speed and direction of the servo motors.

The rest of the paper starts off with a short introduction to the Functional Mockup Interface (FMI) since this is essentially the glue that enables coupling between mathematical models produced in different notations and tools in Section 2. This is followed by a brief description of the semantic foundation of FMI which is using Unifying Theories of Programming (UTP) behind the scene in Section 3. Afterwards, an overview of the baseline tools that are adapted to fit an FMI context are presented in Section 4. Then Section 5 introduces a new application that serves as a front-end for end-users (typically domain experts) who need not be experts in any of the models used in the description of a CPS. The intended use of the tool chain is then presented in Section 6, followed by an outline of its design space exploration support in Section 7. Finally, Sections 9 and 10 complete the paper with information about related work, concluding remarks and future work respectively.

2 The Functional Mock-up Interface

When developing a CPS it can be useful to create models of the constituent components, that make up the system. These models can represent both cyber and physical parts and be described in different forms based on their nature such as Discrete Event (DE) and Continuous-Time (CT). These constituent models can then be used in a collaborative simulation (co-simulation), which couples the models created in different formalisms. Thereby the entire system

can be simulated by simulating the constituent models and exchange data as the common simulated time is progressing. In principle many systems can be approximated with a DE or a CT approach alone. However, in order to accurately describe CPSs where the physical dynamics are non-linear such approximations would get far away from reality.

The Functional Mock-up Interface (FMI) defines a standardised interface to be used in computer simulations to develop complex CPSs. Such co-simulations are typically organised with a master-slave architecture, where a Master Algorithm (MA) is used to orchestrate a simulation. The simulation often consists of three phases: Initialisation, simulation, and tear down. In the initialisation phase the master retrieves the properties of the slaves and establishes communication links. Next, in the simulation phase the MA resolves the dependencies between slaves and invokes each slave to progress for a given time step. The slaves might reject the step and a rollback of one or more slaves can be necessary, and the simulation can be attempted again with a different step size. Lastly the outputs of the slaves are retrieved and the process repeats until a predetermined end time is reached. The final phase is freeing the slaves, releasing resources, and similar.

As mentioned above the models are often created in different formalisms and therefore require different specialised simulation tools [2]. This leads to developing solutions for specific systems instead of a general applicable approach, which is expensive. FMI was created to solve these challenges, as it is a tool-independent standard for co-simulation [3]. The standard describes C interfaces, that a slave must partly or fully implement in order to participate in a co-simulation using FMI. Such a slave is then referred to as a Functional Mock-up Unit (FMU). This makes it possible for the FMUs to contain their own solvers while still adhering to FMI, and provides an opportunity for developing generalised solutions[6].

3 Semantic Foundation

The complete semantic foundation of the INTO-CPS tool chain consists of individual semantics for the fundamental underlying activities: modelling of CT and DE systems, and co-simulation in accordance with the FMI standard. Semantics for models of CT systems is provided by a UTP formalisation of a new hybrid relational calculus with differential algebraic equations [11]. Semantics for models of DE systems is provided by a novel UTP semantics of object orientation [10], the newest semantic foundation for the Vienna Development Method's real-time dialect (VDM-RT), which also forms the semantic basis for a C code generator. The semantics of FMI co-simulation is captured in a new formalisation [6] of the FMI standard in *Circus*, a re-casting of earlier work [6] expressed in the process algebra of Communicating Sequential Processes (CSP) [14].

The *Circus* semantics of FMI captures formally the description of co-simulation given in the standard. A generic MA is modelled which determines how FMUs

[6] See [5] for more information regarding MAs for co-simulation using FMI.

are orchestrated in terms of passage of time, requests to take a simulation step and exchange of simulation results. A model of a valid FMU is also defined. These elements combine into a full semantics of co-simulation according to the FMI standard used inside the Co-simulation Orchestration Engine (COE). The key advantage of this formalisation is that it can be checked for desirable properties, such as freedom from livelock and deadlock, using the CSP refinement checker called FDR3 [13]. Indeed this has already revealed that the example master algorithm given in the standard makes an implicit assumption that FMUs do not fail in a way that is fatal to the overall co-simulation. When models of specific putative master algorithms and FMUs are also constructed as *Circus* processes, FDR3 can be used to check that the resulting specific co-simulation model is a refinement of the FMI co-simulation semantics. Expressing a particular co-simulation using the semantics can also be used to formally verify the result of executing the same co-simulation using the COE. An ongoing study is investigating how to perform this verification process [23].

4 Baseline Tools

The INTO-CPS tool chain has been defined on top of five existing baseline tools. Each tool, described in the following paragraphs, is extended to fit the FMI context defined into the INTO-CPS project.

Modelio[7] is an open-source modelling environment supporting industry standards like UML and SysML. This is used for high-level system architecture modelling, Modelio extends the SysML language [21] and proposes extensions for CPS modelling. The extended system modelling language allows, in particular, requirement, FMI interface, and FMU connections definition which can be used for generation of FMI model descriptions and configurations of co-simulations.

Overture[8] supports modelling and analysis in the design of discrete, typically, computer-based systems using VDM-RT dialect including both time and distribution of functionality on different computational nodes [24]. VDM-RT is based upon the object-oriented paradigm where a model is comprised of one or more objects. An object is an instance of a class whereas a class gives a definition of zero or more instance variables and operations an object will contain. Instance variables define the identifiers and types of the data stored within an object, while operations define the behaviours of the object.

The *20-sim*[9] tool can represent continuous time models using connected blocks [16]. Bond graphs may implement such blocks [12]. Bond graphs offer a domain-independent description of a physical system's dynamics, realised as a directed graph. The vertices of these graphs are idealised descriptions of physical phenomena, with their edges (bonds) describing energy exchange between vertices. Blocks may have input and output ports that allow data to be passed between them. The energy exchanged in 20-sim is the product of effort and flow,

[7] `http://www.modelio.org/`
[8] `http://overturetool.org/`
[9] `http://www.20sim.com/`

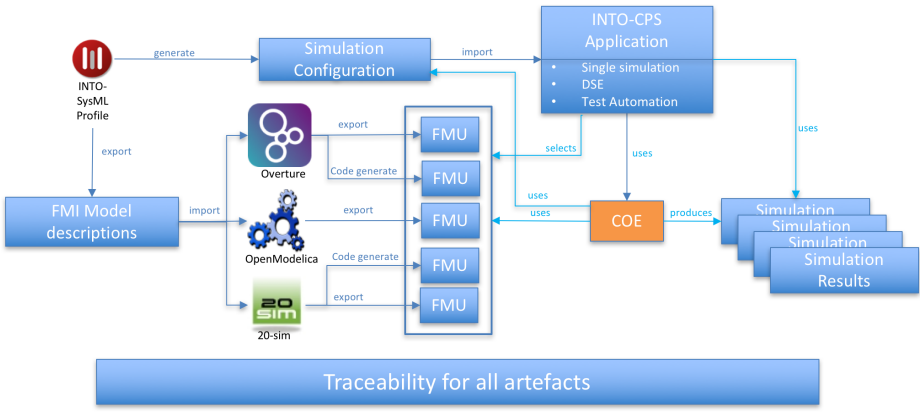


Fig. 2: The current INTO-CPS Tool Chain

which map to different concepts in different domains, for example voltage and current in the electrical domain.

OpenModelica[10] is an open-source Modelica-based modelling and simulation environment. Modelica is an object-oriented language for modelling of large, complex, and heterogeneous physical systems [16]. Modelica models are described by schematics, also called object diagrams, which consist of connected components. Components are connected by ports and are defined by sub components or a textual description in the Modelica language. Overture, 20-sim, and OpenModelica are used for specifying FMI behaviour in their own formalism. These three tools are extended in order to consume the FMI interface definition defined previously, and, after modelled the FMI implementation, provide a FMU, conform to given FMI, for co-simulation.

RT-Tester[11] is a test automation tool for automatic test generation, test execution, and real-time test evaluation [18]. The RT-Tester Model Based Test Case and Test Data Generator supports model-based testing: automated generation of test cases, test data, and test procedures from SysML models. In our context, tests are generated as FMUs which are executed against the system under test.

The different baseline tools are combined together forming a chain of tools as illustrated in fig. 2. The core of the integration here is ensured by the INTO-CPS Application introduced below.

5 The INTO-CPS Application

In the INTO-CPS Project, the INTO-CPS application[12] has two primary responsibilities: defining an INTO-CPS project structure, and providing a UI for tool chain features that are not exposed via baseline tools such as co-simulation.

[10] `https://www.openmodelica.org/`
[11] `http://www.verified.de/products/rt-tester/`
[12] Available from `https://github.com/into-cps/intocps-ui`.

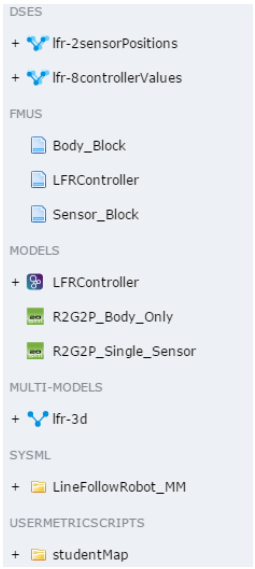


Fig. 3: The project browser.

The INTO-CPS application has two regions – the project browser on the left side, and the main view in the center. The project browser shows the main artefacts in an INTO-CPS project. The browser is shown, with an example project open, in fig. 3. The main view changes dynamically, based on the activity currently being carried out by the user.

An INTO-CPS project is based on two kinds of entities: models that are produced by the baseline tools, and artefacts that are derived from models such as the results from co-simulations. The INTO-CPS application primarily interacts with model-derived artefacts. A particularly relevant model-derived artefact is the *Multi-Model*, that is produced from a combination of a connection mapping and loaded FMUs and submitted to the Co-simulation Orchestration Engine (COE) for co-simulation. The INTO-CPS application is capable of creating and editing Multi-Models, as shown in fig. 4.

From a Multi-Model, the application is capable of generating and then editing a Co-Simulation configuration which originally can be generated from SysML, as shown in fig. 5. This enables application users to set various relevant co-simulation parameters such as start and end time, the desired co-simulation algorithm, and which variables to livestream.

It is possible for the user to download the COE (and other INTO-CPS tools) and execute it from within the INTO-CPS application, as shown in fig. 6. In this way the newest released version of all the tools in the overall tool chain can always be obtained with minimum effort.

If variables have been selected for live-streaming in the co-simulation configuration, the application will plot these variables dynamically as they are streamed

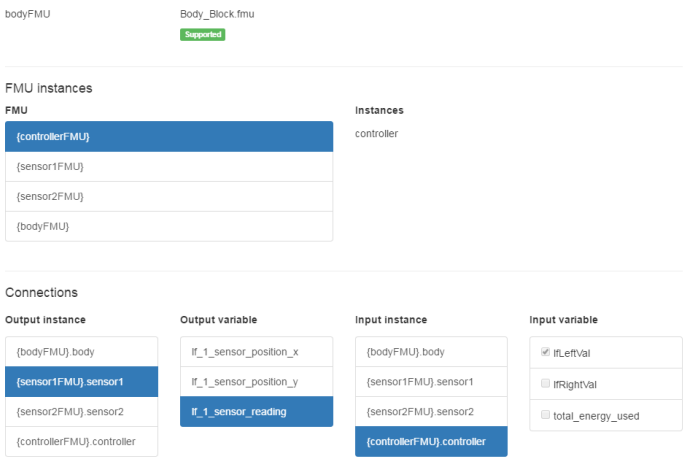


Fig. 4: Multi-Model editor.

by the COE – see fig. 7. Afterwards, the plot can be explored visually and exported as an image. The full results of the co-simulation are also exported as a Comma-Separated Values (CSV) file, for further analysis after a co-simulation.

In terms of end users, the primary intent behind the application is to enable stakeholders that are not experts in any of the INTO-CPS modelling notations to still be able to execute and evaluate co-simulations. This is possible in the current version of the application via the co-simulation configuration view and plotting and export of results. Additional views are in development for tool chain features such as traceability analysis and model checking. These views are kept isolated from each other in order to allow different kinds of experts to focus on their own tasks without being distracted by UI elements that are not relevant to them. The only view that is always visible is the project browser, since it provides navigation between views by selecting the relevant files.

6 Work flow

The INTO-CPS tool chain includes many tools and spans from requirements through to simulation results and generated source code and as such it may not be immediately apparent how to begin using it. Figure 8 shows and outline of the suggested workflow along with two entry points into the tool chain. The first entry point is to use SysML to model and decompose the system into tractable blocks for later analysis and development. Here the modelling is guided by an INTO-CPS SysML profile that defines suitable diagrams and model elements. This entry point requires knowledge about SysML. The second entry point is used when an organisation has pre-existing FMU models, here a subset of the SysML profile diagrams may be used to compose the FMUs into a model of the whole system. These approaches are not mutually exclusive and it is possible to compose system models from a mix of new and pre-existing models.

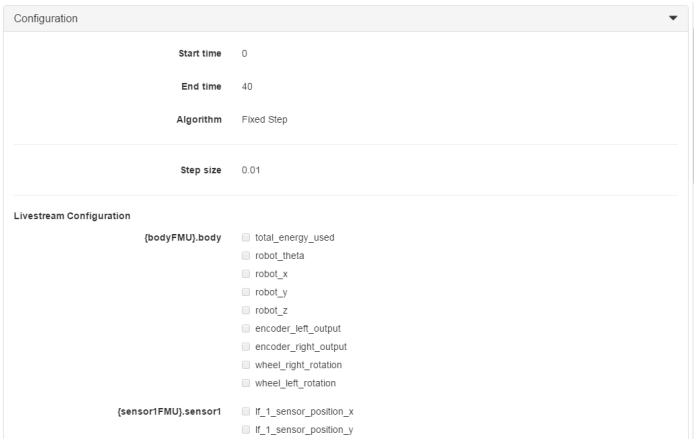


Fig. 5: Co-Simulation configuration editor.

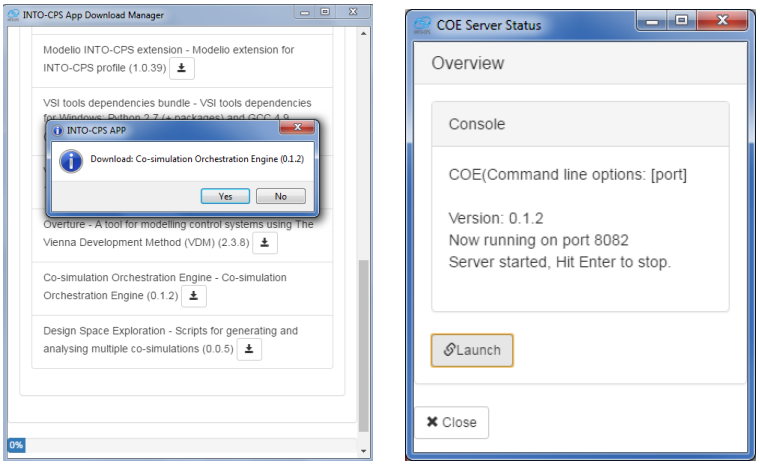
Using either the first or the second entry point both lead to the definition of a set of FMU and their connections. A specific diagram has been defined inside the INTO-CPS-SysML profile for this purpose. This is called a Connections Diagram (CD) and it represents the instantiation (possibly multiple) of FMUs and the connection existing between FMU inputs and outputs. Here fig. 9 has been extracted from the line following robot case study of the INTO-CPS project. Four instances (named controller, body, sensor1, and sensor2) of three FMUs (named Controller, Body, and Sensor) are connected.

From this diagram, a simulation configuration can be generated and then enhanced using the INTO-CPS Application.

The resulting multi-models may be analysed using a range of techniques. Simulation is the primary technique, where single designs or sets of designs, automatically generated by Design Space Exploration (DSE) scripts (see Section 7), are measured according to objectives and the values of these objectives are used to rank designs in partial order of preference. Formal analysis techniques are also supported in the form of Linear Temporal Logic (LTL) formulas acting as witnesses that temporal constraints on simulations are respected and the model checking of state machine representations of suitably abstracted CT and DE models [19].

As development proceeds further confidence may be gained by performing software in the loop (SiL) and hardware in the loop (HiL) simulation. Here the open nature of FMI and the COE allows selected model components to be replaced by their realised counterparts that then take part in simulations. Cyber components may be based upon source code automatically generated from DE models while CT models are replaced by physical components, such that simulation results may be validated.

To help manage the complexity of CPS development including many modelling artefacts, the tool chain includes support for tracking model provenance



(a) Download Manager. (b) COE Execution.

Fig. 6: Downloading and launching the COE.

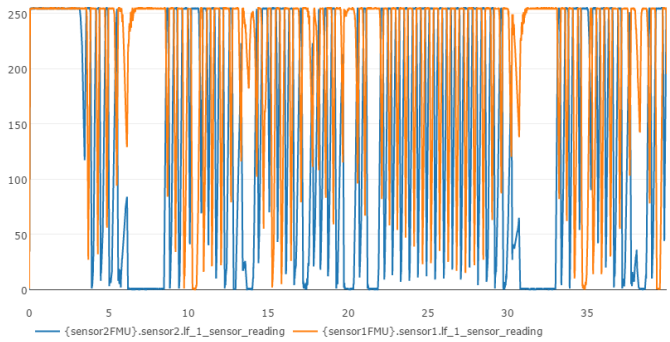


Fig. 7: Plotting of livestream variables.

and requirements traceability using a complimentary set of PROV [17] and OSLC [1] relations. Using the tools to capture these relations results in a graph that records the provenance of the modelling artefacts along with links to the related requirements. The resulting graph,of modelling and simulation activities and artefacts may then be queried to support, for example, an impact analysis exercise. A fragment of such a graph can be seen in FIgure 10.

7 Design Space Exploration

As an engineer proceeds with the design of a CPS they will likely be faced with many options and design parameters that must be decided upon for the final CPS to be produced. Here DSE support within INTO-CPS can be of assistance. This

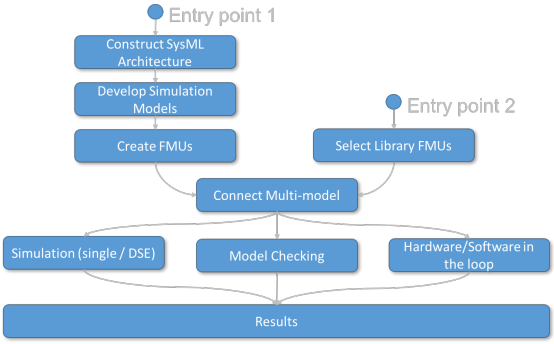


Fig. 8: Outline work flows for using the INTO-CPS tool chain

can be divided into three parts, support for analysing each simulation; support for ranking of competing designs; and algorithms that automatically sweep over ranges of parameter values.

The DSE scripts support both open and closed loop exploration of a design space. The open loop, exhaustive algorithm, is simplest of all and it will result in every combination of the design parameters being simulated. This results in a complete coverage of the design space but it suffers from the space explosion problem and so it only generally practical for small design spaces. The closed loop algorithms, such as a genetic algorithm, use past simulation results to generate new designs to be simulated with the goal of finding a set of optimal designs without having to explore the entire design space.

In order to compare individual simulations we must have measures that characterise their behaviour in some way, these we term the objective values. The DSE scripts include built-in support to calculate a range of simple objectives from the raw simulation data, for example finding the maximum value of some variable of the simulation. It also allows the user to define their own objective scripts to calculate measures that are specific to a model or its configuration. Taking the line follow robot as an example, it uses two user defined objective scripts, one to calculate the lap time round a track and another to calculate the mean cross track error, which is a measure of how accurately the robot followed the line.

Using these objective scripts to reduce the raw simulation results to a few measures of performance allows the engineer to define a method ranking a set of designs. The engineer may define which objective values are important for the ranking of designs and whether higher or lower values are preferred for each. Using this information the scripts are able to rank the results of all the simulations that have been run using Pareto efficiency [13] to produce a non-dominated set of results representing a range of trade-offs between the selected

[13] A description of Pareto Efficiency may be found at `https://en.wikipedia.org/wiki/Pareto_efficiency`

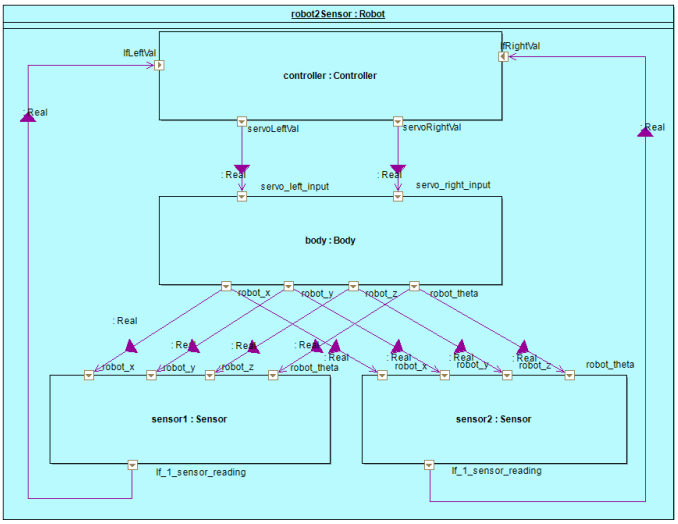


Fig. 9: The connections diagram for the line-following robot

objectives. Here fig. 11 shows the result of varying controller parameters for the line follower robot, using the lap time and mean cross track error as the means to compare designs. Here the non-dominated set, which is the green bottom-left most line, plots the results of the non-dominated set and therefore the best designs according to these measures. The DSE functionality can also be invoked from the INTO-CPS application where the graph is accompanied by a table allowing the engineer to determine the design parameters that produced each of these results.

