SCHOOL OF
COMPUTING

Title: Proceedings of the 17th Overture Workshop

Names: Carl Gamble and Luid Diogo Couto (Eds.)

TECHNICAL REPORT SERIES

No. CS-TR- 1530 - 2019

TECHNICAL REPORT SERIES

No: CS-TR- 1530 **Date: September 2019**

Title: Proceedings of the 17th Overture Workshop

Authors: Carl Gamble and Luid Diogo Couto (Eds.)

Abstract: The 17th Overture Workshop is held on 07 October 2019 in association with the the 3rd World Congress on Formal Methods (FM2019). The 17th Overture Workshop is the latest in a series of workshops around the Vienna Development Method (VDM), the open-source project Overture, and related tools and formalisms. VDM is one of the best established formal methods for systems development. A lively community of researchers and practitioners in academia and industry has grown around the modelling languages (VDM- SL, VDM++, VDM-RT, CML) and tools (VDMTools, Overture, Crescendo, Symphony, and the INTO-CPS chain). Together, these provide a platform for work on modelling and analysis technology that includes static and dynamic analysis, test generation, execution October 2019

Printed and published by Newcastle University,
School of Computing, Urban Sciences Building,
Newcastle upon Tyne, NE4 5TG, England.

Bibliographical Details

Title and Authors: Proceedings of the 17th Overture Workshop
Carl Gamble and Luid Diogo Couto (Eds.)

NEWCASTLE UNIVERSITY
School of Computing. Technical Report Series. CS-TR- 1530

Abstract: The 17th Overture Workshop is held on 07 October 2019 in association with the the 3rd World Congress on Formal Methods (FM2019).

The 17th Overture Workshop is the latest in a series of workshops around the Vienna Development Method (VDM), the open-source project Overture, and related tools and formalisms. VDM is one of the best established formal methods for systems development. A lively community of researchers and practitioners in academia and industry has grown around the modelling languages (VDM- SL, VDM++, VDM-RT, CML) and tools (VDMTools, Overture, Crescendo, Symphony, and the INTO-CPS chain). Together, these provide a platform for work on modelling and analysis technology that includes static and dynamic analysis, test generation, execution October 2019

About the authors: Carl Gamble is a Senior Software Developer at Asset55 and a guest researcher in the School of Computing at Newcastle University, UK. He received his PhD from Newcastle University and was a founding member of their Cyber-Physical Systems (CPS) Lab. His research interests focus on the modelling of CPSs, predominantly through the use of multi-modelling, and the development of tools and techniques to aid in the optimisation of CPS designs.

Luis Diogo Couto is a Software Engineer at Forcepoint. He received his PhD in 2015 from Aarhus University, DK. He has a broad range of interests, including formal methods, cybersecurity, software architecture, and cloud computing. Of late, he is particularly interested in how teams of people work together to develop complex software systems. Luis is a member of the VDM Language Board and a contributor to the Overture tool.

Suggested keywords: VDM, Overture, formal methods

Pre-Proceedings of the 17th Overture Workshop

Luis Diogo Couto and Carl Gamble (Editors)

October 2019

Preface

The 17th Overture Workshop is held on 07 October 2019 in association with the the 3rd World Congress on Formal Methods (FM2019).

The 17th Overture Workshop is the latest in a series of workshops around the Vienna Development Method (VDM), the open-source project Overture, and related tools and formalisms. VDM is one of the best established formal methods for systems development. A lively community of researchers and practitioners in academia and industry has grown around the modelling languages (VDM-SL, VDM++, VDM-RT, CML) and tools (VDMTools, Overture, Crescendo, Symphony, and the INTO-CPS chain). Together, these provide a platform for work on modelling and analysis technology that includes static and dynamic analysis, test generation, execution support, and model checking.

October 2019

Luis Diogo Couto
Carl Gamble

Program Committee

Nick Battle	Newcastle University, UK
Luis Diogo Couto	Forcepoint, Ireland (Chair)
Leo Freitas	Newcastle University, UK
John Fitzgerald	Newcastle University, UK
Carl Gamble	Newcastle University, United Kingdom (Chair)
Fuyuki. Ishikawa	NII, Japan
Peter Gorm Larsen	Aarhus University, Denmark
Paolo Masci	National Institute of Aerospace (NIA), USA
Ken Pierce	Newcastle University, UK
Peter W. V. Tran-Jørgensen	Aarhus University, Denmark

Contents

Towards Graphical Configuration in the INTO-CPS Application

Christian Møldrup Legaard[1], Casper Thule[1], Peter Gorm Larsen[1]

DIGIT, Department of Engineering, Aarhus University, Aarhus, Denmark,
201408498@post.au.dk, casper.thule@eng.au.dk, pgl@eng.au.dk

Abstract. The INTO-CPS Application is a common interface used to access and create different artefacts in the development of a Cyber-Physical System (CPS) making use of a collection of tools centred around the Functional Mockup Interface (FMI). For the configuration of the composition and adaptation of Functional Mockup Units (FMUs) the application currently imports this information as a multi-model from a SysML model made using the Modelio tool. The contribution presented in this paper is the addition of a unified graphical editor to the INTO-CPS application which eliminates the need for external tools. This is accomplished by using a standard called System Structure and Parameterisation (SSP) complementing FMI, since it is more expressive than the current multi-model representation. SSP enables the description of co-simulation scenarios in a graphical form including semantic adaptation of FMUs using SSP's extension capabilities.

1 Introduction

In Cyber-Physical Systems (CPSs), computing and physical processes interact closely. Their effective design therefore requires methods and tools that bring together the products of diverse engineering disciplines. Without such tools it would be difficult to gain confidence in the system-level consequences of design decisions made in any one domain, and it would be challenging to manage trade-offs between them.

The INTO-CPS project has created a tool chain supporting different disciplines such as software, mechatronic and control engineering that have evolved notations and theories tailored to their specific domains. It would be undesirable to suppress this diversity by enforcing uniform general-purpose models [3,4,11,12,13].

The configuration of co-simulations has, in the past, been carried out using a special CPS profile for the SysML support inside the Modelio tool [2]. The current SysML support also includes diagrams for Design Space Exploration (DSE) where different alternative designs can be explored automatically over different parameters [5]. However, the SysML CPS profile does not yet support hierarchical co-simulations [16] or other semantic adaptations [7]. It would be useful to examine if these concepts could be supported by a single unified editor inside the INTO-CPS application. We take inspi-

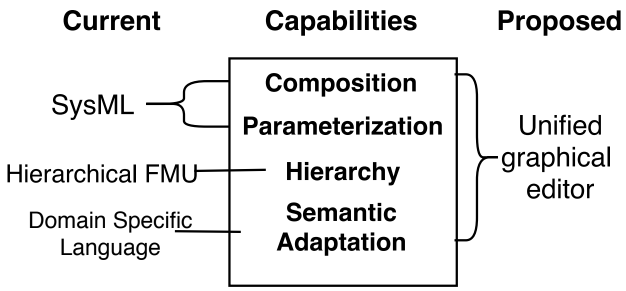


Fig. 1. The migration from current capabilities to the proposed graphical solution/representation.

ration from the modelling tools 20-sim[1], OMEdit[2] and Simulink[3] in order to come up with new ideas for this. Figure 1 depicts the tools currently involved to access different capabilities of the application, as well as the proposed graphical editor which unifies access to these.

The rest of this paper starts with background information for the reader in Section 2. Afterwards Section 3 illustrates how the graphical connections are made directly inside the INTO-CPS Application. Then Section 4 provides an introduction to the overall design of the graphical editor. Finally, Section 5 provides a few concluding remarks while Section 6 provides the future directions expected for this work.