8 Model Refinement and Implementation

The various optimization and verification mechanisms of the INTO-CPS tool chain enable the development of CPS multi-models to high levels of maturity. Once it is confirmed that the constituent models behave as expected in their environment, it is desirable to refine some of these to executable implementations. This mostly applies to models of control software, but there are situations in which executable implementations of models of continuous systems are desired (for instance, real-time co-simulation against cost-prohibitive environments such as large engines.) Modelica and 20-sim can generate such implementations.

With INTO-CPS, control software is discrete in nature, and is modelled in VDM-RT using Overture. There exist two approaches to refinement of models to executable implementations: formal stepwise refinement, and code generation. Since there is currently no formally defined refinement strategy for VDM-RT, Overture adopts the code generation approach. Overture's C code generator embodies a refinement strategy for VDM-RT that builds on the semantic foundation due to Foster *et al.* described in Section 3. The code generator essentially

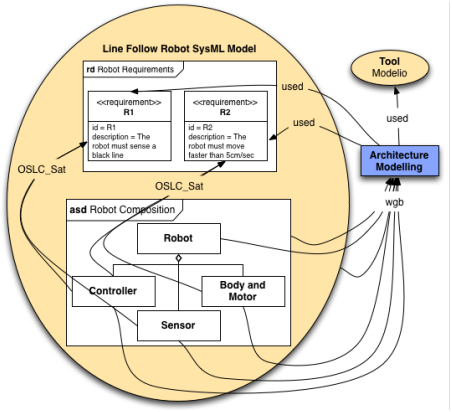


Fig. 10: Traceability links around architecture modelling of the line follower robot

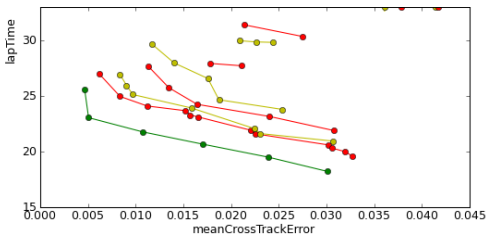


Fig. 11: Pareto plot of DSE results.

achieves an automated, one-step refinement directly to C. This refinement process is therefore not formal, but the strategy implemented is thoroughly tested to ensure that the resulting implementations conform to the aforementioned semantics. Because the refinement step is fully automated, only an executable subset of VDM-RT can be used for model construction. Naturally some underspecification (or *looseness*), for example an arbitrary choice of value from a set, can be accommodated. In contrast, "manufacturing behaviour" in accordance with contract-based specifications is considered outside the remit of code generation in the INTO-CPS context, and such constructs are not allowed in the executable subset of the language.

As a proof of concept, Overture's C code generator was used to generate an implementation of a model of a simple on/off controller that maintains the level of water in a tank between some specified limits. The core of the model is excerpted in Fig. 12. The implementation was compiled and executed on an Atmel ATmega 1284P development board[14]. A potentiometer was used to manually emulate the water level in the tank and an LED was used as feedback of the status of the tank drain valve. This is likewise shown in Fig. 12. This example

[14] A demonstration video can be found at `https://youtu.be/Qgw5NAgv3pw`.

```
loop()==
  cycles(2)
  let level : real = levelSensor.getLevel()
  in
  (if( level >=
      HardwareInterface‘maxlevel.getValue())
   then valveActuator.setValve(open);
   if( level <=
       HardwareInterface‘minlevel.getValue())
   then valveActuator.setValve(close);
  ); );
```

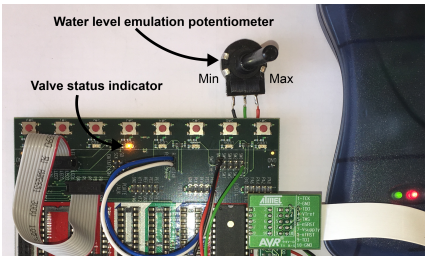


Fig. 12: Excerpt of water tank valve controller and hardware simulation.

deployment is typical of the final stages of the INTO-CPS workflow, and configuration of the hardware interface and periodic call to the control task was the only necessary manual intervention.

This exercise revealed that the value semantics of VDM-RT is one of the most problematic aspects of the language, as a naïve strategy results in implementations with very large memory footprints. For very resource-constrained embedded platforms, very aggressive measures for reducing memory usage are necessary. We have observed that, of all the language features, value semantics must receive priority when designing these measures.

9 Related work

The INTO-CPS tool is related to the DESTECS and SPEEDS projects, which both supports co-simulation with their own protocols and tools but they do not make use of the FMI standard as in the INTO-CPS tools. The Ptolemy II [20] is a single-tool simulation and modelling platform which can perform simulation of heterogeneous models. The tool has the ability to import standalone FMUs, leading to a high degree of model heterogeneity through a combination of native domains and external FMUs. However, it is unclear at this time whether tool-wrapper FMUs can be co-simulated. The iCyPhy project [9] focuses on the semantics of component interoperation, but a simulation tool based on Ptolemy II, FIDE [7], achieves co-simulation of FMUs.

The DANSE project models System of System (SoS) using block diagrams and is able to export this as FMUs which can be simulated in their DESYRE environment. The project developed its own specification language, the DANSE language. In addition to simulation, the project also supports statistical model checking and optimised simulation based on metrics of interest. Both are carried out by reading information directly from DANSE specifications, since the FMI standard does not include the required structural information. Of note is the fact that the technology allows multiple levels of abstraction of any given model component in a simulation. The connection between the two levels is made stochastically. It is believed that allowing such multi-level abstraction makes simulations requiring high numbers of components more tractable while still yielding accurate results. In terms of simulation support, the project supports both local

simulation (termed "hosted simulation"), as well as distributed co-simulation, in the sense of INTO-CPS. However, the project makes use of FMI only for hosted simulation, where essentially only standalone FMUs are co-simulated on a single host, whereas distributed co-simulation is achieved through the use of the US Department Modeling and Simulation's High-Level Architecture (HLA). Further information is available from the project website, as all project deliverables are publicly available [15].

The CosiMate project develops a co-simulation approach for heterogeneous systems which is very similar to INTO-CPS. However, it allows the connection of external simulation tools not only through FMI, but also through their native control interfaces. Addition of a new simulation tool to a co-simulation scenario is facilitated by an Eclipse-based interface construction environment. The co-simulation platform supports variable time steps in the same way at the COE from INTO-CPS.

The ADVANCE project[15] [22] allows co-simulation of Event-B machines with external continuous-time FMUs through FMI version 1. The resulting technologies support model-based testing and model-checking of CPS using ProB. The co-simulation capabilities of the ADVANCE MultiSim simulation framework are similar to those projected for the INTO-CPS tool chain, and are implemented as a plugin for the Rodin platform for Event-B. However, owing to the capabilities of Rodin, proof in that domain is better integrated with the relevant tool than current proof support for VDM-RT, but INTO-CPS has the main advantage that it seeks to make a co-simulation platform. The aim in INTO-CPS is to co-simulate both discrete-event and continuous-time FMUs together without knowing the details about the implementation of the FMUs, as long as they are compatible with the FMI version 2 standard. Further information is available from the project website, as all project deliverables are publicly available. This work will like the above mentioned projects support FMI for all base line tools and therefore enable fixed/variable-stepsize co-simulation. In addition, it will provide traceability support, and test automation at the FMU level as well as model checking, and design space exploration for optimised simulation based on metrics of interest.

10 Concluding Remarks and Future Work

In this paper we have provided an overview of the INTO-CPS tool chain and briefly touched upon its foundations. We believe that in order to ensure interoperability between different models of different constituent elements of a CPS, a semantic foundation such as suggested above is paramount. This is an area where we hope that others in the formal methods community will take inspiration from this work.

The INTO-CPS tool chain described in this paper is not yet complete, but the connectivity between the different parts has already been demonstrated: the tool

[15] `http://www.advance-ict.eu/`

chain is being used for industrial case studies in railways, agriculture, automotive and building automation. Most notably, support for traceability, essential for providing well-founded arguments for the analysis conducted for the different models to be presented to external stakeholders, is not yet implemented.

Acknowledgments. We would like to thank the anonymous referees for valuable input on this work. Our current work is partially supported by the INTO-CPS project (Horizon 2020, 664047). We would also like to thank all of the INTO-CPS participants for all their contributions making the INTO-CPS tool chain a reality.

References

1. Open Services for Lifecycle Collaboration (OSLC). `http://open-services.net/`
2. Bastian, J., Clauss, C., Wolf, S., Schneider, P.: Master for Co-Simulation Using FMI. In: 8th International Modelica Conference (2011)
3. Blochwitz, T.: Functional Mock-up Interface for Model Exchange and Co-Simulation. `https://www.fmi-standard.org/downloads` (July 2014)
4. Broenink, J.F., Larsen, P.G., Verhoef, M., Kleijn, C., Jovanovic, D., Pierce, K., Wouters, F.: Design Support and Tooling for Dependable Embedded Control Software. In: Proceedings of Serene 2010 International Workshop on Software Engineering for Resilient Systems. pp. 77–82. ACM (April 2010)
5. Broman, D., Brooks, C., Greenberg, L., Lee, E.A., Masin, M., Tripakis, S., Wetter, M.: Determinate Composition of FMUs for Co-Simulation. In: 13th International Conference on Embedded Software (EMSOFT), Montreal (September 2013), `http://chess.eecs.berkeley.edu/pubs/1002.html`
6. Cavalcanti, A., Woodcock, J., Amálio, N.: Behavioural models for fmi co-simulations. In: Sampaio, A.C.A., Wang, F. (eds.) International Colloquium on Theoretical Aspects of Computing. Lecture Notes in Computer Science, Springer (2016)
7. Cremona, F., Lohstroh, M., Tripakis, S., Brooks, C., Lee, E.A.: FIDE – An FMI Integrated Development Environment. In: Symposium on Applied Computing (2015)
8. Fitzgerald, J., Gamble, C., Larsen, P.G., Pierce, K., Woodcock, J.: Cyber-Physical Systems design: Formal Foundations, Methods and Integrated Tool Chains. In: FormaliSE: FME Workshop on Formal Methods in Software Engineering. ICSE 2015, Florence, Italy (May 2015)
9. Fizher, A., Jacobson, C.A., Lee, E.A., Murray, R.M.: Industrial Cyber-Physical Systems – iCyPhy. In: et al., M.A. (ed.) Complex Systems Design and Management. pp. 21–37. Springer (January 2014)
10. Foster, S.: Final version of the Semantics for VDM-RT. Tech. rep., INTO-CPS Deliverable, D2.2b (December 2016)
11. Foster, S., Thiele, B., Woodcock, J.: Differential Equations in the Unifying Theories of Programming. Tech. rep., INTO-CPS Deliverable, D2.1c (December 2015)
12. Gawthrop, P.J., Bevan, G.P.: Bond-graph modelling: A tutorial introduction for control engineers. IEEE Control Systems Magazine 27(2), 24–45 (2007)
13. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.: FDR3 — A Modern Refinement Checker for CSP. In: Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 8413, pp. 187–201 (2014)

14. Hoare, C.: Communicating Sequential Processes. Communications of the ACM 21(8) (August 1978)
15. IEEE Standard for Modeling and Simulation: High Level Architecture (HLA)– Framework and Rules. IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000) pp. 1 –38 (August 2010)
16. Kleijn, C.: Modelling and Simulation of Fluid Power Systems with 20-sim. Intl. Journal of Fluid Power 7(3) (November 2006)
17. Moreau, L., Missier, P.: PROV-DM: The PROV Data Model. Tech. rep., World Wide Web Consortium (2012), http://www.w3.org/TR/prov-dm/
18. Peleska, J.: Industrial-Strength Model-Based Testing - State of the Art and Current Challenges. Electronic Proceedings in Theoretical Computer Science abs/1303.1006, 3–28 (2013)
19. Pnueli, A.: The Temporal Logic of Programs. In: 18th Symposium on the Foundations of Computer Science. pp. 46–57. ACM (November 1977)
20. Ptolemaeus, C. (ed.): System Design, Modeling, and Simulation using Ptolemy II. Ptolemy.org (2014), http://ptolemy.org/books/Systems
21. Sandford Friedenthal, Alan Moore, R.S.: A Practical Guide to SysML. Morgan Kaufman OMG Press, Friendenthal, Sanford, First edn. (2008), ISBN 978-0-12-374379-4
22. Savicks, V., Butler, M., Colley, J.: Co-simulating event-B and Continuous Models via FMI. In: Proceedings of the 2014 Summer Simulation Multiconference. pp. 37:1–37:8. SummerSim '14, Society for Computer Simulation International, San Diego, CA, USA (2014), http://dl.acm.org/citation.cfm?id=2685617.2685654
23. Thule, C.: Verifying the Co-Simulation Orchestration Engine for INTO-CPS. In: Doctoral Symposium FM 2016. Limassol, Cyprus (November 2016)
24. Verhoef, M., Larsen, P.G., Hooman, J.: Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006: Formal Methods. pp. 147–162. Lecture Notes in Computer Science 4085, Springer-Verlag (2006)

Using JML-based Code Generation to Enhance Test Automation for VDM Models

Peter W. V. Tran-Jørgensen[1], Peter Gorm Larsen[1], and Nick Battle[2]

[1] Aarhus University, Department of Engineering, Finlandsgade 22, 8200, Denmark
{pvj,pgl}@eng.au.dk
[2] Fujitsu Services, Lovelace Road, Bracknell, Berkshire. RG12 8SN,UK
nick.battle@gmail.com

Abstract. The Vienna Development Method (VDM) uses contract-based elements such as invariants to constrain data, and pre- and post conditions to specify intended behaviour. Data and behaviour can be validated against these contract-based elements using test automation for VDM. This technique uses traces – a kind of pattern – to specify test sets, which can be executed against the VDM model. In this paper we demonstrate how traces can be code generated to Java and used to test a version of the VDM model, implemented as a Java Modeling Language (JML) annotated Java program. This approach has potential to allow a larger number of tests to be executed since the tests are run as compiled code rather than using a VDM interpreter. To study the performance of code generated traces, execution times are compared to those obtained using different VDM interpreters.

1 Introduction

Inspired by TOBIAS [22, 23], the Vienna Development Method (VDM) [10, 14] has been enhanced with combinatorial testing – a test automation feature that uses a pattern-based notation to describe and execute test sets. In VDM, combinatorial testing is used to validate data and behaviour against invariants and pre- and post conditions [18, 19]. The tests are generated from a pattern, referred to as a *trace*, which describes a finite subset of all possible executions of a VDM model. Combinatorial testing for VDM is supported for an executable subset of VDM by the Overture [20] and the VDMJ [1] interpreters.[3] However, execution of large combinatorial test sets can be slow and demanding in terms of memory resources. To improve on this situation, this paper presents a technique that allows traces to be executed as compiled code.

Using Overture's [17, 5, 25] VDM-to-Java code generator [15, 28] a VDM-SL model can be translated to a Java program, where the contract-based VDM elements are described using Java Modeling Language (JML) constraints [21]. In this paper we extend this code generator to support traces.[4] When a code generated trace is executed using a

[3] We note that although VDMJ used to be the core of the Overture tool, Overture has been redesigned to promote its extensibility [6, 7]. The Overture interpreter and VDMJ are therefore different interpreters, although the former is inspired by the latter.

[4] The trace code generator is included in the Overture tool.

JML tool, the trace tests are expanded and run against the code generated version of the VDM specification. The verdict of each test is determined by checking the generated code against the JML constraints. There are two benefits to this approach. First, it has potential to support faster execution of larger test sets since the tests are executed as compiled code rather than using a VDM interpreter. Secondly, it allows the trace to be used to validate the generated code against the properties described by the VDM model. In our work we use the OpenJML [3] runtime assertion checker to execute the code generated trace. The reason for this is that OpenJML is the only tool that we are aware of that currently supports Java 7 as well as the subset of JML produced by Overture's Java code generator.

We have used code generated traces to analyse properties of an algorithm used to obfuscate Financial Accounting District (FAD) codes, which are used to identify branches of a retailer. The algorithm was modelled in VDM and validated using combinatorial testing. One of the interesting aspects of this case study is that it involves the generation and execution of one million tests, which code generated traces enabled us to execute. However, as we shall see, very recent advances in trace expansion techniques allows traces to be executed much more efficiently. Currently VDMJ is the only VDM interpreter that supports this expansion algorithm. To study the performance of code generated traces we compare the execution times to those obtained using Overture and VDMJ.

Combinatorial testing has been researched for several years [24] with most of the work centred around tools that support combinatorial testing for different programming languages [29]. At the level of specification languages it is worth comparing our work to [9]. In their work, Dick et al. use a finite-state automaton approach to support test sequencing of implicit-style VDM models. However, since their technique uses symbolic values, one needs to provide the means to select concrete ones.

Similar to VDM, TOBIAS supports test case generation from a pattern that describes a test set. The test cases generated by TOBIAS can be output as sequences of VDM operation calls or as JUnit [16] test cases. In particular, the latter can be used to test a Java implementation against a JML specification. Although our work currently only supports Java and JML, it also enables code generation of the VDM specification in addition to the tests. Our technique may, however, be adopted to use other Design-by-Contract (DbC) implementation technologies (section 6).

The test automation technique presented in this paper, is also comparable to what can be achieved using SAT solvers [2] since both techniques conduct an analysis for a finite collection of cases. However, while SAT solvers are limited by the number of values for each domain, our approach is guided by traces for the finite number of combinations to be analysed. Similar to our approach the Spin model checker translates a Promela model into an optimised C-program that performs exhaustive exploration of the state space [12]. In order to avoid the state explosion problem that is inherent to model checking, one needs to limit the size of the state space subject to exploration. In VDM the state space is limited by the desired paths undertaken, which is expressed using a trace.

The structure of the paper is as follows: after this introduction, combinatorial testing for VDM is described in section 2. Next, our approach to code generating traces is

presented in section 3. Afterwards, the performance of code generated traces is studied using an industrial case study in section 4. This is followed by a discussion of the performance results in section 5. Finally, this paper concludes and outlines future work in section 6.

2 Background

2.1 VDM

VDM is one of the longest established formal methods for the development of computer-based systems. In VDM data are defined by means of types built using constructors that define records and collections such as sets, sequences and mappings from basic values such as booleans, characters and numbers. Data types can be further constrained with type invariants and the overall system state can similarly have a state invariant defined. Functionality is defined in terms of functions and operations over data types. These can be defined implicitly by pre conditions and post conditions that characterise their behaviour, or explicitly by means of specific algorithms. Function and operation arguments are passed by value, which avoids the complexity of value aliasing.

Collectively, the state/type invariants and pre- and post conditions define "contracts" that the specification must meet. A specification implicitly defines functions that represent each of its constraints. For example, a pre condition defines a total function which has the same parameters as the function (or operation) that it guards and a boolean result; the body of the function is the pre condition expression. Similarly, a type definition with an invariant implicitly defines a total function with parameters that match its value constructor and a boolean result; the body of the function is the invariant expression.

As an informative annex to the VDM-SL ISO standard [13], a module-based extension has been added to the language. This extension enables structuring of a VDM-SL model into a collection of modules with capabilities to import and export definitions to/from other modules. A module may define a single state component which can be constrained by an invariant.

2.2 Combinatorial testing of VDM models

Combinatorial tests are a form of animation that allow large numbers of tests to be generated using patterns. A combinatorial test generator expands these patterns, or traces, by considering every possible combination of values that would match the pattern. Then for each combination of values, the system is reset and an animation performed. It is not unusual to generate hundreds of thousands of tests this way, which therefore explores far more of the possible system states than ad-hoc animation testing.

Traces are closely related to normal VDM-SL expressions, but expand "looseness" to consider all possible results. For example, in a normal specification the expression `let a in set S be st p(a)` selects a value from the set `S` such that `p` is true. If there are many such values in `S`, VDM does not define which one is selected, only that the one selected meets `p`. In a trace, the same expression generates a new test, with a new binding for `a`, for every value in `S` that meets `p(a)`. Similarly, the non-deterministic statement `||(op1(), op2(), op3())` usually means that the three

operations are called sequentially, but in an undefined order. When this appears in a trace, it generates one test for every possible ordering of the calls.

Futhermore, when a trace contains two or more clauses that would expand to multiple tests, a test is generated for every *combination* of the expansions. For example, the trace in listing 1.1 calls the operation `op` with every possible combination of a value from the set `A` and a value from the set `B`.

```
let a in set A in
  let b in set B in
    op(a, b);
```

Listing 1.1. Example of trace in VDM.

At the end of the expansion, every generated test is an ordered sequence of variable assignments and operation or function calls. So if `A`=$\{$`1`$\}$ and `B`=$\{$`7,14`$\}$, the example above would expand to `a=1; b=7; op(a,b)` and `a=1; b=14; op(a,b)`. Each test will then execute successfully if the sequence of operation calls do not violate any constraints, either in the creation of the variable values or in the execution of the operations. In this context, a constraint violation is a condition that leads to a runtime-error such as an invariant violation or division by zero. A test which does not violate any constraints is considered a `PASS`. A test which breaks a constraint is normally considered a `FAIL`, but tests which violate a constraint directly when an operation is called from the test is considered `INCONCLUSIVE`. The reason is that it is possible that the specification is correct but the test generation is at fault. For example, the generation may produce test cases that violate the outermost operation pre conditions.

2.3 Code generation for VDM-SL

The work presented in this paper is implemented as an extension of Overture's VDM-to-Java code generator. This code generator represents each VDM-SL module using a **`final`** Java class that has a **`private`** constructor to protect against instantiation and subclassing. The module class uses a **`static`** field to represent the state component (if defined) and functions and operations are translated to **`static`** Java methods that are added to the module class.

The generated Java code is supported by a small runtime library, which includes Java implementations of some of the VDM types and operators. For example, sets, sequences and maps are implemented using the `VDMSet`, `VDMSeq` and `VDMMap` classes, which are extensions of standard Java collections.