2 Background

It is important to make a distinction between what the FMI standard defines and the information required to form a valid co-simulation scenario as illustrated on Figure 2. The standard is solely concerned with specifying the interface of an individual FMU. A co-simulation scenario describes what instances of FMUs exist, how they are connected, their parameters and their experimental frame [8]. In order to provide the reader with sufficient background about the existing technologies Section 2.1 briefly introduces the existing SysML diagrams to describe the composition of FMUs, Section 2.2 provides a brief introduction to separate existing work on semantic adaptation including hierarchical compositions and Section 2.3 provides a brief introduction to the new SSP standard.

2.1 Existing SysML Connection Diagram

A SysML Internal Block Diagram in the context of INTO-CPS is referred to as a *Connection Diagram*. It is used to define the composition and connectivity of several FMUs. The connection diagram contains instances of FMUs (SysML Block instances) which

[1] `https://www.20sim.com/`

[2] `https://openmodelica.org/?id=78:omconnectioneditoromedit&catid=10:main-category`

[3] `https://se.mathworks.com/products/simulink.html`

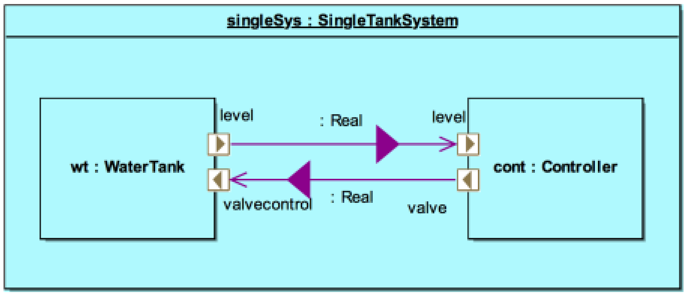


Fig. 2. Connection diagram for the WaterTank example to connect FMUs.

are connected together by means of connectors to the input/output ports of the FMU instances. The connection diagram also enables association of existing FMUs to the block instances for a co-simulation.

Using a general-purpose modeling language such as SysML for defining simulation scenarios, comes with an inherent challenge. The lack of simulation specific concepts in the language makes expression of these more cumbersome and error prone. First, it is required that the users memorise the exact way these concepts must be expressed in SysML. Secondly, it limits the assistance the tool can provide the user, due to its limited understanding of domain specific concepts. These issues are amplified by the fact that errors are only detected when the SysML model is imported in the tool.

In the DSE diagrams inside the SysML profile the exploration capabilities makes use of parameters for the different FMUs, so this is also important for the work we describe here.

2.2 Semantic Adaptations

While the FMI standard defines a common format for exchanging FMUs it does not guarantee the absence of so called *interaction mismatches* when these are composed [7]. The goal of semantic adaptations (SA), is to allow these mismatches to be corrected without the need to involve the original author of the FMU.

A concrete example where this would be if *valve* input of the *WaterTank* accepted a bool instead of a real. In this case it would not be possible to connect the *valvecontrol* and *valve* ports. A SA could be used to apply a threshold to the *valve* output in order to convert it into a bool, as seen in figure 3.

Conceptually, a SA can be thought of as an *external* FMU, which wraps around one or more *internal* FMUs. The external FMU is responsible for managing the execution of the internal FMUs. In the previous example the external FMU would simply pass all signals directly, except for the *valve*, which would be thresholded before being written to the *valve* output of the external FMU. The approach grants a great deal of flexibility, allowing a large number of adaptations to be implemented. In addition, it allows these to be applied in a hierarchical manner, e.g. "on top of" each other.

As described in Section 1 the SysML CPS profile does not yet support semantic adaptations [7]. In general such semantics adaptations are currently established by a

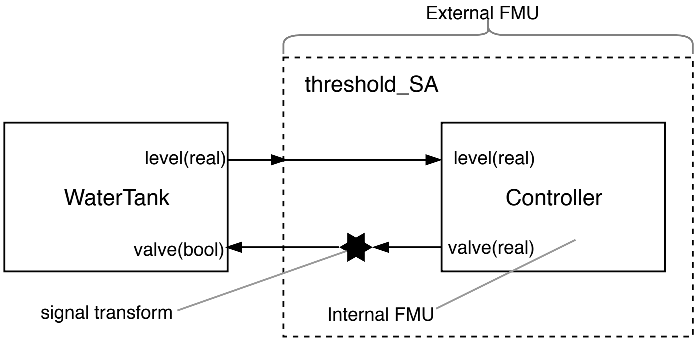


Fig. 3. Example of how semantic adaptation may be applied to correct signal data mismatch.

separate Domain Specific Language (DSL). This is introduced in order to perform different kinds of adaptations to existing FMUs that enable their behaviour to be adjusted in different ways. However, the DSL from [7] also enables support for hierarchical co-simulations. A different approach for hierarchical co-simulations has been introduced in [16].

2.3 System Structure And Parametrisation

The central goal of the INTO-CPS Application is to enable the composition and co-simulation of a system consisting of one or more FMUs. The use of the FMI standard enables exchange at the level of the individual FMUs. However, it does not describe an exchange format for composing and parametrising a simulation scenario consisting of multiple FMUs. Due to the lack of an established standard, the INTO-CPS Application uses its own ad-hoc format referred to as the *multi-model* format.

Recently a new companion standard to FMI was published: the *System Structure and Parameterization* (SSP) standard [14][4]. The standard uses a zip-based packaging format similar to FMI, where multiple artefacts are bundled into a single package referred to as a *SSP package* – each representing a ready to simulate system. An obvious benefit of adopting this standard is exchangeability of complete systems between tools. Currently only a few such as OpenModelica and *FMPy*[5], however given its close relation to FMI, it seems likely that more tools will adopt it. An example of what the running example may look like when bundled as a package is seen on Figure 4.

The standard covers the functionality present in the existing format, but also adds several capabilities which are useful for the INTO-CPS Application. Additionally, there are also some important semantic differences between the existing format and SSP. Ultimately, some of these influence the interaction in the GUI editor, as such the most relevant differences are highlighted below.

Components and Hierarchy: While the standard is closely aligned with the FMI it is not limited to describing systems purely consisting of FMUs. Instead the standard

[4] https://ssp-standard.org/

[5] https://github.com/CATIA-Systems/FMPy

```
\---single_tank.ssp         : collection of multi models
    | SystemStructure.ssd   : default multi model
    | VarA.ssd              : alternative multi model
    | ParameterSet.ssv      : set of parameters
    | ParameterMappings.ssm : binding params to instances
    | SignalDictionary.ssb  : mechanism to group signals
    |
    +---documentation
    | index.html
    |
    +---extra               : extra files extension mechanism
    |
    \---resources           : implementations of components
      WaterTank.fmu
      Controller.fmu
      subsystem.ssp         : potential subsystem
```

Fig. 4. Example of the file structure of the running example stored as a SSP package. The comments on the side describe the role of the file.

establishes the concept of a *component* – an atomic block of the system, which is backed by some underlying implementation. For example, a component may reference a FMU as its implementation. A component may itself be a SSP package, which in this context, is referred to as a *sub-system*. The latter enables the creation of hierarchies of components.

Parameterization: There is an important difference between how parameters are treated in a multi-model and by SSP. In the former all parameters are defined inline, there is no mechanism to reference a “shared” parameter from multiple components. SSP on the other hand provides the *parameter sets* and *parameter mappings* concepts which makes it possible to declare a set of parameters and then bind these to the concrete components[6].

Extensibility: Similar to FMI the standard is largely based on XML and is designed to be extensible through three types of extension mechanisms: Annotations, extra files and MIME type-based format dispatch. This allows for the creation of new extensions to the standard referred to as *layered standards*. A potential use case of this is to extend the standard with support for semantic adaptations.

3 Graphical Editor

Currently, several external tools are required for the creation and composition of co-simulation scenarios. The primary goal of the development of an integrated graphical editor is to integrate the functionality of these tools and provide a more efficient workflow. We refer to this editor as the *Graphical Editor*.

[6] Inside the SSP standard these are referred to as System Structure Parameter Values and System Structure Parameter Mappings.

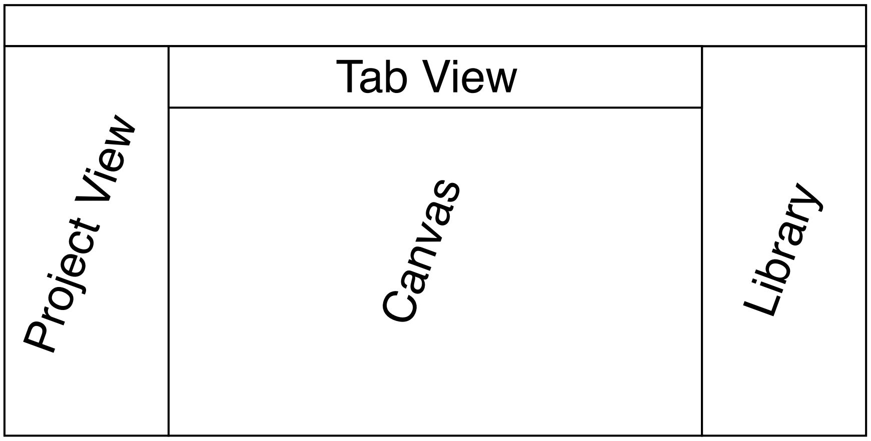


Fig. 5. Mock-up of the envisioned editor annotated with the names of its constituent parts.

The editor aims to provide an interaction that is familiar to users of modelling software such as 20-sim, OMEdit or Simulink. Common for these is that they provide a block-based design flow which allows the user to compose scenarios by dragging components from a library onto a canvas. The scenarios are further elaborated by dragging connections between compatible ports.

An outline of the editor and its different panels can be seen in Figure 5. The functionality of each of these is described in the subsections below.

3.1 Project View

The *project view* provides a tree view of the current project's *artefacts*. An example of these, and the relation between project view and the canvas are is depicted in Figure 6. As shown these are categorised based on their type such as components and connections. Artefacts may themselves be hierarchically composed as is the case of the "cont" component. In these cases they may be expanded in the tree to reveal their internal contents.

3.2 Composition

A central part of defining a simulation scenario is defining instances of components and how these are connected. Instances are created by dragging an item from the *components-group* into the diagram as seen in figure 7. Naturally every instance must be assigned a name. This may be handled by either prompting the user upon creation or automatically resolving the instance name based on the component name as seen in figure 7a.

After having instantiated the needed components the connections between their ports may be established. Ports are represented by symbols reminiscent of a less-than sign with the name of the port next to it. An input is indicated by a symbol pointing

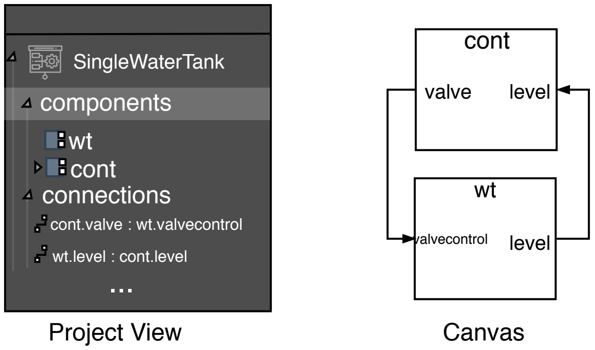


Fig. 6. Project view and canvas relationship

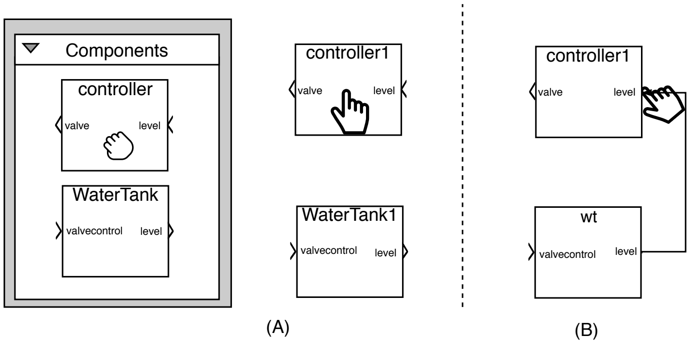


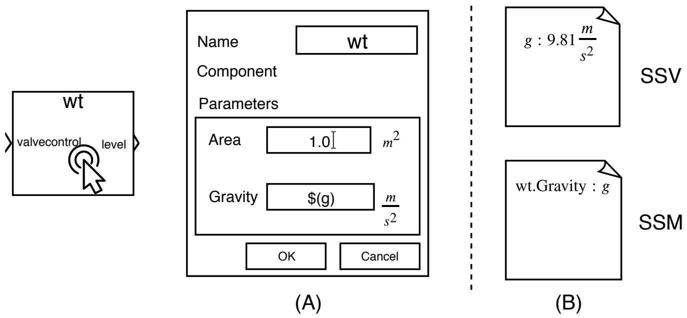
Fig. 7. Illustration depicting how the inputs and outputs of FMUs may be connected graphically.

inwards whereas an output is indicated by a symbol pointing outwards. This representation is proposed as it is compact and shows the name and direction of a port.

A pair of ports are connected by hovering the mouse over an unconnected output port, initiating a drag and finally releasing the drag over an unconnected input port. To limit the probability of connecting ports incorrectly the GUI will not accept a connection from the source port to an incompatible target. An simple rule is to check if the units of the two ports matches, but it is possible to imagine more elaborate validation procedures.

3.3 Parameterisation

A component may expose several parameters allowing its behaviour to be adjusted without the need to modify its implementation. This process is referred to as parameterisation. In the context of the running example a parameter of the water tank component would be its area or the gravitational constant. Double clicking on a component instance will open an *configuration-window*, as seen in figure 8. The window contains a section which list all available parameters of the component. A concrete value may be entered for a given parameter, or it may be bound to a parameter of the parameter set.



Name: wt

Component

Parameters

Area: 1.0 m^2

Gravity: $(g) $\frac{m}{s^2}$

Fig. 8. Parameterization of a component. A *configuration-window* is opened when a component is double clicked.

The editor allows such bindings to be defined directly from the configuration-window. In the context of the editor these are referred to as *parameter-binding*. The bindings are defined using a special notation: *$(parameter)*, as seen on 8a.

The bound parameters of the components now refer to the value of the corresponding parameters of the parameter set. An example of how this may be used would be if multiple water tanks existed, in this case they would all depend on the gravitational acceleration, g, experienced in their environment. Using the binding mechanism the value of g may adjusted in the parameter set to examine its impact on the whole system's performance.

From a users standpoint it would be useful to know the valid ranges and constraints of a parameter's values. This would allow the editor to inform the user of any inconsistencies during configuration rather than resulting in error during simulation. The SSP standard itself does not define any mechanism for specifying these. However, due to its extensibility it would be possible to implement a layered standard for this purpose.