Overture's Java code generator can translate pre- and post conditions and invariants of VDM-SL specifications to JML annotations that are added to the generated Java code [28]. JML is a DbC specification language, used for specification of Java classes and interfaces. To demonstrate how the code generator uses JML, consider a code generated method `f`, with input parameters $\texttt{i}_1, \ldots, \texttt{i}_n$, derived from a user-defined VDM-SL function. Then `f` has a code generated **`static`** pre condition method `pre_f` with the same parameters as `f`. To indicate that `pre_f` has no write-effects it is

marked with the JML **pure** modifier. In addition, `f` is annotated with the **requires** `pre_f(i1,...,in)` annotation to make `pre_f` a pre condition of `f`. The generated Java program can be validated against the JML annotations in order to ensure that the generated code meets the properties described by the contracts of the VDM specification.

The Java code generator also produces JML checks to ensure the consistency of VDM types when they are translated to Java. This is achieved using a function `Is(v,T)` which checks that a Java value or object reference `v` is consistent with the VDM type `T` that it originates from. For example, consider a Java value or object reference `v` which represents a VDM identifier that is either a natural number or a boolean value, i.e. it is of type **bool|nat**. In the generated Java code `Is(v,`**bool|nat**`)` is a JML predicate that is used to test whether `v` is either **true**, **false** or some positive integer. For this simple example the JML annotation checks that `Utils.is_nat(v) || Utils.is_bool(v)` is true. More complex types, such as user-defined types constrained by invariants or collection-based types, generate more complicated JML checks. The full definition of `Is(v,T)` and a description of how this function is used by the Java code generator, is described in detail in [28]. Some of the checks generated by `Is(v,T)` uses functionality of the Java code generator's runtime library, including a small extension of it called `V2J`. For example, `V2J` has functionality to check if a Java object represents an injective mapping or a non-empty sequence, which is used to check type constraints in the generated code.

The trace code generator uses the JML annotations to detect contract and type violations in the generated code. This is used to determine the verdicts of the trace tests. For example, if one of the tests violates a JML post condition then this test is considered a `FAIL`. As another example, a test that passes arguments to an operation that do not match the operation signature – with respect to the VDM-SL specification – is considered `INCONCLUSIVE`. This is detected using the JML predicates produced by `Is(v,T)`.

3 Code Generating Traces

Internally Overture represents a trace as an Abstract Syntax Tree (AST) composed of nodes that correspond to the different kinds of trace constructs (e.g. a **let** binding or a non-deterministic statement). This AST represents a pattern that can be expanded into a test set. The code generator takes a similar approach to representing traces by constructing the trace AST using trace constructs or nodes available via the Java code generator's runtime library: the `Alternative` trace node is used to represent the tests produced by the `|` trace operator or the **let be st** bindings. The **let** binding only defines trace variables. The `Concurrent` trace node represents the `||` trace operator and expands to all possible orderings of the tests of its child nodes. The `Repeat` trace node is used to repeat tests a specified number of times according to some repetition pattern, e.g. `op(x){1,2}`. The `Sequence` trace node expands to the sequencing of the tests of its child nodes. The `Statement` trace node expands to a single test, which is the invocation of a `Call` statement. The `Call` statement nodes constitute the leaves of the trace AST.

In order to construct a code generated version of the trace, the code generator produces Java code that when executed builds a trace AST, composed of nodes from the Java code generator's runtime library. To demonstrate the process of code generating a trace AST, consider the VDM-SL trace in listing 1.2. In this listing, the non-deterministic choice between `fun(x)` and `op1(x)` produces two tests: `fun(x); op1(x)` and `op1(x); fun(x)`. The repetition of `op2(x)` further produces two tests: `op2(x)` and `op2(x); op2(x)`. Since the repeated and concurrent trace operators are grouped as alternatives, and the tests are expanded for all bindings for `x`, the total number of tests accumulates to eight. These tests are shown in listing 1.3.

```
let x in set {1,2} in (
  ||(fun(x),op1(x)) | op2(x){1,2}
)
```

Listing 1.2. Example of a trace specified using VDM-SL.

```
x = 1; fun(x); op1(x);   /* Test 1 */
x = 1; op1(x); fun(x);   /* Test 2 */
x = 1; op2(x);           /* Test 3 */
x = 1; op2(x); op2(x);   /* Test 4 */
x = 2; fun(x); op1(x);   /* Test 5 */
...
x = 2; op2(x); op2(x);   /* Test 8 */
```

Listing 1.3. The tests generated from the trace in listing 1.2.

At runtime the code generated version of a trace is represented as an object tree composed of the different trace nodes and trace variables used during the expansion of the trace. For the trace in listing 1.2, the trace AST is visualised using the Unified Modeling Language (UML) object diagram in fig. 1.

Expansion and execution of the trace is handled entirely by the runtime library, and performed using the `ExecTests` method in the `TraceNode` class. The execution of the trace tests is shown using a UML sequence diagram in fig. 2. In this figure `ast` represents the trace AST, `module` is the code generated version of the module enclosing the trace, `testAcc` is used to record the test results, and finally `store` is used to manage system states between the different test runs. As described in section 2, the system is reset between each test, i.e. the tests are executed independently. The code generator uses the `store` to achieve this when executing the code generated version of the trace. The test accumulator (`testAcc` in fig. 2) receives information about each test that has been executed via the `registerTest` method. This method call has been omitted from fig. 2 to keep the figure simple. A test accumulator is implemented as a strategy [11], i.e. one test accumulator may print the test results directly to the console, another test accumulator may write the results to the file system and so on.

As a first step of the test expansion and execution, the module class enclosing the code generated trace is registered in the `store`. In case there are other module classes

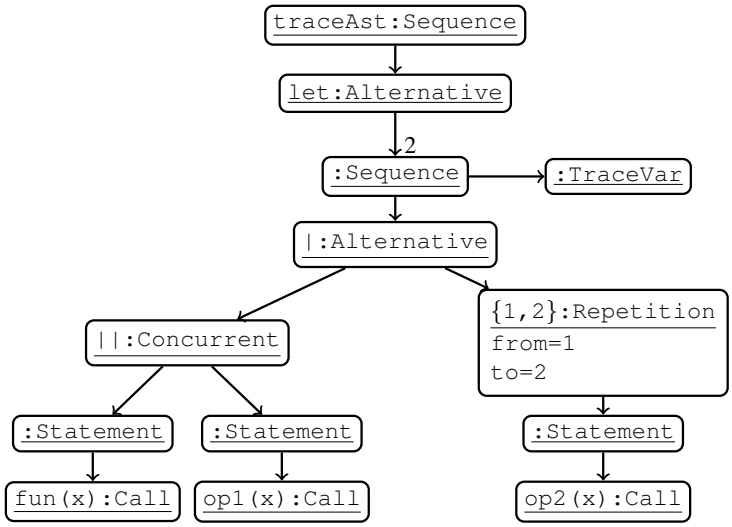


Fig. 1. Runtime representation of a code generated trace, shown using a UML object diagram.

they are also registered in the `store`, since they might also have their state changed during test execution. Subsequently the tests are derived by invoking the `getTests` method on the root of the `ast`, which returns a `TestSequence` that contains the generated tests. Next, each test is executed and the `store` is reset to restore the system state. This process continues until there are no more tests to be executed.

The `Call` statement is an abstract class defined by the runtime library. When a trace is code generated, the code produced implements and instantiates the `Call` statements as anonymous classes. As shown in fig. 3, the `Call` statement defines three methods: The `isTypeCorrect` method is used to determine whether the input to the `Call` statement matches the types of the formal parameters of the function or operation that the `Call` statement originates from. If this method returns **`false`** the current test is considered `INCONCLUSIVE`. Since the implementation of the `isTypeCorrect` method depends on the signature of the function or operation that the `Call` statement originates from, this method is implemented by the Java code generator when the trace is code generated. The `isTypeCorrect` method returns **`true`** by default, which corresponds to the situation where the input to the `Call` statement can be guaranteed, using static analysis, to be type correct. In this particular case the code generator does not have to implement the `isTypeCorrect` method. To illustrate, the construction and implementation of the `Call` statement object used to represent `op2` is shown in listing 1.4. Note that the code generated version of the argument `x` is accessed from a scope enclosing the `Call` methods.

For a `Call` statement to be type correct the input arguments $a_1, \ldots, a_n$ must match the types of the formal parameters, $T_1, \ldots, T_n$, of the function or operation that the `Call` statement originates from. Using the function `Is(v,T)`, described in section 2, we require that $\texttt{Is}(a_1, T_1)$ `&&` $\ldots$ `&&` $\texttt{Is}(a_n, T_n)$ holds in order for

the `Call` statement to be considered type correct. This check is code generated to a JML assertion, which can be configured to produce an `AssertionError` that signals that the `Call` statement is not type correct for the given arguments. As indicated by the generated JML check in listing 1.4, it is assumed that the formal parameter of `op2` is of type **`nat`**.

If a test is considered type correct the runtime library will check that the pre condition of the `Call` statement is met, which is checked using the `meetsPreCond` method. The `meetsPreCond` method returns **`true`** by default, which corresponds to the situation where no pre condition is defined. If either the `isTypeCorrect` or `meetsPreCond` method yield false the test is `INCONCLUSIVE`. Otherwise the runtime library proceeds by calling the `execute` method, which evaluates the `Call`

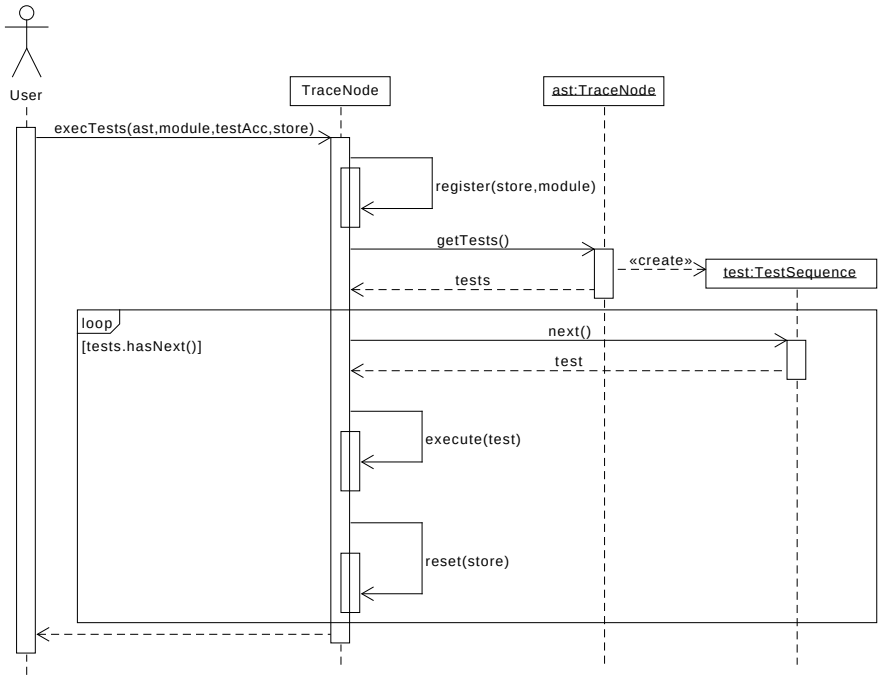


Fig. 2. Execution of a code generated trace, shown using a UML sequence diagram.

Call
`isTypeCorrect() : Boolean` `meetsPreCond() : Boolean` *`execute() : Object`*

Fig. 3. The interface of the `Call` statement node.

```
Call callStm_3 = new Call() {
   public Boolean isTypeCorrect() {
     try {
       //@ assert Utils.is_nat(x);
     } catch (AssertionError e) {
       return false;
     }
     return true;
   }
   public Boolean meetsPreCond() {
     return pre_op2(x);
   }
   public Object execute() {
     return op2(x);
   }
   public String toString() {
     return "op2(" + Utils.toString(x) + ")";
   }
 };
```

Listing 1.4. Implementation of a `Call` statement.

statement and returns the result to the runtime library. This method is abstract and implemented by the code generator when the trace is code generated (see listing 1.4).

4 Case study

4.1 FAD codes

A FAD code is a six digit number, used to identify branches of a retailer. The customer asked us to consider a scenario where FAD codes were obfuscated such that codes were still six digits, still unique per branch, and the entire 0-999999 value range was still available. This is equivalent to creating a permutation on the list of all possible FAD codes. But the obfuscation of a FAD code also needed to be a lightweight calculation, rather than a lookup in a large table, for example.

The design team thought that a permutation could be defined using an injective map of the 0-9 digits onto themselves, but with no digit mapping to itself. If that map was then used to transform the individual digits of a FAD code, then it was believed that the overall set of FAD codes and their transformations would itself form an injective map, defining a permutation. This is intuitively true, but it was not considered “obvious”. To investigate whether it did meet the requirements, a VDM model was created that defined FAD codes and the injective digit map. It was then stated that if the map was applied to every possible FAD code, then the set of obfuscated FAD codes would meet the requirements. Relevant excerpts from the VDM model are shown in listing 1.5.

```
values
  SIZE = 6; -- FAD code size
  MAX = 10 ** SIZE - 1; -- The highest FAD code
  DM1 : DigitMap = -- Arbitrary digit mapping
   { 1 |-> 9, 2 |-> 8, 3 |-> 7, 4 |-> 6, 5 |-> 0,
    6 |-> 4, 7 |-> 3, 8 |-> 2, 9 |-> 1, 0 |-> 5 };
types
  DigitMap = inmap nat to nat
  inv m ==
    let digits = {0, ..., 9} in
      dom m = digits and rng m = digits
      and forall c in set dom m & m(c) <> c;

  FAD = nat
  inv f == f <= MAX
functions
  convert: FAD * DigitMap -> FAD
  convert(fad, dm) ==
    let digits = digitsOf(fad) in
      valOf([ dm(digits(i)) | i in set inds digits ])
  post RESULT <> fad;
traces
  AllDifferent:
    let fad in set {0, ..., MAX} in
      convert(fad, DM1);
```

Listing 1.5. Excerpts from the FAD code VDM model

The trace in listing 1.5 has no combination of cases, but still generates one million test cases when `SIZE` is set to six. The validity of each test is checked by the fact that the digit map `DM1` must meet its constraints, and when applied to every FAD code that map must produce an obfuscated result which meets the `convert` post conditions – that it is a different value.

One could write a trace which applies every possible injective map to the entire set of FAD codes to test that every case produces a permutation. However, with one million FAD codes per map, this would be intractable. This illustrates the point that although trace expansion is useful, combinatorial explosions can easily limit the size of the traces that can be executed.

4.2 Performance results

To analyse the performance gained by using code generated traces, the trace in listing 1.5 was executed using different VDM tools, for FAD codes consisting of up to six digits. These VDM tools exhibit different performance characteristics in terms of execution time and memory consumption due to the way they expand the trace. The memory consumption is an indication of how well a tool scales for large test collections. For example, when the memory consumption of a tool approaches the maximum

amount of memory available, the execution time will increase significantly. In the worst case the tools run out of memory and crash.

Each tool was run twice for each FAD code size. The first run was used to measure execution time. The second run was used to confirm that the tool did not suffer from memory starvation, which would yield a misleading execution time. Checking the memory consumption was performed using a separate run to avoid affecting the execution time. The execution times are those reported by the tools. The memory consumption was analysed using the `/usr/bin/time` tool available on Ubuntu Gnome 15.10 (wily). This tool will report the maximum resident set size for a process, i.e. the maximum amount of memory allocated by the process that is stored in RAM during execution. This is not an accurate measure of the actual memory consumption but it gives an idea of whether the VDM tools are suffering from memory starvation.

All the performance measurements were performed on a Fujitsu LIFEBOOK U772 laptop with a 1.7GHz Intel Core i5 processor and 8Gb of memory running a Linux OS (Ubuntu Gnome). The VDM tools were executed on a 64-bit Java 7 virtual machine with a maximum heap size of 5Gb.

The execution times for FAD codes of sizes one through six are shown in table 1 and visualised using a logarithmic data plot in fig. 4. For the scenario that did not complete, due to the VDM tool running out of memory during trace expansion, the result is specified as "failed". For FAD codes of size six, the maximum resident set sizes for VDMJ was measured to 2.17 Gb, for code generated traces it was 2.31 Gb, whereas no measurement was made for Overture because this tool crashed. Based on the maximum resident set sizes it was confirmed that the tools did not suffer from memory starvation during the trace expansion and execution.

Table 1. Execution times for different VDM tools and FAD code sizes.

Size	VDMJ-3.1.1 [ms]	Overture-2.3.2 extension [ms]	Code Generated [ms]
1	46	124	211
2	465	621	633
3	2,139	3,288	3,217
4	8,692	9,068	29,032
5	35,610	57,999	279,401
6	379,635	failed	2,953,318

5 Discussion

This section discusses the performance results presented in section 4.2. As illustrated using the plot in fig. 4, execution times increase exponentially as more digits are added to a FAD code. This is expected since adding more digits cause an exponential increase in the number of tests generated.

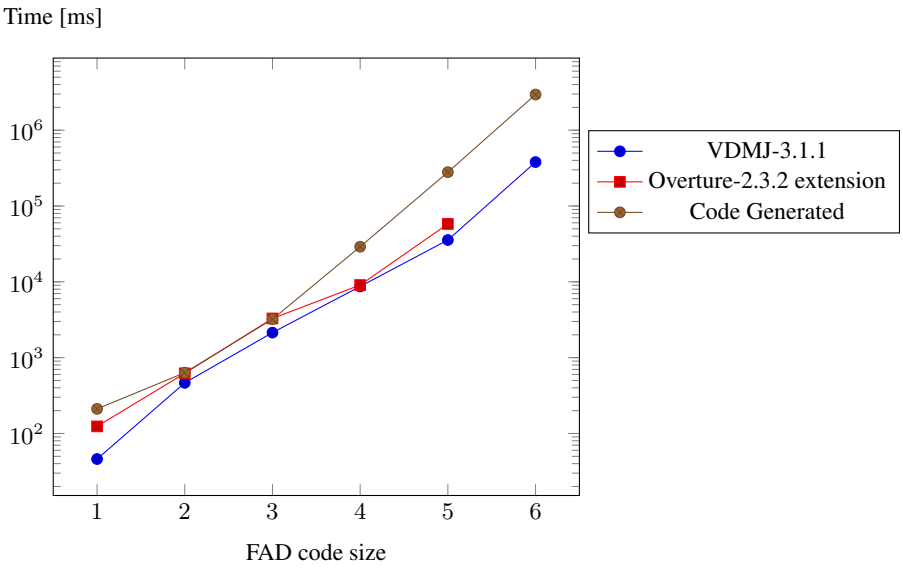


Fig. 4. The execution times in table 1 visualised using a logarithmic data plot.

Overture did not manage to run the one million tests generated for six digit FAD codes because the tool ran out of memory. The code generated trace, on the other hand, completed these tests in over $2,900$ s, or 49.22 minutes. What is surprising about the results obtained using these two tools is that Overture expands and executes the tests significantly faster than the code generated trace, for FAD code sizes smaller than six. Therefore, the only performance gain of the code generated traces is the reduction in memory needed to expand and execute the trace. This is surprising since traces that are run as compiled code are, by intuition, expected to run faster. Comparing the execution time of a code generated trace to that obtained using Overture provides a good indication of the performance that can be gained, since these tools use the same algorithm to expand the trace.

VDMJ completes all one million tests in over 379 s, or 6.33 minutes and is the fastest tool to execute the tests. Compared to Overture, VDMJ manages to complete the tests because it uses a more memory efficient algorithm to expand the trace. Older releases of VDMJ use an algorithm similar to that of Overture. However, very recently VDMJ was released with a new expansion algorithm that addresses some of the performance issues with the old expansion algorithm.

To study the performance overhead posed by OpenJML, we first tried to compile and execute the code generated trace as a normal Java program – without using OpenJML. This is similar to running the code generated trace without checking the generated JML annotations. When this is done, even with the poor expansion algorithm, the one million tests are expanded and executed in 33.94 seconds. If the expansion algorithm is changed to that used by VDMJ, this will most likely be significantly lower. The reason

for this is that VDMJ seems to scale better than Overture for larger test sets, as indicated by the execution times in table 1.

To further analyse the overhead of using OpenJML we tried to remove the JML annotations from the generated code and compile and execute the tests using the OpenJML runtime assertion checker. The point of this is to eliminate the overhead directly related to checking the JML constraints and focus solely on the overhead posed by OpenJML. Although this reduces the number of extra checks that are performed, OpenJML still guards against variables and fields that hold the value `null`, as this is not allowed by default. When the JML annotations are removed, OpenJML expands and executes the one million tests in 11.15 minutes. Ideally, the execution time should be close to that obtained by running the code generated trace without the OpenJML runtime assertion checker (33.94 seconds). This is an indication that OpenJML has a significant influence on the rather disappointing performance results obtained using code generated traces.

To address the performance issues we believe that two things must be done. First, the expansion algorithm must be updated to that used by VDMJ, which is available as open-source. Secondly, we plan to look into other DbC technologies that can be used to support our work (section 6).

6 Conclusion and future plans

In this paper we have shown how VDM traces can be code generated and used to test the system realisation, or some part of it. Code generated traces have potential to allow a larger number of tests to be executed since they are run as compiled code rather than using a VDM interpreter. Our work is implemented as an extension of Overture's Java code generator, which translates a VDM-SL model to a Java program annotated with JML derived from the VDM constraints. When the code generated trace is expanded and executed, using a JML tool, the code generated version of the VDM specification is validated against the JML annotations.

In the FAD code case study, code generated traces allow more memory efficient expansion and execution of traces, compared to using the Overture interpreter. However, we still regard our current performance results as disappointing. Especially because the FAD code trace executes much faster with VDMJ compared to the code generated version of the trace. In order for code generated traces to execute faster than traces interpreted using VDMJ, we believe that two things must be addressed. First, the algorithm used for the expansion must be improved, for example, by using that of VDMJ. Secondly, there are also indications that the performance issues are directly related to the use of OpenJML.