3.4 Representing Hierarchy

A central feature of the SSP standard is its ability to describe hierarchical systems. Specifically it allows for the composition of a system from multiple subsystems, which may themselves be SSP-packages.

More related to the editor itself is how the hierarchy of components is represented. 20-sim and Simulink both allow the creation and navigation of this hierarchy graphically. Both use a very similar approach, the primary differences being notation. An an example of what this may look like in the editor is shown in Figure 9.

Navigation is straightforward, double clicking a subsystem changes the view such that it shows the subsystems internals. We refer to this as *opening* a subsystem. This process can be repeated to reach deeper into the hierarchy. Given the lack of a parent component to click on, navigating upwards is done using a toolbar button.

The process of creating subsystems from existing components may be done in the following way. First one or more components are selected, then the "Subsystem from Selection" item is chosen from the context menu. This results is a new subsystem in place of the selected components. The subsystem contains the now replaced components.

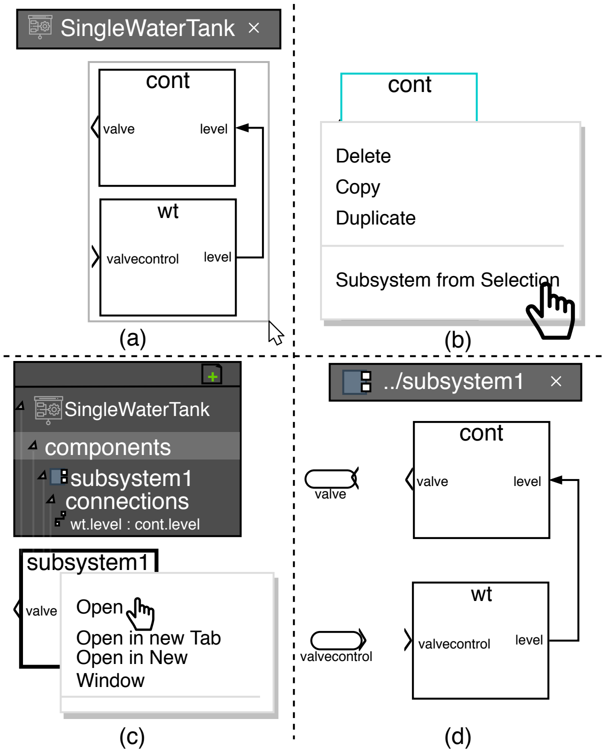


Fig. 9. Proposed method to create subsystem from a selection of components. (a) components are selected. (b) "Subsystem from Selection" is selected in the context menu. (c) the two components have been converted into a single subsystem, which is double clicked to access the internal view. (d) internal view of the subsystem.

3.5 Semantic Adaptations

A solution is proposed that allows different types of semantic adaptations graphically using the familiar drag and drop system, as shown in Figure 10. A group containing entries corresponding to commonly used types of adaptations is added to the library. An adaptation can be dragged onto a component thereby "decorating" it with the adaptation. The parameters of an adaptation may be configured in a similar fashion to how a component is parameterised.

The suitability of this approach relies on the available adaptations being sufficiently modular, such that they can be composed to cover most needs. Alternatively, its possible to imagine an "DSL"-adaptation which allows for arbitrary adaptations made in a domain specific language.

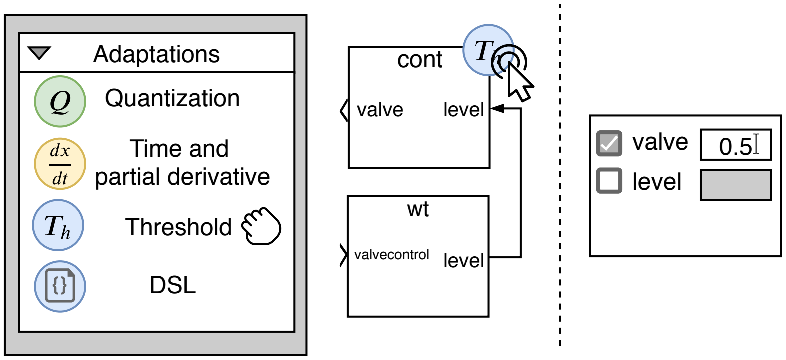


Fig. 10. applying and configuring of semantic adaptations.

4 Graphical Editor Design

Similar to the existing application the graphical editor is written in the Typescript[7] language on top of the Angular framework [1]. Angular is a front end framework consisting of several libraries and utility programs that aid in the development of web applications.

The editor implements the *model-view-controller* (MVC) design pattern [6]. This emphasizes the separation of data of the application from how its represented graphically. It allows the data of the model to be represented by multiple views. A specific example of two views that share the same data are the project and canvas views, shown earlier in Figure 5.

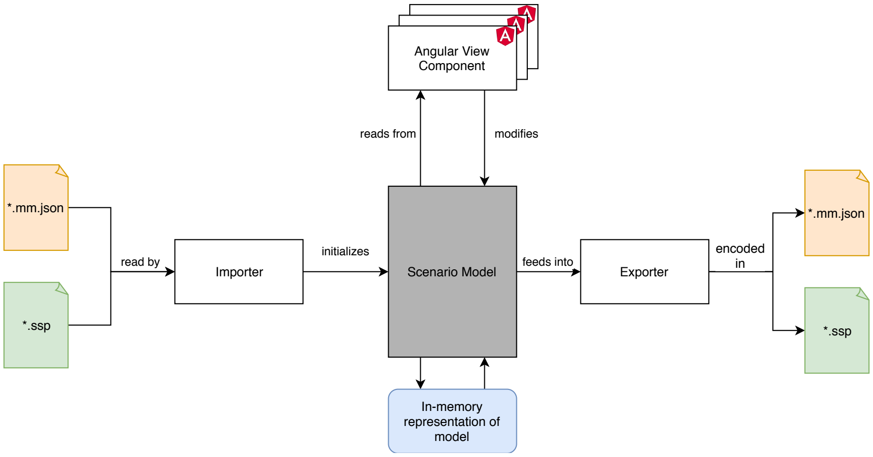


Fig. 11. Flow importing and exporting multi-models and SSPs

[7] `https://www.typescriptlang.org/`

A conceptual view of the architecture can be seen on Figure 11. Starting from the left we see that a co-simulation scenario can be imported from a file encoded as either the current multi model format or the SSP format. The file is parsed and used to instantiate an in-memory representation of the scenario referred to as the *scenario model*. The model has several uses. It provides an abstraction of the underlying model to components such as the editor. The editor reads and generates a graphical representation of the model. Conversely, changes to the scenario made through the editor are propagated back to the model. Finally, the model may be exported to a format suitable for exchange, such as multi-model or SSP.

4.1 Rendering of components

The canvas-view is built on top of the javascript library mxGraph[8]. It provides the functionality of creating interactive graphs, creating connectors, undo-redo, automatic layout and much more.

4.2 Modules

Specifically the implementations consists of several components, referred to as a *Modules* within the context of Angular[9]. A module is a collection of components and services which provide a coherent set of functionality to the application. In this setting the work presented in this paper naturally needs to fit into the recently developed cloud version of the INTO-CPS Application [15].

To facilitate version management the modules are published on NPM under the namespace @*into-cps*. The concrete packages are listed below:

@*into-cps/graphical-editor*
@*into-cps/scenario-model*
@*into-cps/model-serialization*

5 Concluding Remarks

This paper has demonstrated our work in progress towards supporting SSP in order to provide graphical editing support for the composition (and adaptation) of FMUs directly inside the INTO-CPS Application. We envisage that this will increase the user-friendliness of the INTO-CPS Application since it requires one external tool less and it will be able to support more than what is possible in the SysML CPS profile. Furthermore, there are additional ideas (see below) for how additional improvements can be made for this technology.