Looking forward, we plan to investigate other technologies that can support our work and help us achieve better performance results. One way to ensure that the contracts and type constraints, as specified in VDM, hold across the translation, is by adding extra Java checks to the generated code – without using a particular DbC technology. Although this allows us to control exactly when these checks are triggered, the separation between specification and code becomes less clean. One DbC technology that is worth investigating, as an alternative to OpenJML, is the .NET-based technology, Microsoft Code Contracts [27, chapter 15]. Compared to JML, which uses a dedicated

syntax for program specification, Code Contracts provides its features via libraries to support all languages within the .NET framework. JML and Code Contracts share many of the same concepts although their semantics sometimes differ. Changing our work to use another DbC technology therefore requires new rules for representing VDM constraints in the generated code. However, due to the similarities between Java and C#, we expect the approach used to code generate traces to be readily reusable. In a C++ context another DbC technology that is worth investigating is the Contract++ library [4], which has been accepted into Boost [26]. On a longer term, contracts may also be a native feature of C++17 [8].

Acknowledgements We would like to thank the anonymous referees for valuable input on this work.

References

1. Battle, N.: VDMJ website. `https://github.com/nickbattle/vdmj` (2016)
2. Brummayer, R., Lonsing, F., Biere, A.: Automated Testing and Debugging of SAT and QBF Solvers. In: Strichman, O., Szeider, S. (eds.) Theory and Applications of Satisfiability Testing – SAT 2010. pp. 44–57. Springer, Edinburgh, UK (Jul 2010)
3. Cok, D.: OpenJML: JML for Java 7 by Extending OpenJDK. In: Bobaru, M., Havelund, K., Holzmann, G., Joshi, R. (eds.) NASA Formal Methods, Lecture Notes in Computer Science, vol. 6617, pp. 472–479. Springer Berlin Heidelberg (2011), `http://dx.doi.org/10.1007/978-3-642-20398-5_35`
4. Contract++ website. `https://sourceforge.net/projects/contractpp/` (2016)
5. Couto, L.D., Larsen, P.G., Hasanagić, M., Kanakis, G., Lausdahl, K., Tran-Jørgensen, P.W.V.: Towards Enabling Overture as a Platform for Formal Notation IDEs. In: Proceedings of the 2nd Workshop on Formal-IDE (F-IDE) (Jun 2015)
6. Couto, L.D., Tran-Jørgensen, P.W.V., Coleman, J.W., Lausdahl, K.: Migrating to an Extensible Architecture for Abstract Syntax Trees. In: Proceedings of the 12th Working IEEE / IFIP Conference on Software Architecture (May 2015)
7. Couto, L.D., Tran-Jørgensen, P.W.V., Lausdahl, K.: Principles for Reuse in Formal Language Tools. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC) (Apr 2016)
8. Thoughts on C++17. `http://www.infoq.com/news/2015/04/stroustrup-cpp17-interview` (2016)
9. Dick, J., Faivre, A.: Automating the Generation and Sequencing of Test Cases from Model-Based Specifications. In: Woodcock, J.C.P., Larsen, P.G. (eds.) FME'93: Industrial-Strength Formal Methods. pp. 268–284. Formal Methods Europe, Springer-Verlag (Apr 1993)
10. Fitzgerald, J., Larsen, P.G.: Modelling Systems – Practical Tools and Techniques in Software Development. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, Second edn. (2009), ISBN 0-521-62348-0
11. Gamma, E., Helm, R., Johnson, R., Vlissides, R.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series, Addison-Wesley Publishing Company (1995)
12. Holzmann, G.J.: The model checker SPIN. IEEE Transactions on Software Engineering 23(5) (May 1997)

13. ISO: Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language (Dec 1996)
14. Jones, C.B.: Scientific Decisions which Characterize VDM. In: Wing, J.M., Woodcock, J.C.P., Davies, J. (eds.) FM'99 - Formal Methods. pp. 28–47. Springer-Verlag (1999)
15. Jørgensen, P.W.V., Larsen, M., Couto, L.D.: A Code Generation Platform for VDM. In: Battle, N., Fitzgerald, J. (eds.) Proceedings of the 12th Overture Workshop. School of Computing Science, Newcastle University, UK, Technical Report CS-TR-1446 (Jan 2015)
16. The JUnit website. http://www.junit.org (2016)
17. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The Overture Initiative – Integrating Tools for VDM. SIGSOFT Softw. Eng. Notes 35(1), 1–6 (Jan 2010), http://doi.acm.org/10.1145/1668862.1668864
18. Larsen, P.G., Lausdahl, K., Battle, N.: Combinatorial Testing for VDM. In: Proceedings of the 2010 8th IEEE International Conference on Software Engineering and Formal Methods. pp. 278–285. SEFM '10, IEEE Computer Society, Washington, DC, USA (Sep 2010), http://dx.doi.org/10.1109/SEFM.2010.32, ISBN 978-0-7695-4153-2
19. Larsen, P.G., Lausdahl, K., Battle, N., Fitzgerald, J., Wolff, S., Sahara, S., Verhoef, M., Tran-Jørgensen, P.W.V., Oda, T.: VDM-10 Language Manual. Tech. Rep. TR-001, The Overture Initiative, www.overturetool.org (Apr 2013)
20. Lausdahl, K., Larsen, P.G., Battle, N.: A Deterministic Interpreter Simulating A Distributed real time system using VDM. In: Qin, S., Qiu, Z. (eds.) Proceedings of the 13th international conference on Formal methods and software engineering. Lecture Notes in Computer Science, vol. 6991, pp. 179–194. Springer-Verlag, Berlin, Heidelberg (Oct 2011), http://dl.acm.org/citation.cfm?id=2075089.2075107, ISBN 978-3-642-24558-9
21. Leavens, G.T., Cheon, Y.: Design by Contract with JML (2005), ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf, draft, available from jmlspecs.org.
22. Ledru, Y., du Bousquet, L., Maury, O., Bontron, P.: Filtering TOBIAS Combinatorial Test Suites. In: Wermelinger, M., Margaria-Steffen, T. (eds.) FASE 2004. pp. 281–294. LNCS 2984, Springer-Verlag Berlin Heidelberg (2004)
23. Ledru, Y., Dadeau, F., du Bousquet, L., Ville, S., Rose, E.: Mastering Combinatorial Explosion with the Tobias-2 Test Generator. In: ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. pp. 535–536. ACM, New York, NY, USA (2007)
24. Nie, C., Leung, H.: A Survey of Combinatorial Testing. ACM Comput. Surv. 43(2) (Feb 2011), http://doi.acm.org/10.1145/1883612.1883618
25. The Overture tool website. http://www.overturetool.org/ (2016)
26. Schling, B.: The Boost C++ Libraries. XML Press (2011)
27. Skeet, J.: C# in Depth, Second Edition. Manning Publications (2010)
28. Tran-Jørgensen, P.W.V., Larsen, P.G., Leavens, G.T.: Automated translation of VDM to JML annotated Java (Jan 2016, submitted to the International Journal on Software Tools for Technology Transfer (STTT))
29. Yu, L., Lei, Y., Kacker, R.N., Kuhn, D.R.: ACTS: A combinatorial test generation tool. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation. pp. 370–375 (Mar 2013)

Towards integration of Overture into TASTE

T. Fabbri [1], M. Verhoef [2], V. Bandur [3], M. Perrotin [2], T. Tsiodras [2], P.G. Larsen [3]

[1] Dept. of Information Engineering, University of Pisa, Italy
[2] European Space Agency, ESTEC, Noordwijk, Netherlands
[3] Department of Engineering, Aarhus University, Denmark
tommaso.fabbri@for.unipi.it, marcel.verhoef@esa.int,
victor.bandur@eng.au.dk, maxime.perrotin@esa.int,
thanassis.tsiodras@esa.int, pgl@eng.au.dk

Abstract. Both TASTE and Overture have successfully demonstrated that method integration is a very promising strategy to create robust and effective tool suites. We show how the automated generation of bidirectional translation functions between VDM and ASN.1 type and value definitions allows the smooth integration of C-code generated from VDM models into TASTE. This enables rapid prototyping and early design validation directly in the target environment, while seamlessly interacting with other parts of the system specified in other notations supported by TASTE.

Keywords: Overture, TASTE, VDM, ASN.1, code generation

1 Introduction

The introduction of formal techniques in an engineering setting is notoriously difficult. Formal methods are typically focused on performing analysis on abstract models of a system, while traditional software engineering is aimed at producing source code that is usually much more detailed. Moving from the former to the latter, requires either to lower the level of detail in the formal models, or relying on translation techniques to do this (semi-) automatically, or by using a combination of both. Automatic generation of source code from formal models is certainly not new, but one of the key issues that hampers their adoption in practice is the fact that it is usually an "all or nothing" approach, which complicates the integration with other system software artifacts, i.e. legacy code, required run-time libraries, device drivers and the operating system.

In previous work [14], we explored an approach consisting in integrating a front-end formal modelling tool, called Overture [8], with a back-end software integration platform, called TASTE [6]. We show in this paper how such a tool integration can be achieved, providing us with a very powerful tool chain that allows the production of high quality executables created from a heterogeneous set of models and other software artifacts.

We first provide an overview of the TASTE environment in Section 2, followed by a short recap of the involved Overture features in Section 3, as we expect that the reader is already fairly familiar with the latter. We then discuss how the integration of the two tools is achieved in Section 4, which is demonstrated on a small case study presented in

Section 5. Finally, we draw some conclusions from our work in Section 6 and discuss how these results can be taken further.

2 The TASTE tool chain

TASTE is an acronym and stands for The ASSERT Set of Tools for Engineering, referring to the European FP6 project Automated proof based System and Software Engineering for Real-Time applications (ASSERT), which ran from September 2004 to December 2007 [5], from which the development philosophy and tool chain originate. Since then, TASTE has evolved into an active open-source initiative, dedicated to the development of high integrity distributed real-time systems [4].

TASTE addresses the problem of developing complex distributed systems, by following a strict heterogeneous model-based approach, supported by high levels of automation. This allows the user to concentrate on building the functionality for each system component in the language of choice that is most appropriate. Advanced code generation and build automation ensures consistency at each development step, removing tedious and error prone manual tasks that often are the root cause of integration and test problems. This philosophy is quite fitting with modern agile development and continuous integration approaches, however these are not at all common in this domain.

In order to achieve that goal, TASTE has adopted mature programming and modelling languages based on open standards with long-term open source and commercial support. At its core, TASTE relies on two formal modelling techniques:

AADL [3] captures the overall system architecture: the main components and their internal structure, the deployment and dynamic behaviour of these components, their interfaces (see [3], used in the TASTE Interface and Deployment Views).
ASN.1 [4] plays a pivotal role in TASTE, as it is used to provide bidirectional translations between all types used to exchange data between the constituent models (see [4], used in the TASTE Data View).

The tool chain ensures that native data types living in one model can be transfered vice versa to any other model without loss of fidelity, while enforcing all consistency requirements, such as invariants. One of the unique selling points of this approach is that the formal definition of the data type in ASN.1 and the physical encoding of the associated value are fully independent, which allows the user to change the value encoding without the need to change the interface of the application that relies on that data type. This provides both flexibility as well as the means to optimize the implementation, even in very late stages of the design. ASN.1 is both the Swiss army knife and super glue of TASTE.

For specifying system behaviour, the user has a wide range of techniques at his disposal. For example, SDL [11] can be used to specify state machines, and interfaces can be generated to directly include code generated from SCADE and Simulink. Also the inclusion of hand-written or legacy code (for example written in C and Ada) is possible. And finally, hardware/software co-design is supported by the automatic generation of both device drivers in C and for the counterpart hardware interfaces in VHDL.

[4] The main project web-site is `http://taste.tuxfamily.org`

TASTE allows to seamlessly integrate all these artifacts into a set of executables, compliant to the defined distributed system architecture in AADL. It relies on a portable and light-weight middleware called PolyORB-HI, which provides the portability abstraction across a wide range of target operating systems such as Windows, Linux, Xenomai (real-time Linux) and RTEMS. PolyORB-HI has been designed with high integrity applications in mind (i.e. ensuring low and deterministic performance overhead, low memory footprint, no dynamic memory management and so on). TASTE will produce binary application images that can be directly downloaded and executed on a wide range of embedded targets, emulators and simulators. Switching from one platform to another is merely a matter of setting a few configuration parameters and rebuilding the application. Last but not least, TASTE provides several features to assist in advanced system validation, such as the automatic generation of test suites and Python test scripts, the import and export of datasets into SQL databases, the creation of graphical user interfaces with features to record, plot and even play back data, for example using Message Sequence Charts [9].

Most if not all notations used within TASTE have a well-defined formal syntax and semantics, which not only allows to seamlessly integrate the artifacts generated from the constituent heterogeneous models, but also to perform upfront analysis. For example, schedulability of the design can already be analysed at the AADL level before even a single behavioral specification is available or the choice for the hardware platform is made. Moreover, consistent design documentation can be generated directly from the models, which is in particular useful for interface control documents. The well defined semantics also enables the integration of other tools and techniques, making TASTE extensible in a coherent and controlled way. This is the key to creating a tool chain that is aimed at improving productivity.

3 The Overture tool and `vdm2c`

Overture is a tool that enables the definition, validation and simulation of VDM models, supporting the three dialects VDM-SL, VDM++ and VDM-RT [8]. VDM-SL is the core language, with object-oriented specification facilities built on top of it as VDM++. Facilities for specifying timing and distributed architectures are in turn built on top of VDM++ as VDM-RT. Note that VDM++ is the dialect of interest in this paper, as the VDM-RT aspects are already provided by TASTE in the Deployment View.

The code generation platform of Overture [7] is being employed to create a C code generator, which is called `vdm2c`, for an executable, object-oriented subset of VDM-RT. Note that the choice for C was made deliberately over C++. At the time of writing this paper, an early prototype of this code generator was available (version 0.0.2), providing support for basic language features. Despite the tool limitations, it has proven to be sufficient to demonstrate our proof of concept.

The C code is generated under the assumption that the original VDM specification has been validated using Overture, at the very least showing that the specification is syntax and type correct. The translator follows the structure of the specification as closely as possible. It implements VDM++ classes as C structures with appropriate function pointer and name mangling mechanisms in order to implement inheritance, and it car-

ries VDM types explicitly. These VDM types are part of a generic support library which is independent of the source VDM model. The fundamental data type of the support library is the C structure `TypedValue` shown near the bottom of Listing 1.

Listing 1. Fundamental code generator data type.

```
typedef enum {
  VDM_INT, VDM_NAT, VDM_NAT1, VDM_BOOL, VDM_REAL,
  VDM_RAT, VDM_CHAR, VDM_SET, VDM_SEQ, VDM_MAP,
  VDM_PRODUCT, VDM_QUOTE, VDM_RECORD, VDM_CLASS
} vdmtype;

typedef union TypedValueType {
  void* ptr;             // VDM_SET, VDM_SEQ, VDM_CLASS,
                         // VDM_MAP, VDM_PRODUCT
  int intVal;            // VDM_INT and INT1
  bool boolVal;          // VDM_BOOL
  double doubleVal;      // VDM_REAL
  char charVal;          // VDM_CHAR
  unsigned int uintVal; // VDM_QUOTE
} TypedValueType;

struct TypedValue {
  vdmtype type;
  TypedValueType value;
};

struct Collection {
  struct TypedValue** value;
  int size;
};
```

While it is easy to understand how basic VDM types such as **nat** and **char** can be accessed in a value of type **struct** `TypedValue`, structured values such as **set** and **seq** require further explanation. The elements of a value of this type are simply stored as an array inside a value of type **struct** `Collection`. The `size` field records how many elements the collection contains. Naturally, nested types are accommodated. VDM records are treated as VDM classes with no functions or operations.

4 The Integration between Overture and TASTE

The initial step of the integration between TASTE and Overture looked at the interactive simulation of a simple heterogeneous system model [12]. The main goal of this experiment was to define reactive system behaviour as a state machine using the Specification and Description Language (SDL) and perform more complex data manipulation algorithms in VDM, as a means of achieving separation of concerns. The software architecture is presented schematically in Figure 1. The SDL modeller and simulator in

TASTE, OpenGEODE, interacts directly with the Overture interpreter through a TCP socket connection, which exposes the standard remote control API of Overture. This allows OpenGEODE to call the Overture interpreter directly in order to execute specific operations in the VDM model, whereby OpenGEODE converts all TASTE internal data types to and from their VDM counterparts. The TASTE Abstract Syntax Notation One (ASN.1) compiler, asn1scc [13], was extended to translate the ASN.1 data type definitions in OpenGEODE into their VDM counterpart in Overture. This already provided an additional level of consistency, and demonstrated that a translation from ASN.1 to VDM was feasible for all types supported in TASTE. Examples of this translation is provided later in this section.

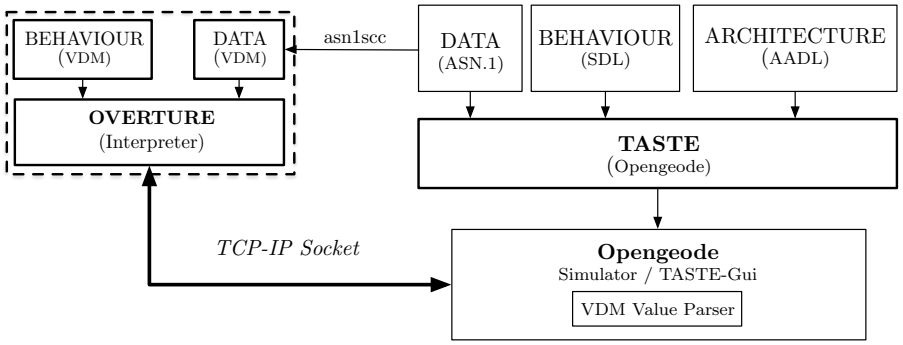


Fig. 1. Initial experiments for integrating SDL and VDM models

At this stage, the integration has involved only a few features of the TASTE tools: more precisely, a) the built-in behavioural modelling facilities (OpenGEODE) and b) the use of the ASN.1 compiler to ensure consistency of the data type definitions. This enabled the interactive simulation shown in [12] which is great for early system validation. However, the TASTE toolset is primarily aimed at the development of embedded, real-time systems: in particular TASTE is able to automatically assemble, glue together and deploy the final software system on a real target; none of these features have been exploited in the initial experiment. In order to have VDM as a new modelling language inside TASTE, a deeper integration is necessary.

As presented in Section 3, the Overture toolset currently supports a prototype code generator for the C language: vdm2c. The main idea is to allow the automatic integration of the generated C-code from Overture with other software artifacts comprising functions coded in other languages through the use of TASTE. Figure 2 presents the new integrated architecture: the compatibility of data exchanged between all software artifacts is ensured by the ASN.1 compiler. Finally, TASTE compiles and links all the pieces together and deploys the generated executable on the target platform.

To realize such integration, it is necessary to develop what makes the two generated pieces of code communicate. To have a better understanding, consider the following example where the function A (i.e. generated from Matlab-Simulink) wants to call the

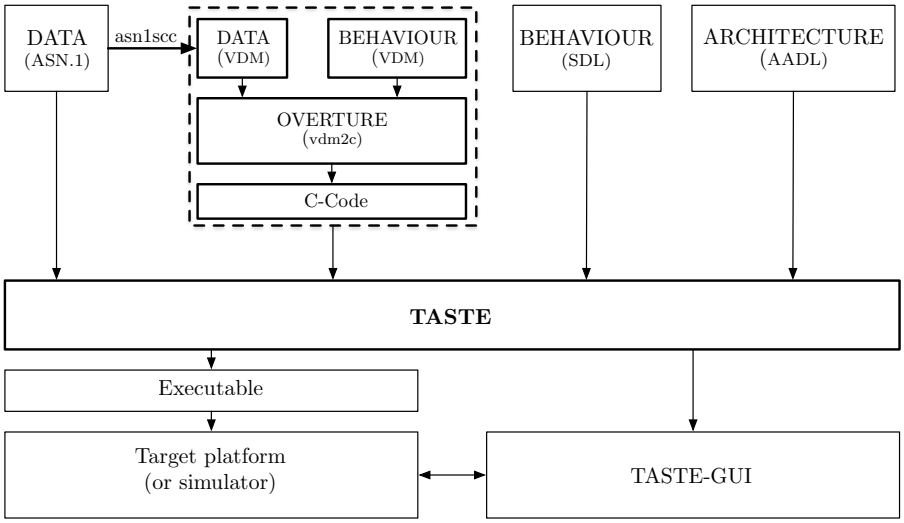


Fig. 2. TASTE Overture integrated architecture.

function `B` (generated from VDM) passing the parameter `c` of a given type. As already explained, TASTE relies on the ASN.1 language to define the exact content of the data types of the function parameters in a way that is independent from a specific implementation language. If the function `B` expects the parameter `c` of type `TypeC`, an 8-bit integer, the ASN.1 definition will include:

Listing 2. Example of a type definition using ASN.1.

```
TypeC ::= INTEGER (0..255)
```

At this stage, we would look at the `TypeC` type and find the best match in VDM language to represent it; then we would look at how `vdm2c` translates the VDM `TypeC` to C type. The same thing has to be done also in the `A` side and make sure they fit. Otherwise, we would write a function `Convert_TypeC_from_Matlab_to_VDM(...)` doing the conversion job. Despite being technically easy to perform, this is an error-prone and long process as soon as there are many functions to develop and maintain, in particular as these interfaces are likely to evolve over time. TASTE has tools that implements this whole process in an automated way, in the TASTE terminology:

`A_mapper` - starting from the ASN.1 types, the Data Modelling Toolchain [1] and/or the compiler templating facilities [13] can generate semantically-equivalent types. The `asn1scc` compiler is dedicated to safety-critical systems, and is able to generate optimized Ada and C code, but also customizable through template files for enabling support to other languages (e.g. VDM) [10].
`B_mapper` - generating the translation functions at code level that convert the data types obtained by `vdm2c` code generator from and to the `C` data types generated by

the ASN.1 compiler, as well as the piece of glue code that wraps the VDM function call with the translated parameter.

In this context, the Data Modelling Toolchain generates the VDM types starting from the ASN.1 data types definition, and currently supports all TASTE allowed data types. At the current time, the `B_mapper` supports simple data types such as `INTEGER`, `BOOLEAN`, `REAL` and `ENUMERATED` and complex types like `SEQUENCE OF`. The support of more structured data types, like `CHOICE` (the VDM union type) and `SEQUENCE` (the VDM record type) is under development [5]. Through the definition of these two components, the `A_mapper` and `B_mapper`, the VDM language is integrated into TASTE to create working prototypes. The Section 5 demonstrates these capabilities on a small case study.

5 Case Studies

First, we recall the data definition reported in Listing 2; through the invocation of the `asn1scc` compiler, the semantically equivalent VDM types are generated as reported in Listing 3, where ASN.1 constraints are translated into VDM invariants.

```
types
public TypeC = int
  inv x >= 0 and x <= 255
```

Listing 3. Generated VDM type starting from the ASN.1 definition of Listing 2.

The ASN.1 compiler `asn1scc` and `vdm2c` are both used for generating the corresponding `C` code. The ASN.1 compiler generates a `C` definition where `TypeC` is an alias of integer primitive type `asn1Sccint`, as shown in the Listing 4:

Listing 4. Native C data type, generated by the `B_mapper`.

```
  typedef asn1Sccint TypeC;
```

The `vdm2c` *maps* the defined `TypeC` into the `struct TypedValue` (see Listing 1), where the field `type` takes the value `VDM_INT` and the field `value` takes the `intVal` representation. The `B_mapper` is now used for generating functions executing the conversion between the two `C` codes obtained; recalling the example the conversion from the `asn1scc TypeC` to the VDM `TypedValue` and vice versa, the generated code is shown in Listing 5.

Listing 5. Generated convert functions by the `B_mapper`.

```
void Convert_TypeC_from_VDM_to_ASN1SCC
  (asn1SccMInt *ptrASN1SCC, TVP VDM)
{
  (*ptrASN1SCC) = (asn1SccSint)((VDM))->value.intVal;
}
```

[5] These features are currently also not supported by vdm2c.

```
void Convert_TypeC_from_ASN1SCC_to_VDM
  (TVP *ptrVDM, const asn1SccMInt *ptrASN1SCC)
{
  (*ptrVDM) = newInt((*ptrASN1SCC));
}
```

The generated functions are based on straightforward conversion as in the translation from VDM to `asn1scc` standard types, or based on the support library of `vdm2c` (see the call to `newInt`) for the definition of `TypedValue` variables. Similar functions are also automatically generated for more complex types, such as sequences, requiring iteration over the respective internal structures in order to perform the mapping. Now the basic elements are in place, we can present a small case study that demonstrates how the tools can be used in practice.

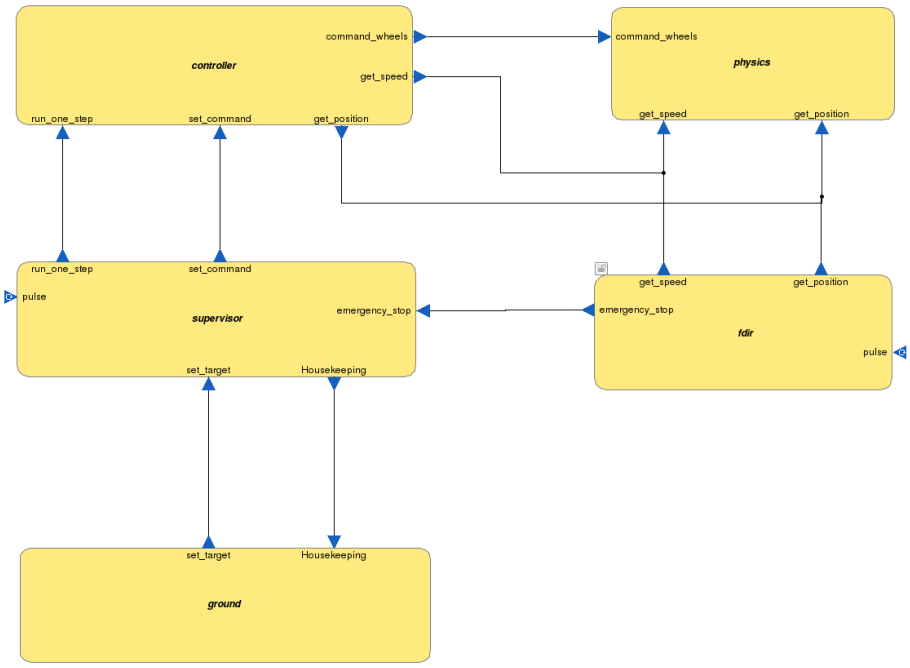


Fig. 3. Case study - TASTE Interface View

Figure 3 presents the interface view of an application consisting of five components. This top-level model, which is specified in AADL, gives an overview of all provided and required interfaces and their interconnections. A target language can be specified for each of the components. For example, the `ground` component is an automatically generated graphical user interface that allows to interact with the system. The `supervisor` is a state machine specified in SDL and the `controller` is specified

in VDM. Note that the `supervisor` has a periodic provided interface called `pulse`, for which the period is specified in the interface view, in this case 1 second. The TASTE middleware will ensure that this interface is called at this rate and that any deadlines specified are also enforced. The SDL specification of the `supervisor` is provided in Figure 4.

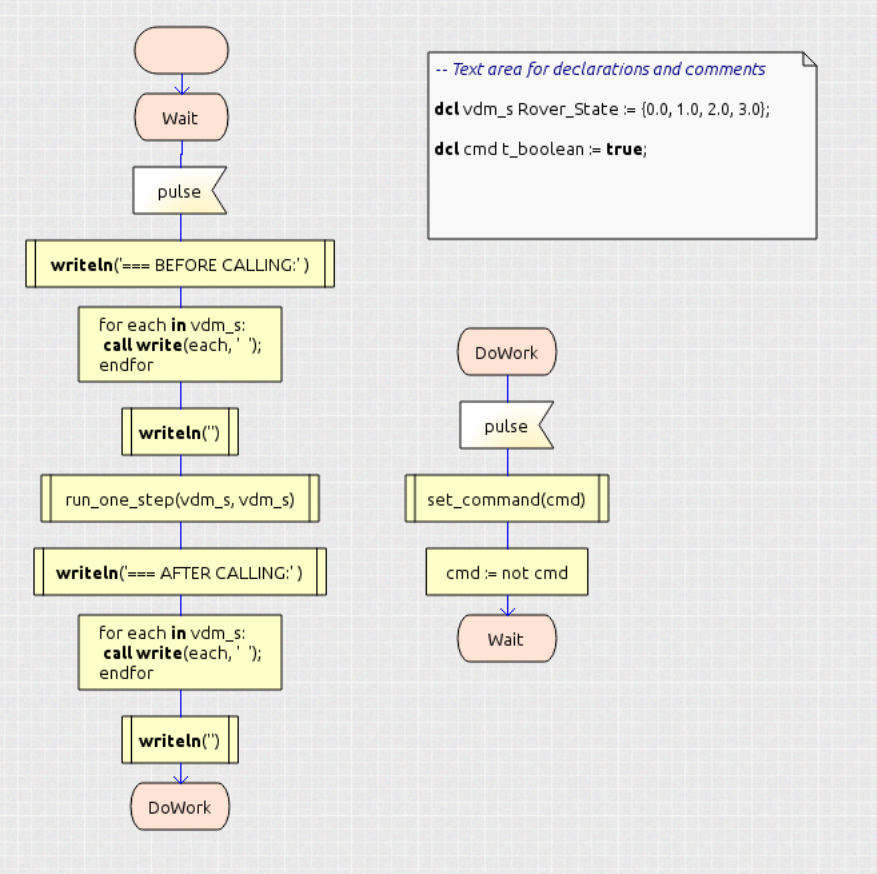


Fig. 4. Case study - TASTE supervisor SDL specification

The `supervisor` waits for the `pulse` event, then prints the `vdm_s` variable, calls the `run_one_step` operation and prints the `vdm_s` variable again. It continues in the $DoWork$ state, calls the `set_command` operation and toggles the `cmd` variable. Finally, it resumes to the $Wait$ state, which repeats the cycle at the next `pulse` event that is received. Note that `run_one_step` and `set_command` are provided interfaces of the `controller`. Also note that this model uses data types as specified in the TASTE data view, as shown in Listing 6.

Listing 6. Case study : type definitions ASN.1 (TASTE data view).

```
TASTE-Dataview DEFINITIONS ::=
BEGIN
IMPORTS T-Boolean FROM TASTE-BasicTypes;
Rover-State ::= SEQUENCE (SIZE(4)) OF REAL (0.0 .. 1000.0)
END
```

Using the `A_mapper` in `asn1scc`, TASTE will generate the following VDM model:

```
class TASTE_Dataview
types
    public Rover_State = seq of real
    inv x == len x = 4
end TASTE_Dataview

class TASTE_BasicTypes
types
    public T_Boolean = bool
end TASTE_BasicTypes
```

Listing 7. Case study: generated VDM data model.

and the following VDM specification template for the provided interfaces:

```
class controller_Interface
operations
  public Startup: () ==> ()
  Startup () is subclass responsibility;

  public PI_run_one_step: TASTE_Dataview`Rover_State ==>
    TASTE_Dataview`Rover_State
  run_one_step (-) == is subclass responsibility;

  public PI_set_command: TASTE_BasicTypes`T_Boolean ==> ()
  set_command (-) == is subclass responsibility;

end controller_Interface
```

Listing 8. Case study: generated VDM data model.

which can be conveniently modified to provide the following behavior:

```
class controller
  is subclass of controller_Interface

instance variables
  updateState : bool := false
```

```
operations
  public Startup: () ==> ()
  Startup () == updateState := true;

  public PI_run_one_step: TASTE_Dataview`Rover_State ==>
    TASTE_Dataview`Rover_State
  PI_run_one_step (vdm_state) ==
    if updateState then
      ( dcl newState : TASTE_Dataview`Rover_State :=
          [ vdm_state(1) + 1, vdm_state(2) + 2,
            vdm_state(3) + 3, vdm_state(4) + 4];
        return newState )
    else return vdm_state;

    public PI_set_command: TASTE_BasicTypes`T_Boolean ==> ()
    PI_set_command (cmd) == updateState := cmd;

end controller
```

Listing 9. Case study: user specified VDM model.

The specified behavior shown in this example is trivial of course, but it just serves the purpose of showing the process. In principle, the VDM model can be arbitrarily complex. The Overture `vdm2c` code generator can now be used to generate the C-code directly from the model shown in Listing 9. For conciseness, we do not include the generated code here, but it is available on-line, as part of the TASTE example suite [2]. The final step is the glue code and the ASN.1 conversion functions, as generated by the `B-mappers`. The ASN.1 conversion functions follow the same scheme as shown in Listing 5, but the additional glue code to link everything together, is shown below:

Listing 10. Glue code generated by the `B_mapper`.

```
#include "Vdm_ASN1_Types.h"
#include "controller.h"

static TVP controller;

void controller_startup()
{
  controller = _Z10controllerEV(NULL);
  CALL_FUNC(controller, controller, controller,
    CLASS_controller__Z7StartupEV);
}

void controller_PI_run_one_step (
  const asn1SccRover_State *IN_vdm_state,
  asn1SccRover_State *OUT_feedback )
{
  TVP ptr_vdm_state = NULL;
```

```
  Convert_Rover_State_from_ASN1SCC_to_VDM
    (&ptr_vdm_state, IN_vdm_state);
  TVP vdm_OUT_feedback;
  vdm_OUT_feedback = CALL_FUNC
    (controller, controller, controller, 1, ptr_vdm_state);
  Convert_Rover_State_from_VDM_to_ASN1SCC
    (OUT_feedback, &vdm_OUT_feedback);
}

void controller_PI_set_command(const asn1SccT_Boolean *IN_cmd)
{
    TVP ptr_cmd = NULL;
    Convert_T_Boolean_from_ASN1SCC_to_VDM(&ptr_cmd, IN_cmd);
    CALL_FUNC(controller, controller, controller, 2, ptr_cmd);
}
```

All the artifacts shown in this section are automatically collected by TASTE and built into an executable as specified in the TASTE Deployment View, which is shown in Figure 5, here indicating that all components are running on a single Intel x86 based Linux 32-bit operating system. The output of the running executable is shown in Listing 11, which demonstrates that the heterogeneous model works as specified.

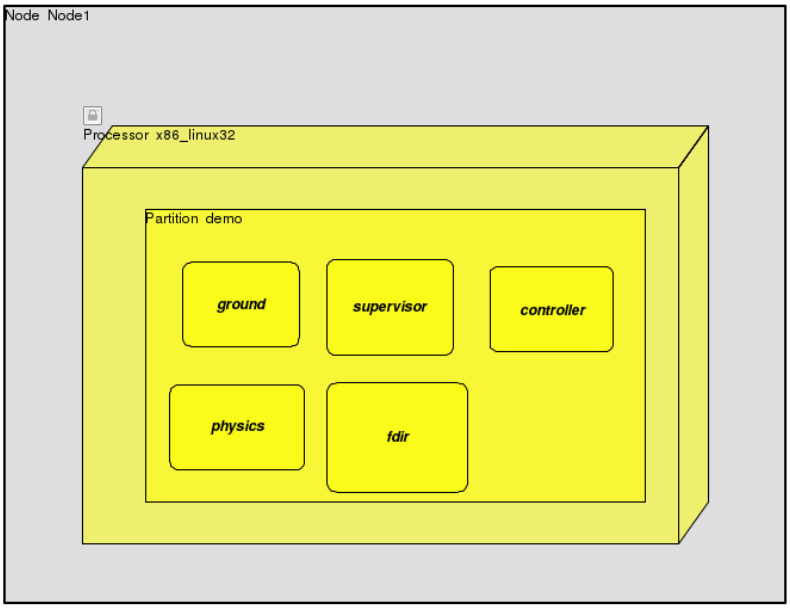


Fig. 5. Case study: simplified TASTE deployment view

Listing 11. Debug logging of the running application.

```
=== BEFORE CALLING: 0   1   2   3
=== AFTER CALLING:  1   3   5   7
=== BEFORE CALLING: 1   3   5   7
=== AFTER CALLING:  2   5   8   11
=== BEFORE CALLING: 2   5   8   11
=== AFTER CALLING:  2   5   8   11
=== BEFORE CALLING: 2   5   8   11
=== AFTER CALLING:  3   7   11   15
=== BEFORE CALLING: 3   7   11   15
=== AFTER CALLING:  3   7   11   15
=== BEFORE CALLING: 3   7   11   15
=== AFTER CALLING:  4   9   14   19
```

6 Conclusions and future work

The paper has presented the integration procedure of the VDM language as new technology for designing functionalities inside a TASTE environment. Such implementation involves the code generator `vdm2c` from the Overture side, the `asn1scc` compiler and tools from the TASTE side able to generate pieces of *glue code* to make heterogeneous technologies communicate. The case studies demonstrate how functionalities developed in the VDM language are mixed together with components developed in other languages to create a unique executable to be deployed on a target platform or tested on a simulator. At the current time only ASN.1 basic types (like `INTEGER, REAL, BOOLEAN, ENUMERATE`) and `SEQUENCE OF` basic types are completely supported.

But even in this limited setting, it has been clearly demonstrated that this kind of integration is only feasible if the steps are completely automated, and this is exactly what our solution now offers. Most of the complexity shown in the previous section is completely hidden from the user, allowing to concentrate on his main task: to specify all the system components in the most convenient formalism available in TASTE, as shown in Figure 3, Figure 4, Listing 6, Listing 9 and Figure 5. Next short term developments will be focused on the completion of the support for structured data types like `SEQUENCE` and `CHOICE`, following the on-going development of the `vdm2c` compiler, which is work in progress at this stage.

Our work has also shown that a bi-directional mapping between VDM and ASN.1 is indeed feasible. The existing Overture interpreter could therefore be extended with a feature to directly communicate to other outside tools, much akin to our initial experiments described in Section 4, but then using a communication channel based on the exchange of ASN.1 values, using a configurable encoding scheme. This would allow the coupling of Overture to other tools without the need for an external data converter, and enable the interactive use of the Overture analysis features during validation experiments.

Acknowledgments

This work has been supported by the ESA Summer Of Code In Space (SOCIS) 2016 run by the European Space Agency. The work on the VDM to C code generation from the Overture tool is supported by the INTO-CPS project (Horizon 2020, 664047)[6]. The authors would like to thank P. W. V. Tran-Jørgensen and K.G. Lausdahl for their help and participation in the development of the `vdm2c` code generator. Finally, we would like to thank the anonymous referees for valuable input on this work.

References

1. TASTE Data Modelling Tools. `https://github.com/ttsiodras/DataModellingTools/`
2. TASTE Example Suite. `https://tecsw.estec.esa.int/svn/taste/branches/stable/testSuites/WorkInProgress/Demo-TASTE-VDM/`
3. SAE AS 5506B: Architecture Analysis & Design Language (AADL) (2012), `http://standards.sae.org/as5506b/`
4. ITU X.680-X.693 : Information Technology - Abstract Syntax Notation One (ASN.1) & ASN.1 encoding rules, `http://www.itu.int/rec/T-REC-X.680-X.693-200811-I/en`
5. ASSERT: Automated proof-based System and Software Engineering for Real-Time Systems (5 2012), `http://cordis.europa.eu/project/rcn/71564_en.html`
6. Conquet, E., Perrotin, M., Dissaux, P., Tsiodras, T., Hugues, J.: The taste toolset: turning human designed heterogeneous systems into computer built homogeneous software. In: European Congress on Embedded Real-Time Software (ERTS 2010). Toulouse, France (May 2010)
7. Jørgensen, P.W.V., Larsen, M., Couto, L.D.: A Code Generation Platform for VDM. In: Battle, N., Fitzgerald, J. (eds.) Proceedings of the 12th Overture Workshop. School of Computing Science, Newcastle University, UK, Technical Report CS-TR-1446 (January 2015)
8. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The Overture Initiative – Integrating Tools for VDM. SIGSOFT Softw. Eng. Notes 35(1), 1–6 (January 2010), `http://doi.acm.org/10.1145/1668862.1668864`
9. ITU-T Rec. Z.120 (02/2011) Message Sequence Chart (MSC), `http://www.itu.int/rec/T-REC-Z.120/en`
10. Perrotin, M., Grochowski, K., Verhoef, M., Galano, D., Mosdorf, M., Kurowski, M., Denis, F., Graas, E.: TASTE in action. In: 8th European Congress on Embedded Real Time Software and Systems (ERTS 2016). TOULOUSE, France (Jan 2016), `https://hal.archives-ouvertes.fr/hal-01289678`
11. ITU-T Rec. Z.100 (04/2016) Specification and Description Language - Overview of SDL-2010, `http://www.itu.int/rec/T-REC-Z.100/en`
12. Taste-Team, Overture-Team: Integrating VDM and SDL. http://bit.do/sdl-vdm (2015)
13. Tsiodras, T.: ASN1SCC: An open source ASN.1 compiler for embedded systems. `https://github.com/ttsiodras/asn1scc`
14. Verhoef, M., Perrotin, M.: TASTE for Overture to keep SLIM. In: Proceedings of the 13th Overture Workshop. pp. 132–139. Center for Global Research in Advanced Software Science and Engineering, National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-Ku, Tokyo, Japan (June 2015), `http://taste.tuxfamily.org/wiki/images/7/7b/Taste-for-overture-verhoef-perrotin.pdf`, GRACE-TR-2015-06

[6] `http://into-cps.au.dk/`

Modelling Collaborative Systems and Automated Negotiations

Nico Plat[1], Peter Gorm Larsen[2], and Ken Pierce[3]

[1] West IT Solutions, The Netherlands
[2] Aarhus University, Department of Engineering, Denmark
[3] Newcastle University, School of Computing Science, UK

Abstract. Collaboration between different Traffic Management Systems (TMSs) is virtually non-existent. A TMS is a cyber-physical system composed of software and hardware systems under the control of a road authority. Existing road infrastructure is not used in an optimal way, because each road authority tries to achieve its own local goals: minimal congestion and pollution. The performance of the network as a whole is not considered, nor are the interests of individual road users, who travel across boundaries controlled by different TMSs. The TMSs involved do not attempt to help one another to get a better overall result. In this paper we show that collaboration between TMSs can be introduced, by modelling a system of automated negotiations using the Overture toolset. A demonstrator has been made which consists of a VDM model of two TMSs that interacts with a simple traffic simulator. The VDM model receives traffic information from the simulator, then the TMSs negotiate on which traffic control measures to use. The chosen measures are communicated back to the simulator, allowing the effect of the traffic measures to be seen.

Keywords: cyber-physical systems, Overture, VDM, traffic management, automated negotiation, collaborative architectures

1 Introduction

Many countries have dense road networks and significant traffic problems. The flow of traffic on roads is managed by a series of Traffic Management Systems (TMSs), owned and controlled by various local and national authorities. A TMS consists of a collection of distributed systems and devices, usually installed along the roadside, and control logic that decides how these devices operate. Devices can be sensors that collect traffic data, such as cameras, radar detection systems, induction loops, and actuators that give instructions to road users via signs and signals. Current TMS architectures are usually controlled centrally from regional control centres. A low degree of collaboration hinders efficient management of traffic problems that straddle boundaries between authorities, because TMSs cannot communicate between regions and may have competing goals for traffic flow. While cooperation between various road authorities from a governance point of view has improved recently, a technical infrastructure (including the application software) that facilitates collaboration between TMSs, is not yet in place.