6 Future Work

The work presented in this paper is at an early stage and as a consequence there are many extension possibilities that we envisage in the future. Below the most obvious directions are listed.

[8] `https://github.com/jgraph/mxgraph`

[9] `https://angular.io/guide/architecture-modules`

6.1 Generating template of a component

In the existing SysML CPS profile there is a concept of Architecture Structure Diagrams. Inside these it is possible to include all the FMUs included in a particular project. The interface for each of these can be described here and as a consequence it is possible to export model descriptions for each FMU. Such model descriptions can then subsequently be imported by individual modelling and simulation tools such as Overture [10]. We envisage that it also will be easy to include this capability inside the graphical editor. Such model descriptions can also be included as new libraries that others will be able to make use of in their co-simulation composition.

6.2 Implement SSP support in Maestro

The added value of the editor relies on the co-simulation Maestro supporting the SSP standard. A central question is how to support the simulation of hierarchical systems. One potential approach is turning each subsystem into a FMUs. This approach is described in Hierarchical FMU [16] and the DSL for hierarchical co-simulation [7].

6.3 Runtime validation

One of the uses of co-simulation is to validate a system's behaviour during runtime to detect and correct any potential design flaws, as early in the development cycle as possible. Currently there is no mechanism built in to INTO-CPS to support this task. It is useful to investigate whether a suitable method exists for providing runtime validation during simulation. Potential inspiration may be drawn from [9].

6.4 Simplify design space exploration and adapt to SSP

In the context of INTO-CPS design space exploration is a mechanism that allows a scenario to be run multiple times with different combinations of parameters. The performance of each run is evaluated by means one or more cost function, each defined in its own Python script. The scripts takes as input the traces resulting from the INTO-CPS co-simulation, its parameters and global information such as the step size.

Currently this information is passed to the scripts by the means of positional arguments and therefore requires the user to manually define the ordering of the individual arguments. This obviously does not scale well with the number of parameters to the scripts.

To eliminate the need for manually ordering a reference to an object containing the relevant information may be passed instead. This would allow a script to access variables using semantically meaningful names such as *data.traces.wt.level* instead of *sys.args[3].*

6.5 Package manager for components

The tools 20-sim, OMEdit and Simulink all come with an extensive library of components which provide much of the functionality a user would otherwise have to implement themselves. The same benefits are also realised in INTO-CPS application where open-source libraries and framework are used extensively.

While the SSP standard provides the format for exchange it does not describe the infrastructure to share these. It would be interesting to investigate effective mechanisms for publishing and managing components. Potentially inspiration can be drawn from a package manager such as the one used in the application, npm[10].

Acknowledgements

We are grateful to the Poul Due Jensen Foundation, which has supported the establishment of a new Centre for Digital Twin Technology at Aarhus University, which will take forward the principles, tools and applications of the engineering of digital twins. We also acknowledge EU for funding the INTO-CPS project (grant agreement number 644047) which was the original source of funding for the INTO-CPS Application. Finally, we thank the reviewers for their throughout feedback.

References

1. Ang, K.H., Chong, G., Li, Y.: Pid control system analysis, design, and technology. IEEE Transactions on Control Systems Technology 13(4), 559–576 (July 2005)
2. Bagnato, A., Brosse, E., Quadri, I., Sadovykh, A.: SysML for Modeling Co-simulation Orchestration over FMI: the INTO-CPS Approach. Ada User Journa 37(4), 215–218 (December 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)
4. Fitzgerald, J., Gamble, C., Payne, R., Larsen, P.G., Basagiannis, S., Mady, A.E.D.: Collaborative Model-based Systems Engineering for Cyber-Physical Systems – a Case Study in Building Automation. In: Proc. INCOSE Intl. Symp. on Systems Engineering. Edinburgh, Scotland (July 2016)
5. Foldager, F., Balling, O., Gamble, C., Larsen, P.G., Boel, M., Green, O.: Design Space Exploration in the Development of Agricultural Robots. In: AgEng conference. Wageningen, The Netherlands (July 2018)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
7. Gomes, C., Meyers, B., Denil, J., Thule, C., Lausdahl, K., Vangheluwe, H., Meulenaere, P.D.: Semantic adaptation for FMI co-simulation with hierarchical simulators. SIMULATION 0(0), 0037549718759775 (2018), https://doi.org/10.1177/0037549718759775
8. Gomes, C., Thule, C., Broman, D., Larsen, P.G., Vangheluwe, H.: Co-simulation: a Survey. ACM Comput. Surv. 51(3), 49:1–49:33 (May 2018)
9. Havelund, K., Roşu, G.: An overview of the runtime verification tool java pathexplorer. Form. Methods Syst. Des. 24(2), 189–215 (Mar 2004), https://doi.org/10.1023/B:FORM.0000017721.39909.4b

[10] https://www.npmjs.com/

10. 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
11. Larsen, P.G., Fitzgerald, J., Woodcock, J., Fritzson, P., Brauer, J., Kleijn, C., Lecomte, T., Pfeil, M., Green, O., Basagiannis, S., Sadovykh, A.: Integrated Tool Chain for Model-based Design of Cyber-Physical Systems: The INTO-CPS Project. In: CPS Data Workshop. Vienna, Austria (April 2016)
12. Larsen, P.G., Fitzgerald, J., Woodcock, J., Lecomte, T.: Trustworthy Cyber-Physical Systems Engineering, chap. Chapter 8: Collaborative Modelling and Simulation for Cyber-Physical Systems. Chapman and Hall/CRC (September 2016), ISBN 9781498742450
13. Larsen, P.G., Fitzgerald, J., Woodcock, J., Nilsson, R., Gamble, C., Foster, S.: Towards Semantically Integrated Models and Tools for Cyber-Physical Systems Design. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation, Proc 7th Intl. Symp. Lecture Notes in Computer Science, vol. 9953, pp. 171–186. Springer International Publishing (2016)
14. Modelica Association Project SSP: System Structure and Parameterization Specification Document, Version 1.0. Modelica Association, Linköping, Sweden (2019), http://www.ssp-standard.org
15. Rasmussen, M.B., Thule, C., Macedo, H.D., Larsen, P.G.: Moving the INTO-CPS Application to the Cloud. In: The 17th Overture workshop. Porto, Portugal (October 2019)
16. Thule, C., Lausdahl, K., Gomes, C., Meisl, G., Larsen, P.G.: Maestro: The INTO-CPS co-simulation framework. Simulation Modelling Practice and Theory 92, 45 – 61 (2019), http://www.sciencedirect.com/science/article/pii/S1569190X1830193X

Towards a Static Check of FMUs in VDM-SL

Nick Battle[1], Casper Thule[2], Cláudio Gomes[3], Hugo Daniel Macedo[2], and Peter Gorm Larsen[2]

[1] Independent, nick.battle@acm.org
[2] DIGIT, Department of Engineering, Aarhus University, Aarhus, Denmark, {casper.thule,hdm,pgl@eng.au.dk}
[3] University of Antwerpen, claudio.gomes@uantwerp.be

Abstract. In order to ensure that the co-simulation of Cyber-Physical Systems (CPSs) is possible with as wide a variety of tools as possible, a standard called the Functional Mockup Interface (FMI) has been defined. The FMI provides the means to compute the overall behavior of a coupled system by the coordination and communication of simulators, each responsible for a part of the system. The contribution presented in this paper is an initial formal model of the FMI standard using the VDM Specification Language. Early results suggest that the FMI standard defines a number of FMU static constraints that are not enforced by many of the tools that are able to export such FMUs.