The TEMPO[4] project addressed the problem of disconnected TMSs by giving TMS providers collaborative control architectures that engage with each other in automated negotiations. Negotiations are based on policies, which help to reach agreement on which control measures are beneficial for the system as a whole, improving overall network performance.

TEMPO used VDM++ and Overture [6] to perform an initial analysis of both existing traffic management networks and potential collaborative designs, demonstrating the benefits of smart traffic systems. VDM was chosen as way to converge quickly on the core logic in the traffic management domain with limited time and resources. The initial models produced illustrate the core concepts and the possibilities enabled by allowing TMSs to collaborate. It is possible to extend this further to achieve more realistic simulations, for example using either the Crescendo[5] [4] or INTO-CPS[6] technologies [3] to connect the TMS models in VDM to a dedicated traffic simulator. In such extensions it would be necessary to devise a proper protocol for use in negotiations between TMSs. Traffic simulations produce a large amount of numerical data and it is imperative that this data is presented in an understandable way to non-experts, in order to make computer models useful. Overture has been extended with a 2D/3D visualisation library[7] to illustrate the negotiations between TMSs and the overall effect on traffic flow [2].

This paper focuses on the central aspect of the model: the negotiation mechanism used as the basis for collaboration. We start by giving a short introduction in the field of traffic management in Section 2, using the same concepts that will be used in the model itself, in this way making the model more comprehensible. After that a brief overview of the TEMPO project is provided in Section 3. We then explain the idea behind the negotiation process in Section 4, and how this is transformed into a VDM model in Section 5. Afterwards we present initial results of the performance of the model, compared to the original situation, in which no collaboration is facilitated in Section 6. Finally, we conclude the paper and suggest future work in Section 7.

2 Traffic Management

Traffic management involves planning, coordinating, controlling, and organising traffic to achieve efficient and effective use of the existing road capacity. The way traffic management is implemented varies in degree of sophistication, however in all cases the goals are mostly the same:

- Enhancing or optimising the throughput of the network, in order to decrease travellers' journey times;
- Sustainability, ensuring that environmental policies are met;

[4] TEMPO is an acronym for "TMS Experiment with Mobility in the Physical world using Overture". See `http://tempoproject.eu/` for more information.

[5] See `http://crescendotool.org/`.

[6] See `http://into-cps.github.io/`.

[7] The visualisation library is explained further in the latest version of the Overture user manual [8].

– Keeping travellers informed, making sure that travellers get reliable traffic information with respect to their expected journey time and incidents or accidents on the road.

These goals must be achieved with vehicle safety as a vital constraint: making journeys as safe as possible is always a first priority.

In this paper we are only interested in *operational* traffic management[8], i.e. responding to the current traffic situation on (part of) the road network, deciding if the situation requires intervention, and then deploying the instruments available to influence traffic streams or individual vehicles on the network.

To do this one needs:

– *Access to real time traffic data*. Typically this is done using sensors at the roadside or elsewhere. Examples of sensors include induction loops, which measure the speed and density of passing traffic, cameras, and radar systems. Countries like The Netherlands have central repositories in which (soft) real-time[9] traffic data is available from a variety of sources, covering almost the entire Dutch road network[10].
– *Processes for decision making*, to determine whether or not actions need to be taken to influence, steer or guide the traffic streams on the network depending on the current or predicted traffic situation. Decision making can either be done in a Traffic Control Centre (TCC) or along the roadside (local control).
– *Instruments to take action*, to influence traffic using, for example, roadside equipment. At a logical level, we call these *control measures*. Examples include setting up diversions, opening or closing hard shoulders to regular traffic, altering traffic light timings, imposing temporary speed limits and ramp metering (controlling the amount of traffic flowing into an area). At a physical level, such control measures are called actuators. Examples include signs and signals displayed along the roadside or in-car.

A road network will have parts controlled by different TMSs under different ownership. As such, the group of TMSs can be characterised as a System of Systems [10, 11]: it is a system composed of several constituent systems that execute individual control measures on their part of the network, but cannot individually guarantee properties of the whole network without working together. A TMS is also a Cyber-Physical System (CPS) [9], because it consists of distributed, networked physical and software (cyber) components.

Traffic management is traditionally executed by road authorities, where national authorities are typically responsible for a country's motorway network and interurban traffic, and municipalities are responsible for their own urban networks. Other organisations can also influence traffic, and hence act as de facto TMS providers. These include

[8] In addition to operational traffic management we also distinguish tactical traffic management and strategic traffic management. Tactical traffic management addresses planning of resources to be used during operational traffic management and other tactical issues, and strategic traffic management concerns issues related to policy making.

[9] The lowest granularity currently available covers a time interval of one minute, which is suitable for traffic management but not for safety measures.

[10] National Data Warehouse (NDW) for Traffic Information, see `http://www.ndw.nu/en/`.

car manufacturers and manufacturers of navigation equipment, airports and harbours that deal with logistics, and even event organisers that attract large volumes of multi-modal traffic at around the time their events take place.

The means that these parties have to influence traffic vary. In some cases, such as manufacturers of navigation equipment, this is limited to the distribution of traffic information, while official authorities can directly influence traffic through physical measures. Yet all parties can influence traffic on potentially overlapping parts of the road network or across neighbouring parts, for example ramps between urban and motorway networks. Since each party may have individual goals that are conflicting, the final results for individual travellers may not be optimal.

3 The TEMPO project

The TEMPO project was a small scale experiment, in which we attempted to overcome the conflicting interests of TMSs by engaging them in a negotiation process to reach a goal that is acceptable to all parties involved. The concept of "cost" (price) is introduced here to compensate when one TMS needs to give in when global goals get priority over its own goals.

In order to do this we developed a model of the system that treats the road network as a graph, where vehicles move along the edges of the graph in a distributed (but predefined) fashion. The complete model of the system consists of two parts coupled together using the remote access features to the Overture toolset [12]:

- A VDM model that specifies the business logic of multiple TMSs, engaging in a negotiation process to determine which traffic control measures will be taken. We also refer to this model as the "TEMPO engine".
- A traffic simulator written in Java[11], which feeds the TEMPO engine with "traffic situations". Based on the traffic situation, the engine executes calculations (with or without negotiations) resulting in a series of traffic control measures to be applied. These are communicated back to the traffic simulator, which takes the measures into account, performs another round of traffic processing, provides a new traffic situation to the model, and so forth. The simulator provides a simple events language, which allows the user to specify traffic volume for an edge (e.g. simulating rush hours), traffic accidents, and scheduled events such as bridge openings.

The global architecture of the demonstrator is presented in Figure 1.

4 Automated Negotiation

The concept of a "negotiation process" is at the very heart of the TEMPO engine. The idea of working with an automated negotiation process is in itself not new. Beam and Segev published a survey [1] in the mid 90's on the state of the art with respect to automated negotiation. Jennings et al. did the same around the year 2000 [5]. Negotiation

[11] Java was chosen as the implementation language for the simulator for practical reasons. The simulator itself is not in scope of our analysis.

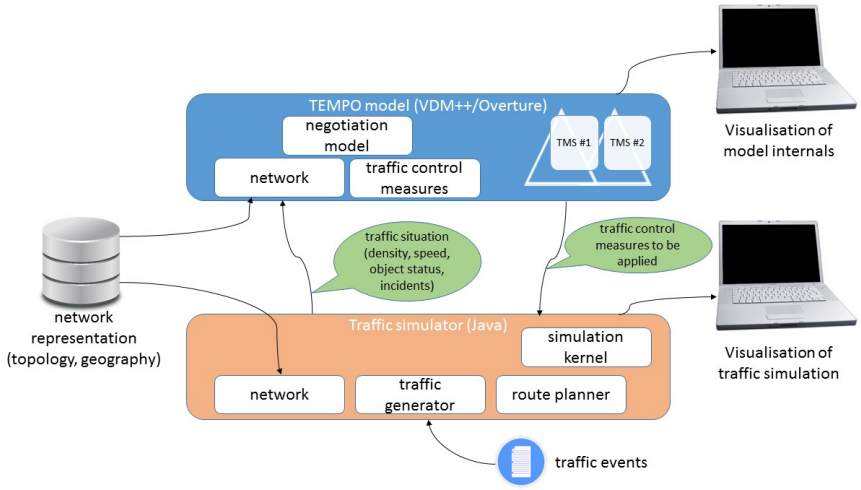


Fig. 1: Architecture of the TEMPO demonstrator

in (electronic commerce) is defined as the process by which two or more parties bargain for mutual intended gain, using the tools and techniques of electronic commerce. Completely automatic negotiations are usually based on intelligent agents. Ontologies (categorising the objects of negotiation such that they are semantically meaningful to software or human agents), strategies for negotiation, negotiation protocols, and decision making models were important research issues at the time. Negotiation support systems are available to help human negotiators make better decisions and negotiate more productively. Online auctions and marketplaces are typical examples of application areas.

The automated negotiation process in TEMPO conceptually uses a rather simple strategy compared to other existing automated negotiation systems[12], although enhancements to make it more powerful are rather straightforward, given the current setup. In that sense, the TEMPO architecture can be regarded as a framework for automated negotiations. Furthermore, automated negotiation as part of a traffic management process is, to our best knowledge, completely new.

The first step in the entire process is not really part of the negotiation itself: it consists of determining whether or not the TMS can "help itself". In order to do this the TMS assesses the traffic situation on all the edges of the network (typically a subnetwork of the overall network) that it controls. A traffic situation typically comprises density and average speeds of the vehicles on the edge, whether or not an incident occurred, and whether or not objects on or close to the network may hinder the traffic (in TEMPO we use the bridges that can be open or closed). Future versions of the traffic

[12] The specification of the negotiation process is less simple though, because of the specific details of the traffic management domain.

situation may include variables like e.g. air pollution which then also can be used as a steering mechanism.

If the TMS has traffic control measures available on a given edge of the network, and if the traffic situation for that edge is such (e.g. a high traffic density, which is a sign of congestion) that measures need to be taken, then that edge is marked as a "problematic edge", and a trigger to activate the measure is generated that will later on be passed on to the simulator.

Then the real negotiation process starts. It consists of the following consecutive steps:

- For each problematic edge a request for help is issues which formulates the service it would like to get from adjacent edges. This service can be either "decrease output" (from an edge in another TMS that is an input edge to the problematic edge), or "increase input" (from an edge in another TMS that is an input edge to the problematic edge.
- Then each TMS checks if it can fulfil the request. It can do this if it has a traffic control measure available that implements the required service (e.g. a ramp metering system is an example of a traffic control measure that implements the service "decrease output"). If it has, it calculates an associated cost which is determined from the severity of the request and a utility function. The utility function values the situation of the edge providing the service, in combination with the priority associated with it: a higher priority increases the costs associated with fulfilling a request. All this information is combined in an "offer" which is sent to the requesting edge.
- The requesting (problematic) edge then determines whether the costs are acceptable, by comparing it to a predefined value. These predefined values are currently arbitrary and static. We regard costs as an abstract notion which in reality could be financial or some other valuable asset (e.g. a guarantee from one TMS to another to "return the favour" at some point). Making predefined cost values dynamic would not be hard to model and could be interesting if, for example, a TMS is willing to pay more during rush hours.
- If the offer is accepted, then it is finalised and turned into a trigger for the traffic simulator. If there is "no deal" then obviously it is not.

At the end of this process all triggers are communicated to the simulator, which takes the effect of each traffic control measure/trigger into account before processing a new simulation step.

5 The VDM Model

In order to explore the different conceptual possibilities in connection with collaboration between different TMSs a VDM++ model[13] has been produced. In general the model contains a number of invariants and other constructs that can only be found in VDM++ and similar modelling notations, but some of these aspects are not included in the paper due to the lack of space. The purpose of this model is to illustrate the potential

[13] The model is available in the standard Overture library with VDM++ examples.

collaboration possibilities between TMSs, so focus has been on these aspects and ways in which it is possible to visualise collaboration for non-technical stakeholders. A class diagram of the most important classes can be found in Figure 2.

The VDM++ model follows the general design principles used for the development of distributed real time systems [7].

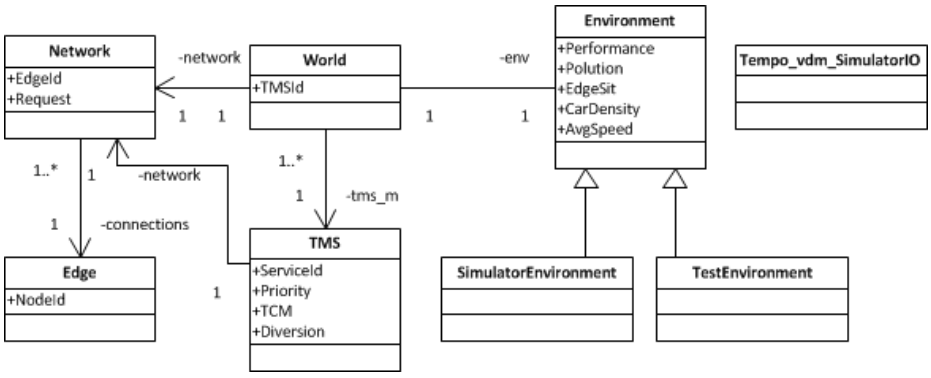


Fig. 2: Class Diagram with the most important TEMPO classes

These classes have the following function in the model:

World: The `World` class is the entry point for the VDM model. It has connections to the `Environment`, to the `Network` and to all the different `TMS` instances (organised in a mapping structure). This class collects the detailed information about the road network and the different TMSs controlling the different parts of the network. At the top-level two different `Run` operations are included in order to enable simulation of the same scenario either with or without collaboration, so that the difference between these two scenarios can be visualised.

Environment: The `Environment` class is a superclass for alternative realisations of the environment to the collection of TMSs that need to collaborate. The role of this class is to act as an interface to the main system, keeping track of the history of the traffic situation measured during a simulation.

SimulatorEnvironment: The `SimulationEnvironment` class is a subset of the `Environment` class and its role is to hook up to the low-level simulation environment realised via the `tempo_vdm_Simulation_IO` class. The latter communicates with the traffic simulator. The simulator visualises the progress of traffic flows and the traffic measures that are deployed.

TestEnvironment: The `TestEnvironment` class is a subset of the `Environment` class and its role is to provide a simpler traffic simulator without the user interface provided by the traffic simulator.

TMS: The `TMS` class represents a single Traffic Management System such that a combination of multiple TMSs can be considered.

Network: The `Network` class provides functionality for the road connections between different points in a graph-like fashion.

Edge: The `Edge` class provides functionality for edges in the graph representing the roads in the network.

tempo_vdm_Simulation_IO: The `tempo_vdm_Simulation_IO` class contains operations that interface to a small application that acts as a front-end to the traffic simulator. The output of this is easy to use for communication by domain experts and it can be seen in Figure 3.

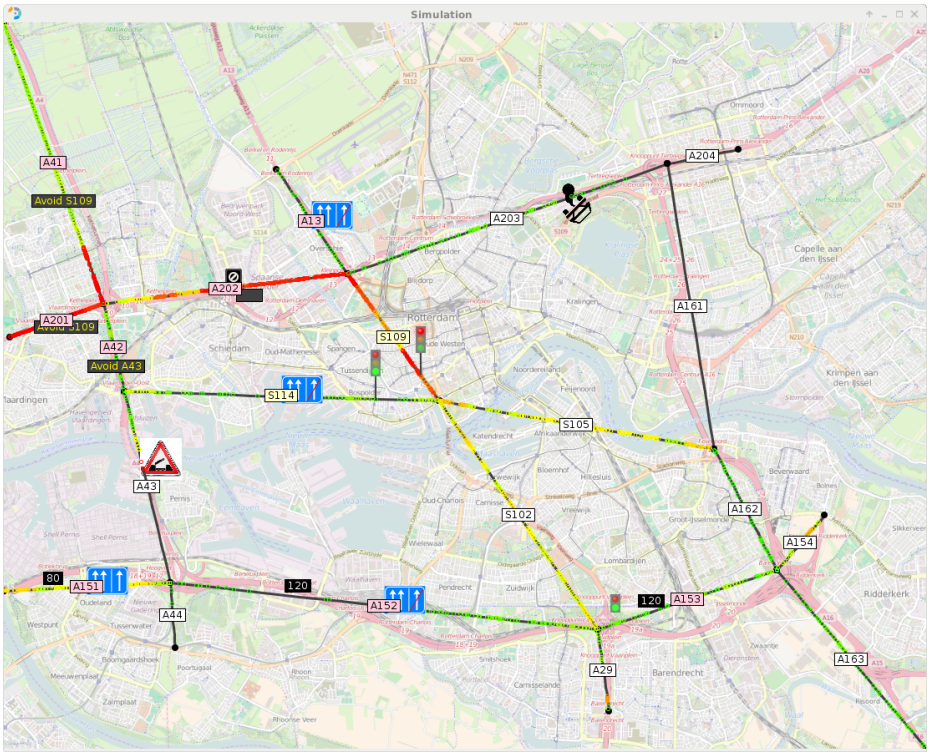


Fig. 3: Screenshot of the TEMPO simulation.

The configuration shown shows the network in and around the city of Rotterdam (i.e. a simplification if it) with two TMSs. The roads marked with an "A" in the figure are controlled by the TMS of the national road authority Rijkswaterstaat. Those marked with an "S" are controlled by the TMS of the municipality of Rotterdam.

In order to investigate the effect of collaboration between the different TMSs both possibilities are made available in the `World` class by having a simple instance variable controlling this:

```
1 collaboration : bool := true;
```

Inputs also are needed for the configuration of TMSs, for the road network, and for the events that can happen at different points in time during a simulation. All this information is stored in Excel files and read into the VDM model using the `CSV` standard library.

In each `Step` every TMS considers the traffic situation, and based on that generates triggers internally in the TMS and subsequently generates requests for help from adjacent TMSs. The requests are then processed by the overall network that has information about all TMSs that have been configured.

```
1  public Step: Environment`TrafficSituation ==> ()
2  Step(ts) ==
3    (trigger:= {|->};
4     trafsit := ts;
5     GenerateMyOwnTriggers(ts);
6     let requests = GenerateRequestsForHelp()
7     in
8       network.AddRequests (requests));
```

Inside the `SimulatorEnvironment` class, if collaboration is enabled, the top-level `Run` takes such service requests into account and makes offers that are subsequently evaluated before they are finalised, which means that the offers are accepted unless the `TMS` itself has deeper reasons for helping itself (instead of helping its neighbours):

```
1  public Run: bool ==> ()
2  Run(colab) == (
3    while not isFinished() do (
4      dcl trafsit: TrafficSituation;
5      dcl control: TMS`Control := {|->};
6      tempo_vdm_SimulatorIO`runSimulator(10);
7      trafsit := UpdateSit();
8      for all id in set dom tms_m
9      do tms_m(id).Step(trafsit);
10     if colab
11     then for all id in set dom tms_m
12           do tms_m(id).MakeOffers();
13     for all id in set dom tms_m
14     do tms_m(id).EvaluateOffers();
15     for all id in set dom tms_m do
16       let c = tms_m(id).FinaliseOffers()
17       in
18         control := control ++ c;
```

The argument passed to `runSimulator` (10) is the number of seconds of simulation time, so each step in the simulation is 10 seconds.

In order to control the traffic there are different Traffic Control Measures (TCMs) that can be applied:

```
TCM = HardShoulder | MaxSpeed | TrafficLight | RampMeter |
      Diversion | LaneClosure;
```

The type definitions for the various TCMs have parameters that determine specific settings for them. For example, a `TrafficLight` has an associated "green time" (duration of the time that traffic in the given direction can cross the intersection), whereas a `HardShoulder` just has a boolean value (open or closed).

Different thresholds are used to determine if it makes sense to activate some of these TCMs at specific edges at the road network. Here the active controls are gathered in an instance variable called `actctrl` that needs to be updated depending upon whether the given edge is problematic. This can for example be seen in the `SetHardShoulder` operation:

```
SetHardShoulder: Network`EdgeId * Environment`EdgeSit ==>
                 set of TCM
SetHardShoulder(eid, mk_(d,v,-,-,-)) ==
  if v < LOWSPEEDTHRESHOLD or d > HIGHDENSITYTHRESHOLD
  then (problematicedges := problematicedges union {eid};
        if eid not in set dom actctrl
        then return {mk_HardShoulder(true)}
        else return {})
  elseif v > HIGHSPEEDTHRESHOLD and d < LOWDENSITYTHRESHOLD
         and eid in set dom actctrl
  then return {mk_HardShoulder(false)}
  else return {}
```

The first part describes under which conditions a hard shoulder is opened: thresholds for either density or velocity at a given edge are crossed. The edge is tagged as *problematic* if this happens. Thresholds are used here to prevent toggling behaviour of the measures involved. The latter part of this operation deals with closing a hard shoulder when the traffic situation improves.

When an edge is "problematic" it sends out "requests for help" to other TMSs that are known to be able to influence that edge (typically input and output edges). For this an operation `GenerateRequestsForHelp` is used, that broadcasts the request for help to those TMSs that can potentially influence the problematic edge.

The request for help is expressed in terms of a *service* required. The service can either be `<IncreaseInput>` or `<DecreaseOutput>` at the moment[14]. The service called `<IncreaseInput>` is requested from downward edges to "absorb" more traffic so that tension on the problematic edge is relieved, and vice versa for `<Decrease-`

[14] Another future service might be `<IncreaseThroughput>`, a service e.g. provided by a hard shoulder opening.

Output>. A TMS asked for help can then see if it has any traffic control measures available that provide that service, and typically there exists a *1:n* relationship between a service and a TCM.