1 Introduction

In Cyber-Physical Systems (CPSs), computing and physical processes interact closely. Their effective design therefore requires methods and tools that bring together the products of diverse engineering disciplines [21,20]. This has been the motivation for establishing common co-simulation technologies to enable full system evaluation through the interconnection of individual simulators [10]. One of the most widely adopted simulation interfaces is the Functional Mock-up Interface (FMI) standard [3,2,7].

In the context of the FMI standard, the simulators are encapsulated with their corresponding models in an executable format. These are denoted as Functional Mock-up Units (FMUs). The FMI interface establishes which operations, and parameters, can be used to communicate with the FMUs.

While the FMI allows for the integration of many simulation tools, it under-specifies the interaction protocol with the simulators. That is, the same pair of simulators can synchronize data in different ways, each leading to a potentially different result. This is a feature and not a limitation: the FMI steering committee recognized that, in order to produce reliable results, different sets of simulators (and the corresponding models) may require different synchronization algorithms. In fact, finding the best algorithm for a particular co-simulation is one of the main research challenges in this field[1,8,9].

However, as recognised in a recent empirical survey [17], “The most acknowledged challenge is related to practical aspects. These include: faulty/incomplete implementations of the FMI standard, ambiguities/omissions in the specification and documentation of the FMUs, . . . ”. We argue that one of the contributing factors is the fact that the

FMI specification is semi-formal, and that the ambiguities lead to different implementations of FMUs. This is a challenge for the orchestration of a collection of FMUs in a semantically sound fashion [18].

Contribution. We propose to formalise what the FMI standard actually states, and this paper is an attempt at moving towards a common agreement about the static constraints for FMUs used for co-simulation in such a formal specification. The intention is to be able to release this as an open source tool that is able to rigorously test the static validity of FMUs. We present early empirical results regarding the application of this tool to the FMI Cross Check repository, the repository used to evaluate whether tools successfully support the standard. We hope to be able to later extend this with the different allowed dynamic semantics interpretations of the actual co-simulation engines.

The rest of this paper starts with background information for the reader in Section 2. This is followed by Section 3 which provides an overview of the VDM-SL formalisation and Section 4 which summarises the results discovered by using the model to analyse FMUs. Finally, Section 5 provides a few concluding remarks and comments on the future directions expected for this work.

2 Background

2.1 The Functional Mockup Interface

The FMI standard is purely concerned with specifying the packaging of, API interface to, and common configuration of individual FMUs. It does not define what particular FMUs model, or the different ways of coordinating the simulation of a collection of FMUs.

By offering a common standard for simulated components, the FMI standard promotes interoperability between a wide variety of tools that want to explore the creation of systems that are composed of multiple interacting components. Today[4] there are around 50 independent tools that conform to version 2.0 of the co-simulation part of the FMI standard. This includes the INTO-CPS tool chain which integrates several tools that are able to deal with FMUs [4,5,14,15,16].

The FMI standard has both static and dynamic semantics, though both are defined informally. The static semantics are concerned with the configuration of FMUs according to the rules defined in the standard. The dynamic semantics are concerned with the behavioural constraints placed on FMUs when a sequence of API calls are made to them, as defined in the standard. The current work is only concerned with the formal definition of the static semantics.

2.2 VDM Modelling

VDM is a long established formalism for the specification and analysis of discrete systems [13,6].

[4] From `https://fmi-standard.org/tools/`, as of August 2019.

A VDM specification comprises a set of data types, constant values, functions, state information and operations that describe the functional behaviour of a system. A specification relies heavily on constraints to define the correct behaviour of the model, for example via data type invariants, function preconditions and postconditions, state data invariants and so on.

A VDM function is an expression over its arguments which returns a result; if the same arguments are passed, the same result is returned. A type invariant is a total function over the possible values of the type, returning “true” for all valid values.

A VDM annotation is a comment in the source of the model which allows user-written plugins to observe values on the fly (but not change them) and affect the behaviour of analyses (e.g., suppressing or generating warnings). By annotating subclauses of a complex expression, a VDM function can be traced or report multiple problems without affecting the behaviour or meaning of the specification.

3 The VDM Model of the FMI Standard

The VDM model is based on version 2.0 of the FMI standard, dated 24th July 2014. A later version 3.0 of the standard is being worked on, but that has not yet been released. Most tools work with the 2.0 standard, so this is the most useful version to formalise initially.

The FMI standard defines how to configure individual FMUs and it also defines a common API for interacting with all FMUs. The VDM model currently only defines the constraints on the static configuration, though a formal specification of the API semantics is a natural extension of this work. The current work builds on earlier work [12] that defined the static semantics of part of the FMI standard.

3.1 The FMI v2.0 XSD Schema

The FMI standard defines an XML configuration file format for FMUs. The structure of the file is defined using an XSD schema with annotations, combined with an informal description of the meaning and constraints on the various fields in the document text. The XSD defines basic constraints, such as the xs:types of fields, and the minOccurs and maxOccurs attributes for repeating or optional elements and attributes. An example XML extract is shown in Listing 1.1.

Listing 1.1. Extract of an FMU configuration XML file.

```
<ScalarVariable
  name="h"
  valueReference="0"
  causality="output"
  variability="continuous"
  initial="exact">
    <Real
      start="1"
      declaredType="Position"/>
</ScalarVariable>
```

A tool has been developed in this work to parse FMU configuration XML files and convert them to the equivalent in VDM, so that specific FMU configurations may be checked automatically using VDM constraint checking tools. The FMI standard deliberately defines an XSD schema that is very simple, and this makes the VDM translation straightforward. The authors are not aware of any XML constraint checking technology that has the power and flexibility of a VDM model, especially as the VDM formalism has the potential to define the dynamic as well as static semantics.

3.2 The VDM-SL Modelling Approach

The VDM model is currently written in the VDM-SL dialect. This is the simplest dialect, without support for object orientation or concurrency, but it is compatible with more sophisticated dialects if their capabilities become essential. It is hoped that the simplicity of the VDM-SL dialect makes the formal FMI specification more readily accessible to domain experts who are not familiar with the VDM notation.

Since it currently only formalises the XSD schema, the VDM model is comprised entirely of type definitions with supporting invariant functions. The type definitions correspond to the XSD element definitions (and have the same names); the invariants correspond to the constraints that are added by the XSD, its informal annotations and descriptions in the main text. The value of the VDM specification is therefore almost entirely contained in the invariant functions, with their ability to unambiguously specify the rules concerning the construction of each XML element.

The most common way to construct a VDM model of a static system is to define types with invariants as an explicit property of each type. This means that it is not possible to construct a value of the type which does not meet the invariant constraints. Unfortunately, when using such a model to "test" real FMU XML configurations, the VDM tools stop when they encounter the first invariant violation (for example, a variable "start" attribute which is not between its "min" and "max" values). Although this is reasonable behaviour (the "start" attribute is definitely wrong), there may well be other errors in the XML that the VDM tools do not report until the first problem is resolved and the specification re-tested.

So to make the VDM model more useful, it defines types without explicit invariants, and provides a number of (nested) "isValid..." functions that do the invariant checks and report all of the errors that they encounter. This is a similar approach to that taken in [12]. To report all of the errors, the normal pattern of "check1 and check2 and check3 and ..." is converted to a style that uses a set of boolean results, "{ check1, check2, check3, ... } = { true }". This causes the evaluation to perform every check in the set, and only fail at the end, if at least one of the checks fails.

3.3 VDM Annotations

The invariant checking functions described above need to be able to capture which sub-clauses are in error, and then carry on with the evaluation. To enable this, a new annotation was developed as part of this work, called @OnFail. The annotation has to be applied to a bracketed boolean sub-clause. If (and only if) the sub-clause returns false then a string with embedded arguments is printed to the console (the arguments work like printf). The evaluation of the function overall is not affected by this. For example:

```
checkMinMax: int * int * int +> bool
checkMinMax(min, max, start) ==
{   -- @OnFail("2.2.7 min (%s) must be <= max (%s)", min, max)
    ( min <= max ),
    -- @OnFail("2.2.7 start (%s) must be >= min (%s) and <= max (%s)", start, min, max)
    ( start >= min and start <= max )
}
= {true};
```

That produces the following result, when the function is evaluated from the console:

```
> print checkMinMax(1, 10, 5)
= true

> print checkMinMax(1, 10, 123)
2.2.7 start (123) must be >= min (1) and <= max (10)
= false

> print checkMinMax(10, 1, 123)
2.2.7 min (10) must be <= max (1)
2.2.7 start (123) must be >= min (10) and <= max (1)
= false
```

Note the set-comparison pattern. This will always perform both checks in the set. If either fail, the corresponding message(s) will be written to the console, but @OnFail cannot effect the overall result (which will be false). If the tests in the set pass, the @OnFail annotation is not triggered.

The number at the start of the messages is (by convention) a link to the FMI standard section number where the corresponding rule is defined. This linkage could be strengthened in future.

3.4 The Top Level Structure - FMIModelDescription

The top level XML element in the configuration of an FMU is called FMIModelDescription. The equivalent structure in VDM-SL is as follows:

```
-- XSD definition in section 2.2.1, p25
FMIModelDescription ::
    -- The common model attributes
    attributes           : ModelAttributes            -- XSD 2.2.1, p31
    -- ModelExchange
    modelExchange        : [ModelExchange]            -- XSD 3.3.1, p89
    -- CoSimulation
    coSimulation         : [CoSimulation]             -- XSD 4.3.1, p109
    -- Unit Definitions that are utilised in "ModelVariables"
    unitDefinitions      : [seq1 of Unit]             -- XSD 2.2.2, p33
    -- A global list of type definitions that are utilised in "ModelVariables"
    typeDefinitions      : [set1 of SimpleType]       -- XSD 2.2.3, p38
    -- Log categories
    logCategories        : [seq1 of Category]         -- XSD 2.2.4, p42
    -- Default experiment
    defaultExperiment    : [DefaultExperiment]        -- XSD 2.2.5, p43
    -- Vendor annotations
    vendorAnnotations    : [seq1 of Tool]             -- XSD 2.2.6, p43
    -- The central FMU data structure defining all variables of the FMU that
    -- Are visible/accessible via the FMU API functions.
    modelVariables       : seq1 of ScalarVariable     -- XSD 2.2.7, p44
    -- Defines the structure of the model. Especially, the ordered lists of
    -- outputs, continuous-time states and initial unknowns (the unknowns
    -- during Initialisation Mode) are defined here. Furthermore, the
    -- dependency of the unknowns from the knowns can be optionally
    -- defined.
    modelStructure       : ModelStructure;            -- XSD 2.2.8, p56
```

The commented XSD section and page references link to the section in the FMI standard where the corresponding XML element types are defined. The names of the VDM-SL types are the same as the XSD types. Note that most fields are optional types (that is, the type is shown in [brackets] and may have the value "nil", indicating "no value", which is the same as the absence of an element in XML).

In addition to this type definition, a validation function is defined, which applies all of the checking rules recursively to each of the fields of an FMIModelDescription. The function (truncated for brevity) is as follows:

```
/**
 * Invariant definition for FMIModelDescription
 */
isValidModelFMIDescription: FMIModelDescription +> bool
isValidModelFMIDescription(md) ==
  -- First fill in effective values for model variables' missing attributes
  let eModelVariables =
      [ effectiveScalarVariable(sv) | sv in seq md.modelVariables ]
  in
  {
      -- @OnFail("2.2.1 ModelAttributes invalid")
      ( isValidModelAttributes(md.attributes) ),
      -- @OnFail("2.2.1 ModelExchange invalid")
      ( isValidModelExchange(md.modelExchange) ),
      -- @OnFail("2.2.1 CoSimulation invalid")
      ( isValidCoSimulation(md.coSimulation) ),
      -- @OnFail("2.2.1 UnitDefinitions invalid")
      ( isValidUnitDefinitions(md.unitDefinitions) ),
      -- etc... for all fields, then inter-field checks...

      ( let outputIndexes = { svi | svi in set inds eModelVariables &
            eModelVariables(svi).causality = <output> }
        in
           if outputIndexes <> {}
           then
               -- @OnFail("2.2.1 ModelStructure.Outputs should be %s", outputIndexes)
               ( md.modelStructure.outputs <> nil
                 and { u.index | u in seq md.modelStructure.outputs } = outputIndexes )
           else
               -- @OnFail("2.2.1 ModelStructure.Outputs should be omitted")
               ( md.modelStructure.outputs = nil )
       ),

       -- etc...
 } = {true};
```

The first part of the validation function produces a modified version of the ModelVariables. This is because variable definitions can omit fields, which then default to an "effective" value according to rules defined in the FMI standard (i.e. bottom of p. 48 of the FMI standard). Subsequent checks that depend on the modelVariables then use these effective values.

The main body of the validation function is a set of boolean checks, as discussed in section 3.2.

All of the validation functions are named `isValid<type>`, and they are defined for each type in the schema. So the first field of the FMIModelDescription structure is of type ModelAttributes (corresponding to the XML attributes of the FMIModelDescription element) and its validity function is `isValidModelAttributes`. That function may have `@OnFail` annotations, but if the attributes are not valid, this outer level will raise its own `@OnFail` to say that the attributes have problem(s).

Checks at this level also verify that values are consistent between fields, for example that all of the “output” variables in the ModelStructure exist in the ModelVariables and are of type “output”, as shown.

The full list of @OnFail messages for the FMIModelDescription level is given here, without further explanation, to give an idea of the set of tests that are performed (most of these have more detailed @OnFail messages from the validation of the field concerned):

```
"2.2.1 ModelAttribute fmiVersion should be 2.0"
"2.2.1 ModelAttributes invalid"
"2.2.1 ModelExchange invalid"
"2.2.1 CoSimulation invalid"
"2.2.1 UnitDefinitions invalid"
"2.2.1 TypeDefinitions invalid"
"2.2.1 LogCategories invalid"
"2.2.1 DefaultExperiment invalid"
"2.2.1 VendorAnnotations invalid"
"2.2.1 ScalarVariables invalid"
"2.2.1 ScalarVariables typecheck against TypeDefinitions failed"
"2.2.1 ModelStructure invalid"
"2.2.1 Neither ModelExchange nor CoSimulation specified"
"2.2.3 TypeDefinition and ScalarVariable names overlap: %s"
"2.2.3 SimpleType %s, unit must be defined for displayUnit %s"
"2.2.3 SimpleType %s, Real unit %s not defined in UnitDefinitions"
"2.2.7 ScalarVariable %s, canHandleMultipleSetPerTimeInstant invalid"
"2.2.7 ScalarVariable %s, Real unit must be defined for displayUnit %s"
"2.2.7 ScalarVariable %s, Real unit %s not defined in UnitDefinitions"
"2.2.7 ScalarVariable %s, Real reinit for model exchange continuous time only"
"2.2.8 Outputs should be %s"
"2.2.8 Outputs should be omitted"
"2.2.8 Real/derivative variables but no Derivatives declared"
"2.2.8 Derivatives section does not match Real/derivative variables"
"2.2.8 Derivatives indexes out of range"
"2.2.8 Derivatives declared, but no Real/derivative variables"
"2.2.8 InitialUnknowns must include: %s"
"2.2.8 InitialUnknowns are not sorted: %s"
"2.2.3 Typedefs have multiple matching names: %s"
"2.2.7 ScalarVariable %s, RealType not referenced by Real variable %s"
"2.2.7 ScalarVariable %s, calculated min/max does not match start %s"
"2.2.7 ScalarVariable %s, Real unit does not match declaredType"
"2.2.7 ScalarVariable %s, Real displayUnit does not match declaredType"
"2.2.7 ScalarVariable %s, IntegerType not referenced by Integer variable %s"
"2.2.7 ScalarVariable %s, calculated min/max does not match start %s"
"2.2.7 ScalarVariable %s, BooleanType not referenced by Boolean variable %s"
"2.2.7 ScalarVariable %s, StringType not referenced by String variable %s"
"2.2.7 ScalarVariable %s, EnumerationType not referenced by Enumeration variable %s"
```