The requests for help are furthermore augmented with a severity indication, calculated by using a utility function:

```
Utility(network.GetMaxSpeed(eid),trafsit(eid), prios(eid))
```

The utility function here uses the maximum speed at an edge, the traffic situation at that edge (to compare with the maximum speed) and the priority of the road (roads with a higher priority get precedence and should be "helped" more quickly). The utility function is a function that in real life situations would be specified by traffic domain experts. The severity is used by the receiver of the request to compare to its own situation: it either determines whether or not help is granted or it influences the "price" that has to be paid for getting help. This is done in the operation MakeOffers, where each TMS iterates over the edges it controls to see if it has any traffic control measures available that provide the service requested. The operation also calculates the cost that comes with providing help:

```
let s = {e.service | e in set requestedservice},
    tcm = RequestedServices2PossibleTCMs(requestor, eid, s)
in
  let cost = Utility(network.GetMaxSpeed(eid),
                     trafsit(eid), prios(eid))-severity
  in
    if tcm <> {}
    then network.MakeOffer (eid, cost);
    ...
```

All requestors then evaluate the offers made to them with the operation EvaluateOffers:

```
public EvaluateOffers: () ==> ()
EvaluateOffers () ==
(for all eid in set myedges do
  let offers = network.GetOffers(eid)
  in
    for all provider in set dom offers do
      let reqs = offers(provider)
      in
        network.AcceptOffers(provider,
                        {if request.cost < ACCEPTABLECOSTS
                         then mu(request,accepted |-> true)
                         else request
                        | request in set reqs})
```

```
14 );
```

So, as seen here, an offer is simply accepted when its associated costs are not above a given threshold (`ACCEPTABLECOSTS`). More sophisticated negotiation models would first proceed by making counter offers, and so on.

When an offer is accepted it is then finalised by transforming it into a trigger to set the traffic control measure that has been offered. The final set of triggers consists of:

- Triggers that are the result of each TMS seeing what it can do itself, i.e. apply the traffic control measures that are under its own control.
- Triggers that result from the negotiation process, i.e. traffic control measures that are part of accepted offers.

In real life, triggers would be passed on to the various local systems that implement the traffic control measures. In the simulator environment they are passed on to the traffic simulator where the effect that they have on the current traffic situation is calculated.

6 Results

The most significant result of the work presented in this paper is that we can demonstrate the working of traffic management with and without negotiation between the TMS involved. The initial results of executing the TEMPO VDM model are reflected in Figure 4, showing a plot monitoring the run-time average speed and density of the two different TMSs using the new Overture extension presented in [2]. We believe that the combination of the live plotting and the GUI from the traffic simulator can be useful in showing traffic management stakeholders that collaborative TMSs have added value.

In the figure the average speed and the average density in the two TMSs are plotted over time. Naturally this is heavily affected by the amount of traffic and the events incorporated in a simulation, so one needs to compare a large collection of cases in order to provide solid proof that negotiations between TMSs has a positive effect on traffic flow.

We also made first steps in comparing the performance of a network with and without collaboration in a more quantitative way by implementing some performance indicators, such as the number of vehicle kilometers produced by the network and the number of congestion kilometers that occur over the entire simulation run. These indicators, however, turn out to be quite sensitive to the specific topology and configuration (which traffic control measures are present and where are they located). For example, if the negotiation results in a better situation on a part of the network where the maximum speed is relatively low, then this will result in a lower number of vehicle kilometers produced, while intuitively the network has performed better. Therefore, it is questionable whether or not the indicators we choose are effective and accepted indicators by experts in the field of traffic modelling.

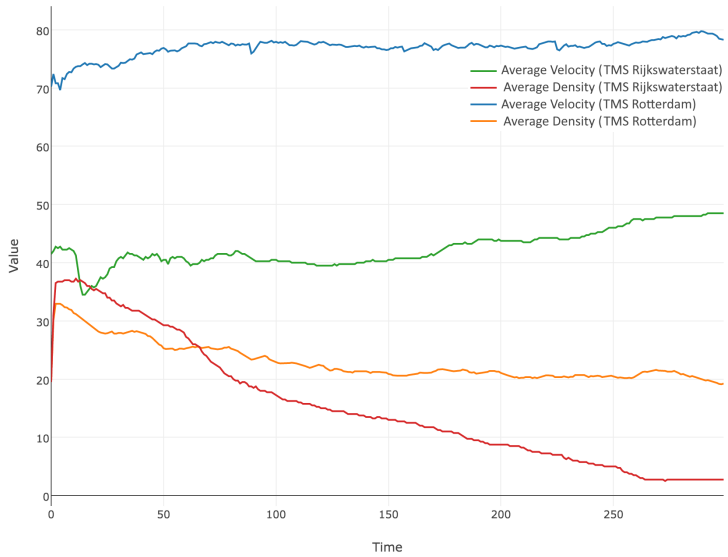


Fig. 4: Plotting speed and density with collaboration.

7 Concluding Remarks and Future Work

In this paper we have presented a novel way of dealing with conflicting interests between various TMSs, each responsible for guiding and steering traffic over (part of) a road network. We did this by specifying a VDM model that facilitates TMSs to negotiate about getting help from each other by applying traffic control measures that will improve overall performance, despite the fact that the measure may not have a positive effect for the TMS providing help itself. The hypothesis is that this results in an improvement of the overall performance of the network.

We built a traffic simulator that feeds the model with the (calculated) traffic situation on the network, and takes steering information from the model into account in each simulation step. The network, distribution of vehicle density and speed, and the status of traffic control measures, are visualised through a graphical user interface. This enables a nice demonstration of the working of the model, including the negotiation principles that have been included. Changes in density and speed for each TMS over time for the entire simulation run are visualised using new extensions to the Overture toolset.

A quantitative comparison between performance of a network with and without collaboration (based on negotiation) is the obvious first step for future work. Although we did build in some performance indicators, such as the number of vehicle kilometers produced by the network and the number of congestion kilometers that occur over the entire simulation run, these all turn out to be quite sensitive to the specific topology and configuration. This will probably require the involvement of experts in traffic modelling.

Ideas for future work include:

- Enhancement of the negotiation system. The negotiation strategy as described in this paper is relatively simple: the costs of offers made are compared to standard, predefined values. The offer is accepted if its costs are below that predefined value, and otherwise they are not. However, one can imagine a more sophisticated system where counter offers are made based on the initial offers that are being provided, possibly iterated over a number of steps. This could lead to more realistic values of what the offers are really worth. It would also be nice to be able to prepare the model for new types of negotiations needed in the future.
- Modelling of different types of TMSs. The two TMSs that are currently configured in the demonstrator (national road authority Rijkswaterstaat and municipality of Rotterdam) are "traditional" executing organisations of traffic management. It would be interesting to include new parties, such as providers of navigation equipment (introducing mobile sensors, and mobile control, in this case in-car dynamic trip planning/traffic information), cooperative systems, TMSs involved with other modalities (public transport, vessels, etc.) or even TMSs that do not have any control measures themselves (such as event organisers), but who can participate in the negotiation process.
- Exploring the possibility to use existing, more powerful traffic simulators (e.g. an open source simulator like SUMO[15]) instead of the one we built ourselves. Our main motivation for doing it this was that the budget for the project was limited, and we needed a simulator that could easily be adapted to the specific interface we are using for communication with the VDM model.
- Exploring the working of the model in a more physical setting, eg. using real traffic and using real traffic control measures. Worldwide there are several facilities where this can be done, varying of course in the level of sophistication they can offer to different experimenters.

Acknowledgments Our work was partially supported by the European Commission as a small experiment chosen for funding by the CPSE Labs innovation action (grant number 644400). Thanks also go to the anonymous reviewers of this paper and to Skip Balk, Pieter Buzing, Luis Couto, Kenneth Lausdahl, John Skovsgaard and Henrik Nymann Jensen for the various contributions they made to the TEMPO project.

References

1. Beam, C., Segev, A.: Automated Negotiations: A Survey of the State of the Art. Wirtschaftsinformatik 39(3), 263–268 (1997)
2. Couto, L.D., Lausdahl, K., Plat, N., Larsen, P.G., Pierce, K.: Extending Overture with Implicit Monitoring of Instance Variables. In: Larsen, P.G., Plat, N. (eds.) The 14th Overture Workshop: Analytical Tool Chains. Cyprus, Greece (November 2016)
3. Fitzgerald, J., Gamble, C., Larsen, P.G., Pierce, K., Woodcock, J.: Cyber-Physical Systems design: Formal Foundations, Methods and Integrated Tool Chains. In: FormaliSE: FME Workshop on Formal Methods in Software Engineering. ICSE 2015, Florence, Italy (May 2015)

[15] SUMO is an acronym for "Simulation of Urban MObility", see `http://sumo.dlr.de/wiki/Main_Page`.

4. Fitzgerald, J., Larsen, P.G., Verhoef, M. (eds.): Collaborative Design for Embedded Systems – Co-modelling and Co-simulation. Springer (2013)
5. Jennings, N.R., Faratin, P., Lomuscio, A.R., Parsons, S., Sierra, C., Wooldridge, M.: Automated Negotiation: Prospects, Methods and Challenges. International Journal of Group Decision and Negotiation 10(2), 199–215 (2001), http://eprints.soton.ac.uk/254231/
6. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The Overture Initiative – Integrating Tools for VDM. SIGSOFT Softw. Eng. Notes 35(1), 1–6 (January 2010), http://doi.acm.org/10.1145/1668862.1668864
7. Larsen, P.G., Fitzgerald, J., Wolff, S.: Methods for the Development of Distributed Real-Time Embedded Systems using VDM. Intl. Journal of Software and Informatics 3(2-3) (October 2009)
8. Larsen, P.G., Lausdahl, K., Tran-Jørgensen, P.W.V., Ribeiro, A., Wolff, S., Battle, N.: Overture VDM-10 Tool Support: User Guide. Tech. Rep. TR-2010-02, The Overture Initiative, www.overturetool.org (May 2010)
9. Lee, E.A.: Cyber Physical Systems: Design Challenges. Tech. Rep. UCB/EECS-2008-8, EECS Department, University of California, Berkeley (Jan 2008), http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-8.html
10. Maier, M.W.: Architecting principles for systems-of-systems. Systems Engineering 1(4), 267–284 (1998), http://dx.doi.org/10.1002/(SICI)1520-6858(1998)1:4<267::AID-SYS3>3.0.CO;2-D
11. Nielsen, C.B., Larsen, P.G., Fitzgerald, J., Woodcock, J., Peleska, J.: Model-based engineering of systems of systems. ACM Computing Surveys 48(2) (September 2015), http://dl.acm.org/citation.cfm?id=2794381
12. Nielsen, C.B., Lausdahl, K., Larsen, P.G.: Combining VDM with Executable Code. In: Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) Abstract State Machines, Alloy, B, VDM, and Z. Lecture Notes in Computer Science, vol. 7316, pp. 266–279. Springer-Verlag, Berlin, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-30885-7_19, ISBN 978-3-642-30884-0

Decoupling validation UIs using Publish-Subscribe binding of instance variables in Overture

Luis Diogo Couto[1], Kenneth Lausdahl[2], Nico Plat[3], Peter Gorm Larsen[2], and Ken Pierce[4]

[1] United Technologies Research Center, Ireland
[2] Aarhus University, Department of Engineering, Denmark
[3] West IT Solutions, The Netherlands
[4] Newcastle University, School of Computing Science, UK

Abstract. Being able to efficiently communicate the way formal models behave to stakeholders who are not experts in formal models is important for the potential commercial exploitation of such technologies. For the Overture/VDM open source initiative, this has so far been accomplished by interpreting executable models in combination with executable Java applications, typically demonstrating a conceptual Graphical User Interface (GUI). However, since that approach requires additional explicit calls to GUI functionality, this paper demonstrates a new publish/subscribe based approach enabling implicit monitoring of changes to variables during an interpretation of a formal VDM model that does not "pollute" the VDM model in the same way as existing approaches.

Keywords: Overture, VDM, user interface, Electron

1 Introduction

The Overture/VDM tool [8] supports three different dialects of VDM [4]. Overture is already capable of interpreting executable subsets of VDM [10] and it is also possible to combine such interpretation with legacy Java code [11]. However, the latter solution can be a little clumsy and "pollute" the VDM models by requiring explicit calls to operations that update, for example, a graphical user interface. The contribution of this paper is to enable an easy way to monitor instance variables as they are changed during an interpretation of a VDM model in an implicit fashion. The primary focus of our paper is on monitoring variables and displaying model information in a user interface. However, our work also supports calls from the User Interface (UI) to the model. This contribution is based on top of the web-based Electron [2] platform and it is connected to the Overture tool as a plug-in extension. Electron is a modern and popular framework for desktop application development. It was chosen because it enables use of web technologies, has a wide amount of existing libraries and modules, and has an active community supporting it.

The TEMPO[5] project investigated collaboration between different Traffic Management Systems (TMSs) by providing them with collaborative control architectures that

[5] See `http://tempoproject.eu/`.

engage with each other in automated negotiation processes [14]. TEMPO used the Overture tool to analyse both existing traffic management networks and potential collaborative designs, demonstrating the benefits of smart traffic systems. Traffic simulations produce a large amount of numerical data; it is imperative to present this in an understandable way to non-experts to make computer simulations useful. Overture is being extended with a 2D/3D visualisation library to show the negotiations between TMSs and the effect on traffic flow.

This paper focusses on extensions to the Overture tool that enable a fast, lightweight way to visualise how variables in a model change during interpretation. In the remainder of this paper, the extensions are explained in Section 2. This is followed by a short explanation of the practical application used for viewing the changes to the instance variables in Section 3. Afterwards Section 4 presents the case study in which this new technology has been used. Then Section 5 presents related work. Finally, we conclude the paper and suggest future work possibilities in Section 6.

2 The Overture Extension

2.1 Design Principles

The Overture interpreter already has features that enable it to interact with external components implemented in Java [11]. These features include the Java bridge which enables VDM models to invoke functionality implemented in Java and the `RemoteControl` interface which enables Java programs to directly control the Overture interpreter. These features have frequently been used to implement prototype user interfaces.

The extension we have developed takes into account the existing work connecting Overture to Java and seeks to address certain issues with it. As such, the extension was designed according to two key principles:

1. The extension must enable the use of modern and fast UI technologies; and
2. the UI code must not pollute the VDM model.

Principle 1 attempted to address an ongoing issue with slow UI prototyping. Pure Java interfaces are typically developed with the Swing toolkit, which is rather slow and laborious. When developing UI prototypes, it is important to be able to construct a working UI or we end up spending too much time on the UI. Swing works against this, particularly for any UI that is non-trivial. Furthermore, Swing interfaces look extremely dated and unappealing. While the aesthetics are not the primary concern of a UI prototype, it should still look presentable and adjust to the modern expectations of stakeholders, or we risk that they focus too much on the old look of the UI and thus cannot properly assess the model.

Principle 2 simply aims to improve on the existing status of the Java bridge. While there was little we could do to improve the status of Swing UIs, we have total control of the Java bridge. In the current implementation, in order to display any information in the UI, the model must explicitly call the UI to set the values. While there are alternatives, this is typically done because it is the most expedient way of displaying new data as

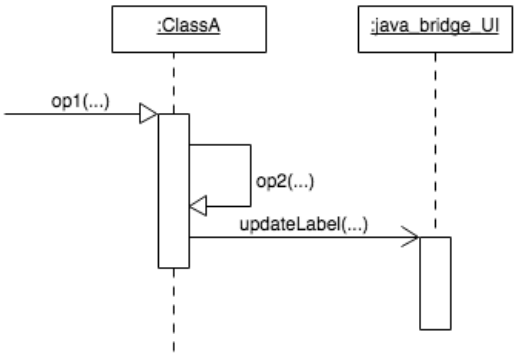


Fig. 1. Explicit call from model to UI.

soon as it is available in the model. This behaviour is shown in a sequence diagram in Figure 1.

While the explicit call approach is efficient and pragmatic, it does lead to models that are polluted with UI calls. These calls may affect the tractability of the model as well as its readability from a purely formal point of view. In addition, these explicit calls might make it impossible to execute the model without the UI present as the UI calls would fail – while this can be worked around using guards, that means polluting the model even further due to the UI.

Finally, it is worth mentioning that the existing approach typically leads to fuzzy divisions between UI, data and logic in the solution. Some UI code is in the model and the logic is often split between model and UI. Modern software development practices typically advocate for a clean separation between UI and logic, using patterns such as model-view-controller [7]. By adhering to the aforementioned design principles, the new extension will enable users to more cleanly separate their UI from their application logic.

2.2 Implementation

The new extension is based on the publish/subscribe design pattern [3]. The extension enables the UI to subscribe to statically known model variables. Whenever the value of a subscribed variable is changed, an event is broadcast with the new value. The UI can listen for this event and then react accordingly. Thus, all UI interaction is hidden from the modeller – leading to the behaviour shown in the in Figure 2, where the UI is notified of updates values in an asynchronous way without the need for explicit calls from the model.

The core of the extension is a new remote control class for the Overture Interpreter [10]. Remote control is a feature of the existing Overture interpreter Java bridge that enables a VDM model to be controlled remotely by a Java program that implements the `RemoteControl` interface [9, chapter 7]. Remote control classes are typically

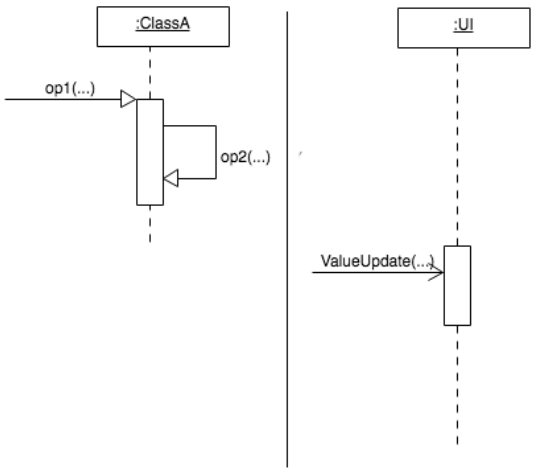


Fig. 2. No calls from model to UI.

used as part of the implementation of Swing UIs in order to allow the UI to control the model. The remote control class can evaluate expressions in the context of the VDM model and invoke model operations in order to modify the model's internal state.

The remote controller of our extension also controls the interpreter in this manner but rather than directly controlling the interpreter according to a given logic, the remote control class of our extension re-exposes the control of the interpreter via a JSON based protocol over web-sockets [1]. In this way, any UI can be connected to any Overture model in a completely generic way, based on REST approaches. Most notably, this extension is backwards-compatible. Any existing VDM model can be controlled with it without a single modification being made to the model.

In addition to control over the interpreter, our protocol also defines the means by which a UI can subscribe to variables, and how the remote controller broadcasts updates to subscribed clients. The remote controller is provided as a single jar that contains everything necessary to execute it via the Overture tool, as shown in Figure 3. Because it is launched from within Overture, all the IDE features such as breakpoints and model coverage are available for models executed with a UI.

The remote control is relatively simple. It contains a Jetty-based web server [6] to expose the protocol and uses the Overture Java bridge's `RemoteControl` Java interface to control the interpreter. The main components are summarised in the class diagram shown in Figure 4.

The most notable class besides `TempoRemoteControl` is `VarListener`, that is responsible for monitoring variables in the VDM model and broadcasting messages with the updated values. This class uses the same mechanism and infrastructure already present in the interpreter to monitor invariant violations during model executions since this mechanism already hooks into every point in a model execution where a variable

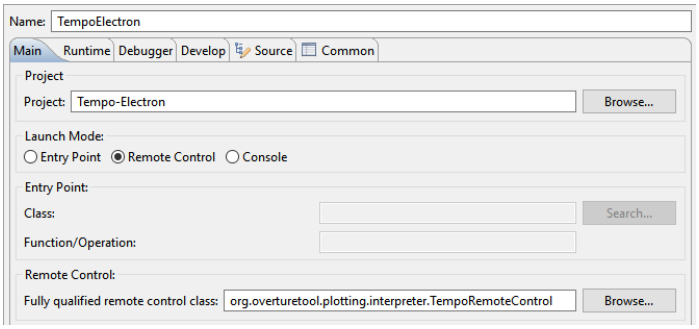


Fig. 3. Overture launch configuration for remote control.

can be updated. A separate `VarListener` is attached to the `Value` objects for each variable that is to be monitored.

2.3 Control and Subscription Protocol

The protocol offered by the extension enables top level control of the Overture VDM interpreter and also defines messages to control various aspects of model execution and monitoring including: definition of model entry point, query model structure information, subscription to value changes in variables and execution of operations.

The most important messages in the protocol are illustrated in Listing 1.1 and Listing 1.2 using the VDM model shown in Listing 1.3:

RunModel tells the interpreter to start simulating the model using a previously configured entry point class and an operation provided in the message. Listing 1.1 line 2-3.

SetRootClass tells the interpreter to set the entry point to the class passed in the message. Listing 1.1 line 6-9.

GetFunctionInfo requests from the interpreter a list of all visible functions and operations that can be used as entry point from the currently configured entry point class. A response is sent with a list of available operation names. Listing 1.1 line 12-13.

GetModelInfo requests from the interpreter a list of all observable variables in a model, accessible from the defined entry point. A response is sent that recursively lists the names of all observable variables. Listing 1.1 line 16-22.

Subscribe subscribes to updates sent out when the variable passed in the message is update. Currently, only instance variables present in the class definition of the root class (both static and non-static) are supported. Listing 1.1 line 25-26.

Execute tells the interpreter to execute the model operation passed in the message. This message can only be sent when the interpreter has an active simulation and the requested operation must be visible. These messages will allow the UI to pass data into the model and control its execution. Listing 1.2 line 2-4.

ValueUpdate a message sent containing the name and updated value of a subscribed variable. This message is broadcast by the server whenever the value of a subscribed variable changes. Listing 1.2 line 7-13.

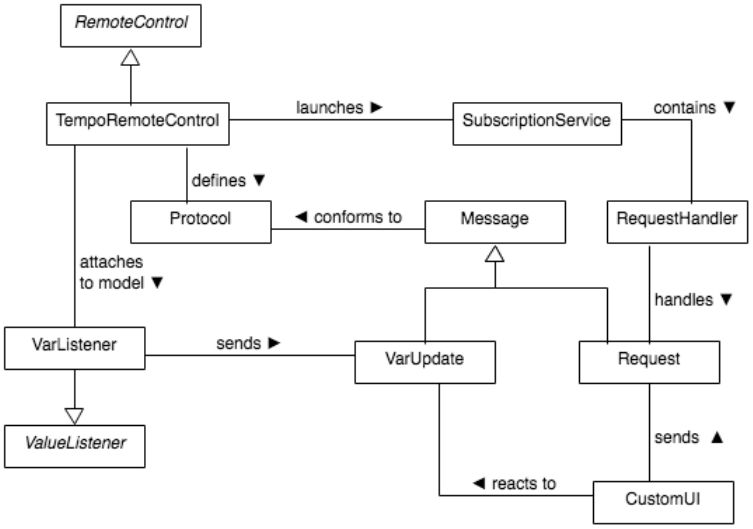


Fig. 4. The main elements of the extension.

StopServer tells the interpreter to stop the simulation. Listing 1.2 line 16.

All messages in the protocol are specified using JSON. The protocol uses a request response pattern where all messages except value change notification has a predefined request and response type. Listing 1.1 in combination with Listing 1.2 shows how the messages looks when the VDM model in Listing 1.3 is simulated by selecting the `A` class as root and observing `x` while running `op()`. The visual 2D output graph of this is shown in Figure 5.

Listing 1.1. Request and response sample messages..

```
1  // Obtain model classes
2  {"type":"REQUEST","data":{"request":"GetClassinfo"}}
3  {"type":"CLASSINFO","data": ["A","B"]}
4
5  // Set current root class
6  {"type":"REQUEST","data":
7    {"request":"SetRootClass","parameter":"B"}
8  }
9  {"type":"RESPONSE","data":"OK"}
10
11 // Get available functions or operations
12 {"type":"REQUEST","data":{"request":"Getfunctioninfo"}}
13 {"type":"FUNCTIONINFO","data": ["op"]}
14
15 // Obtain state info of root class
```

```
{"type":"REQUEST","data":{"request":"GetModelinfo"}}
{"type": "MODEL", "data":
  {" rootClass": "mm", "name":"", "type":"",
    "children": [
      {"name": "x", "type":" int", "children": []}
    ]}
}

// Subscribe to a variable change from the root class
{"type":"SUBSCRIBE","data":{"variableName":"x"}}
{"type":"RESPONSE","data":"OK"}
```

In Listing 1.2 only value change messages are shown for the first two values.

Listing 1.2. Request and response sample messages..

```
// Start simulation
{"type":"REQUEST","data":
  {"request":"RunModel","parameter":"op"}
}

// Receive value updates
{"type":"VALUE","data":
  {"variableName":"x","type":"int","value":"1"}
}

{"type":"VALUE","data":
  {"variableName":"x","type":"int","value":"0"}
}

// Stop server
{"type":"REQUEST","data":{"request":"StopServer"}}
```

The VDM model in Listing 1.3 is a two class model `A` and `B` where the latter class contains a state `x` which can be observed, it alters between `1` and `0` through the execution of `op()`'s execution.

```
class A
end A

class B

instance variables
  x : int := 0;

operations

public op : () ==> ()
op() ==
```

```
for all i in set {1,...,10}
do x := x + if (i mod 2) > 0
            then 1
            else -1;

end B
```

Listing 1.3. Oscillating between +-1

The Electron application will show the following 2D graph shown in Figure 5 when the model in Listing 1.3 is executed.

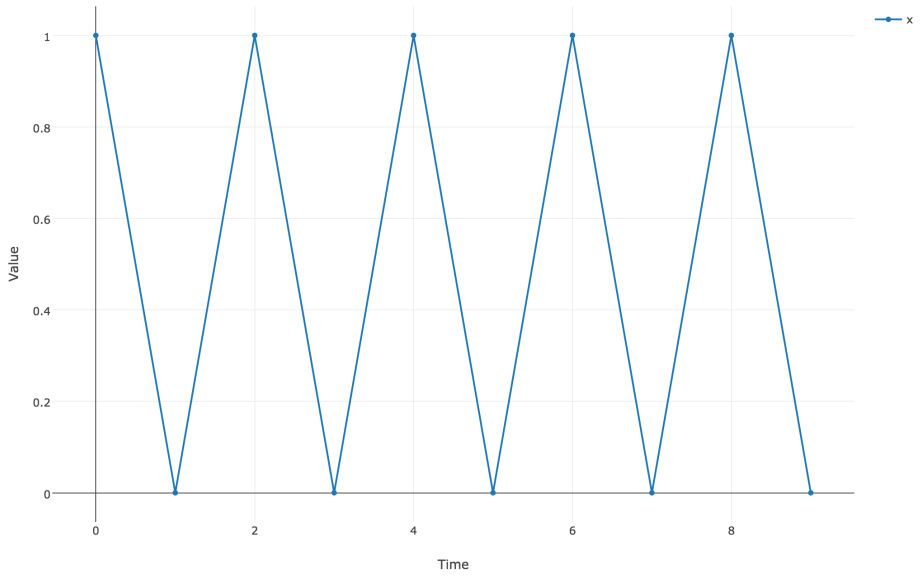


Fig. 5. 2D graph of the oscillating VDM model.

3 The Electron Application

As an example of the kinds of UIs that can be developed with our extension, we present a visualisation application based on variable plotting. The visualisation application is developed using the Electron framework that enables native applications to be developed using web technologies like JavaScript, TypeScript, HTML and CSS. Electron is compatible with Mac, Windows and Linux and therefore easily deployable to the same platforms as Java.

The plotting capability is created as an Electron application that uses web-sockets and JSON to communicate with the Overture Extension from Section 2. It enables the

user to select the entry point of the model in the form of a class and operation/function. Both of these are currently limited to be used without any arguments. The user can then create plots based on variables accessible from the entry point. The variables that are offered as possibilities are all either numeric or lists (of fixed length) of numbers. Once such a configuration is completed the application can request the Overture interpreter to start simulating. The application will then plot all variable changes live during the simulation on the respective graphs. In Figure 6 an example of a 2D plot is shown.

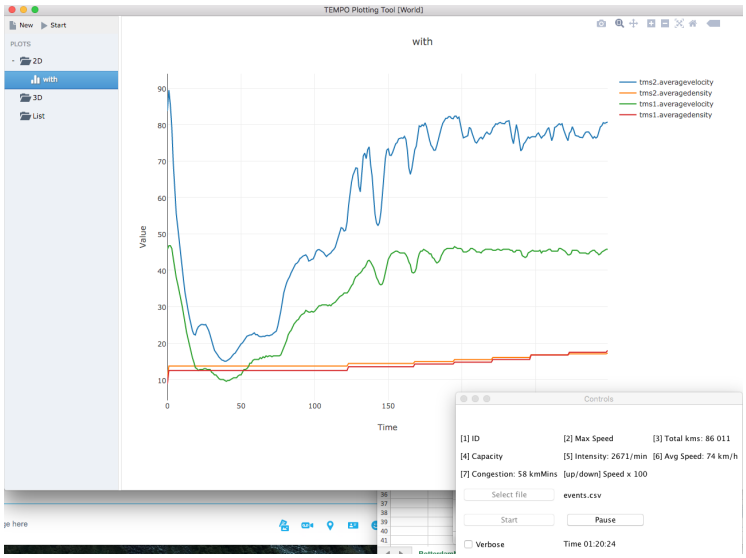


Fig. 6. 2D graph visualisation in the Electron application.

The plot configurations can also be stored and reloaded to make simulations of VDM models with more extensive plot configurations without having to redo this every time.

In addition to the conventional 2D plots it is also possible to plot using 3D as well as plot elements of list of numeric varies with fixed lengths. An example of a 3D plot can be seen in Figure 7. Note that it is possible to turn the different axes around when one is using a 3D plot. However, for an application like the one used in TEMPO the 3D effect does not give so much value so we have mostly used the 2D plots.

4 The TEMPO Case study

The TEMPO[6] case study is the first example in which the plotting extensions to Overture are used to communicate relevant parameters of a model execution to domain ex-

[6] TEMPO is an acronym for "TMS Experiment with Mobility in the Physical world using Overture". See `http://tempoproject.eu/` for more information.

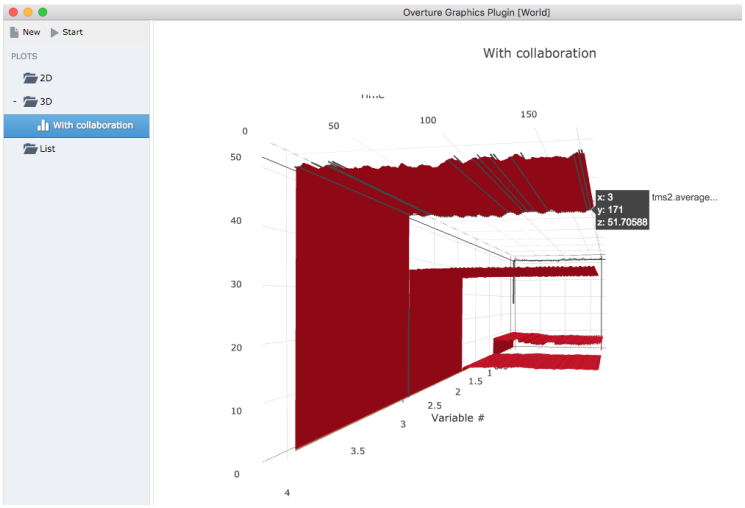


Fig. 7. 3D graph visualisation in the Electron application.

perts. It is a case study in the field of (road) traffic management, and it is based on the following idea.

Many European countries have dense road networks and significant traffic problems. The flow of traffic on Europe's roads is managed by a series of TMSs that are owned and controlled by various local and national authorities. A TMS consists of a collection of distributed systems and devices, usually installed along the roadside. These can be sensors that collect traffic data, such as cameras, radar detection systems, induction loops, or actuators that give instructions to road users via signs and signals. Current TMS architectures are usually run centrally from regional control centres. A low degree of collaboration hinders efficient management of traffic problems that straddle boundaries between authorities, because TMSs cannot communicate between regions and may have competing goals for traffic flow. While cooperation between various road authorities at a governance level has improved recently, technical barriers for collaborative TMSs are still to be removed.

The TEMPO project addresses the problem of disconnected TMSs by providing them with collaborative control architectures that engage with one another in automated negotiation processes. Negotiations are based on policies, which help reach agreement on which control measures are beneficial for the system as whole, improving the overall network performance.

Within TEMPO two VDM models have been made. One specifies a system of systems in which multiple TMSs co-exist that behave in an egocentric way: they do not communicate with one another and therefore they do no collaborate, they just apply the control measures that they have to only help themselves. The other model incorporates communication, negotiation and cooperation between various TMSs. They inform each other of their needs, and see if they can either provide help or get help to or from each other. An example of such help is setting up a diversion route. If one TMS has serious

congestion problems on a part of its network then it can ask the other TMSs to provide a service "decrease output" to its network. Another TMS can then see if it has any measures that can provide this service, e.g. by setting up a diversion route that encourages travellers to avoid the congested part of the other TMS. Of course, setting up such a diversion route may cause problems for the TMS that provides the service itself, and therefore a cost (price) is associated with it. This is negotiated between the TMSs, ending up in a situation where the control measure is actually applied or not, depending on the costs.

In TEMPO, the execution of the two models (non-collaborative/egocentric versus collaborative) are compared to see if the collaborative version really provides better results. In order to do this, a simple traffic simulator has been developed which communicates a traffic situation to the model. The VDM model then executes the calculations (with or without negotiations) ending up with a series of traffic control measures to be applied. These are then communicated to the traffic simulator, which takes the measures into account, performs another round of traffic processing, provides a new traffic situation to the model, and so forth.

To determine how each model "performs" is not easy. One can think of performance parameters calculated over the execution of an entire simulation, e.g. the number of vehicle kilometres "produced" by the network, or the accumulation of congestion (product of length and duration), but these remain debatable and give little (visual) insight over what is happening during the course of the simulation itself. However, the plotting extensions are useful here. Typical indicators that a traffic engineer would look at in these situations are the average density of traffic in the network, and the average velocity (speed) of vehicles. In the model these parameters are modelled as instance variables. In the plots shown in Figure 8, generated with the extensions discussed we see density and speed evolve over time. The first plot shows the situation for a non-collaborative model and the second one for the collaborative one. The network used is a simplification of the road network in and around Rotterdam in The Netherlands. Part of this network is controlled by the municipality of Rotterdam (TMS1) and part by the national road authority Rijkswaterstaat (TMS2). It is easy to see how the plot visualisation gives a good impression of the performance of the two models.

5 Related Work

Tools for visualising VDM models have been developed in the past as well. The first ability to combine the interpretation of a VDM model with executable code was made on the VDMTools platform [5]. Here special dynamic link libraries were introduced. This means that the "pollution" of the VDM model was limited to a link to a `.dll` file and the explicit calls in the VDM model. Compared to the work presented here this is a much more cumbersome approach that requires a lot more knowledge to low-level implementation details.

For more recent related work it is in particular worth mentioning VDMPad [12]. VDMPad has an ability to explore the functionality of a VDM-SL model for example using a Read-Eval-Print Loop (REPL) capability. This lightweight approach for debugging does not require establishing new debug configurations etc. so it makes it faster

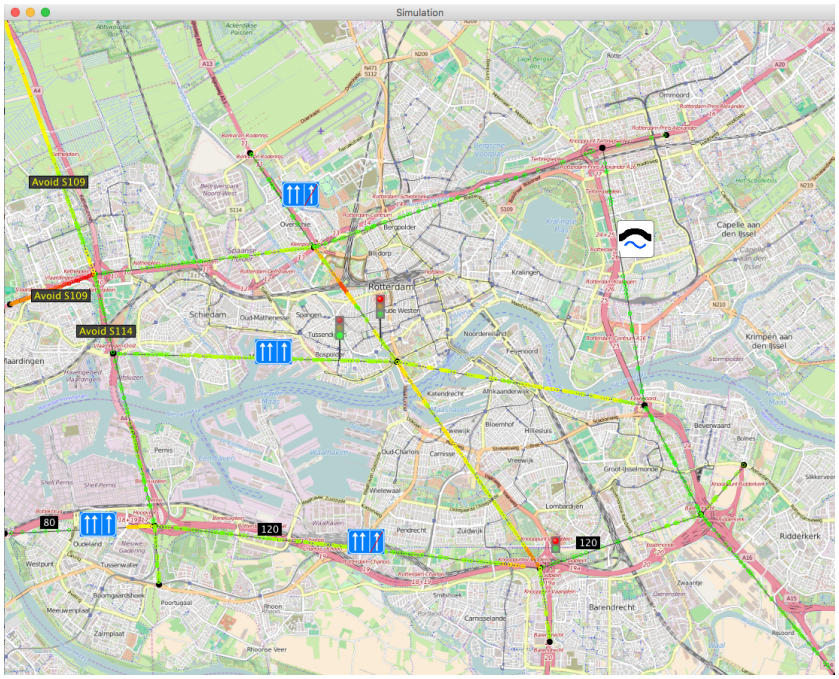


Fig. 8. Screenshot from the TEMPO simulation.

for the users to explore the behaviour. In addition it is worth mentioning that VDMPad enables a graphical view of VDM values for example instance variables which can be convenient for newcomers. The REPL functionality is also enabled in the recent WebIDE which has more IDE capabilities than VDMPad [15]. However, neither of these tools is able to combine executable code with the model and they are not able to monitor selected instance variables enabled in the work presented here. A new tool called ViennaTalk enables the combination with SmallTalk code but again it does not support the implicit monitoring presented here [13].

6 Concluding Remarks and Future Work

In this paper we have presented a new extension to the Overture tool that enables control of the Overture interpreter through a server and a protocol. The extension also features implicit monitoring of model variables, which allows UI interfaces to connect to a VDM model without needing to pollute the model with explicit UI calls. We have validated our extension by developing a UI application to monitor values in a traffic management system model.

The new extension offers greater flexibility in development of model UIs, as it allows the use of any technology that can connect to web sockets. In particular, modern web technologies such as JavaScript and HTML5 can be employed to develop better looking UIs in a faster way than the traditional Swing approach employed so far.

The protocol-based approach also enables a much cleaner decoupling between the model that represents the application and the UI. This leads to an easier separation of concerns between the model and the UI. Furthermore, both model and the UI can be evolved independently. The model does not need to be aware of the UI and the UI can rely on the protocol to act as an interface with the model.

Because both model and the UI can be developed independently, it is easier to go from a UI prototype to a production quality UI. As such, once the modelling phase is over and one begins system implementation, the UI code can be reused.

In this paper we have presented UIs focused on displaying numerical model data, as that was the primary need of the TEMPO project, which supported the development of the extension. However, the extension can enable the construction of other kinds of UIs. Any variable can be subscribed to, so it is possible to display various kinds of data. However, for complex structured data types such as records or classes, it may be necessary to write code to adequately render the data. Alternatively, the ASCII VDM representation of the data type may be used.

In the future, we would like to further validate our approach through the development of additional UIs. In particular, we would like to enrich the current collection of standard examples with user interfaces developed with this extension. Thus, the examples can showcase a new feature of Overture and the UIs themselves can serve as documentation for the extension and inspiration for users. To fully support this effort, we also hope to complete development of the protocol by supporting the `Execute` message and enabling two-way control between UI and model.

Finally, it would be worth investigating the possibility of combining our extension with the Overture code generator so that the UI code is included in the code generation process where it would be automatically modified in order to control the generated code instead of the model.

Acknowledgments Our current work is partially supported by the European Commission as a small experiment selected for funding by the CPSE Labs innovation action (grant number 644400). Thanks also go to the anonymous reviewers and Skip Balk, Pieter Buzing, John Skovsgaard and Henrik Nymann Jensen for their contributions to the different extensions.

References

1. Crockford, D.: The application/json media type for javascript object notation (JSON). RFC 4627, Internet Engineering Task Force (July 2006)
2. The Electron website. `http://electron.atom.io/` (2016)
3. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Computing Surveys (CSUR) 35(2), 114–131 (2003)
4. Fitzgerald, J.S., Larsen, P.G., Verhoef, M.: Vienna Development Method. Wiley Encyclopedia of Computer Science and Engineering (2008), edited by Benjamin Wah, John Wiley & Sons, Inc.
5. Fröhlich, B., Larsen, P.G.: Combining VDM-SL Specifications with C++ Code. In: Gaudel, M.C., Woodcock, J. (eds.) FME'96: Industrial Benefit and Advances in Formal Methods. pp. 179–194. Springer-Verlag (March 1996)

6. The Jetty website. http://www.eclipse.org/jetty/ (2016)
7. Krasner, G.E., Pope, S.T., et al.: A description of the model-view-controller user interface paradigm in the smalltalk-80 system. Journal of object oriented programming 1(3), 26–49 (1988)
8. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The Overture Initiative – Integrating Tools for VDM. SIGSOFT Softw. Eng. Notes 35(1), 1–6 (January 2010), http://doi.acm.org/10.1145/1668862.1668864
9. Larsen, P.G., Lausdahl, K., Tran-Jørgensen, P.W.V., Ribeiro, A., Wolff, S., Battle, N.: Overture VDM-10 Tool Support: User Guide. Tech. Rep. TR-2010-02, The Overture Initiative, www.overturetool.org (May 2010)
10. Lausdahl, K., Larsen, P.G., Battle, N.: A Deterministic Interpreter Simulating A Distributed real time system using VDM. In: Qin, S., Qiu, Z. (eds.) Proceedings of the 13th international conference on Formal methods and software engineering. Lecture Notes in Computer Science, vol. 6991, pp. 179–194. Springer-Verlag, Berlin, Heidelberg (October 2011), http://dl.acm.org/citation.cfm?id=2075089.2075107, ISBN 978-3-642-24558-9
11. Nielsen, C.B., Lausdahl, K., Larsen, P.G.: Combining VDM with Executable Code. In: Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) Abstract State Machines, Alloy, B, VDM, and Z. Lecture Notes in Computer Science, vol. 7316, pp. 266–279. Springer-Verlag, Berlin, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-30885-7_19, ISBN 978-3-642-30884-0
12. Oda, T., Araki, K., Larsen, P.G.: VDMPad: a Lightweight IDE for Exploratory VDM-SL Specification. In: Plat, N., Gnesi, S. (eds.) FormaliSE 2015. pp. 33–39. In connection with ICSE 2015, Florence (May 2015)
13. Oda, T., Araki, K., Larsen, P.G.: ViennaTalk and Assertch: Building Lightweight Formal Methods Environments on Pharo 4. In: Proceedings of the International Workshop on Smalltalk Technologies. pp. 4:1–4:7. Prague, Czech Republic (Aug 2016)
14. Plat, N., Larsen, P.G., Pierce, K.: Modelling Collaborative Systems and Automated Negotiations. In: Larsen, P.G., Plat, N. (eds.) The 14th Overture Workshop: Towards Analytical Tool Chains. pp. 108–123. Aarhus University, Department of Engineering, Cyprus, Greece (November 2016), ECE-TR-28
15. Reimer, R.S., Saaby, K.D.: An Open-Source Web IDE for VDM-SL. Master's thesis, Department of Engineering, Aarhus University, Denmark (May 2016)

Peter Gorm Larsen, Nico Plat and Nick Battle (Eds), The 14th Overture Workshop, 2016

Department of Engineering
Aarhus University
Inge Lehmanns Gade 10
8000 Aarhus
Denmark

Tel.: +45 8715 0000