3.5 Model Variables - ScalarVariable

The ModelVariables section of an FMIModelDescription is a sequence of at least one `ScalarVariable` definition. This section has to be modelled as a sequence (rather than a set) because the order of the variable declarations is used to identify them elsewhere in the configuration. These indexes start at 1, which is the same convention as VDM. The equivalent types in VDM-SL are as follows:

```
Real ::          -- XSD 2.2.7, p52
    declaredType     : [NormalizedString1]
    quantity         : [NormalizedString1]
    unit             : [NormalizedString1]
    displayUnit      : [NormalizedString1]
    relativeQuantity : [bool]
    min              : [real]
    max              : [real]
    nominal          : [real]
    unbounded        : [bool]
    start            : [real]
    derivative       : [nat1]
    reinit           : [bool];

-- ... also Integer, Boolean, String and Enumeration defined similarly

Variable = Real | Integer | Boolean | String | Enumeration;

VarName = NormalizedString1;    -- Has a syntax defined in section 2.2.9, p64

ScalarVariable ::               -- XSD 2.2.7, p44
    -- attributes
    name                               : VarName
    valueReference                     : nat
    description                        : [AnyString]
    causality                          : [Causality]
    variability                        : [Variability]
    initial                            : [Initial]
    canHandleMultipleSetPerTimeInstant : [bool]
    -- elements
    variable        : Variable
    annotations     : [seq1 of Tool];
```

The isValidScalarVariable function checks the validity of the fields, which involves checking the min/max/start values of the variables, and checking that the causality, variability, initial and variable start field values, are a valid combination according to the rules in the standard. The (truncated) function is as follows:

```
/**
 * Verify a sequence of ScalarVariables.
 */
isValidScalarVariables: seq1 of ScalarVariable +> bool
isValidScalarVariables(svs) ==
   {
     -- @OnFail("2.2.7 ScalarVariables.causality defines more than one independent variable")
     ( card { sv | sv in seq svs & sv.causality = <independent> } <= 1 ),
     -- @OnFail("2.2.7 ScalarVariable names are not unique")
     ( card { sv.name | sv in seq svs } = len svs ),
     -- @OnFail("2.2.7 Invalid ScalarVariable aliasing")
     ( -- ... check aliasing of variables with same valueReference )
   }
 union
   {
     -- @OnFail("2.2.7 ScalarVariables[%s] invalid", sv.name)
     ( isValidScalarVariable(sv) )
     | sv in seq svs
   } = {true};

/**
 * ScalarVariable invariant. Rules defined in the table on p49.
 */
isValidScalarVariable: ScalarVariable +> bool
isValidScalarVariable(sv) ==
  let eCausality    = effectiveCausality(sv.causality),
       eVariability = effectiveVariability(sv.variability),
       eInitial     = effectiveInitial(sv.initial, eCausality, eVariability)
  in
  {
      -- Table on p46 defining causality, and p48/49 defining combinations
      -- @OnFail("2.2.7 Causality/variability/initial/start %s/%s/%s/%s invalid",
                         eCausality, eVariability, eInitial, sv.variable.start)
      (
           cases eCausality:
               <parameter> ->
                    eVariability in set {<fixed>, <tunable>}
                    and eInitial = <exact>,       -- (A)
                    ...
      ),
      -- Table on p46 defining variability, and p49 defining combinations
      -- @OnFail("2.2.7 Variability/causality %s/%s invalid", eVariability, eCausality)
      (
           cases eVariability:
               <constant> ->
                    eCausality in set {<output>, <local>},
                    ...
      ),
      -- Table on p47 defining initial
      -- @OnFail("2.2.7 Initial/causality %s/%s invalid", sv.initial, eCausality)
      (
          sv.initial <> nil =>
               (eCausality not in set {<input>, <independent>})
      ),
      -- Table on p47 defining initial
      -- @OnFail("2.2.7 Initial/variability/start %s/%s/%s invalid",
                           eInitial, eVariability, sv.variable.start)
      (
          cases eInitial:
              <exact> ->
                  sv.variable.start <> nil,
                  ...
      ),
      -- @OnFail("2.2.7 Variable min/max/start invalid")
      (
          cases sv.variable:
              mk_Real(-, min, max, -, -, start, -) ->
                   isInRange[real](min, max, start),
              ...
      ),
      -- @OnFail("2.2.7 VendorAnnotations invalid")
      (isValidVendorAnnotations(sv.annotations))
  } = {true};
```

The main validation function checks that the whole sequence of variables passed defines unique names, at most of “independent” causality and that the aliased variables follow the rules. Then it calls a second function to check each `ScalarVariable` in isolation. The second function checks that for each of causality, variability and initial, the other two values are valid. For example, if the effective causality is "parameter", then the effective variability must be “fixed” or “tunable” and the initial value must be "exact"; and later, if the effective initial value is “exact”, then the variable must have a “start” value defined.

3.6 The Model Structure - ModelStructure

The ModelStructure section of an FMIModelDescription lists the outputs, derivatives and initial unknown values for the FMU. This is represented in VDM-SL as follows:

```
-- XSD p57
DependencyKind = <dependent> | <constant> | <fixed> | <tunable> | <discrete>;

Unknown ::             -- XSD p57
    index               : nat1
    dependencies        : [seq of nat1]
    dependenciesKind    : [seq of DependencyKind];

ModelStructure ::      -- XSD p56
    outputs             : [seq1 of Unknown]
    derivatives         : [seq1 of Unknown]
    initialUnknowns     : [seq1 of Unknown];
```

The isValidModelStructure function is small enough to list here in full:

```
/**
 * Validate an Unknown structure in isolation.
 */
isValidUnknown: Unknown +> bool
isValidUnknown(u) ==
  -- @OnFail("2.2.8 Unknown %s has invalid dependencies/kinds", u.index)
  ( if u.dependencies <> nil
    then u.dependenciesKind <> nil =>
    {
      -- @OnFail("2.2.8 Dependencies does not match dependenciesKind")
      (len u.dependencies = len u.dependenciesKind),
      -- @OnFail("2.2.8 Dependencies has duplicates")
      (len u.dependencies = card elems u.dependencies),
      -- @OnFail("2.2.8 Unknown cannot depend on itself")
      (u.index not in set elems u.dependencies)
    } = {true}
    else u.dependenciesKind = nil
   );

/**
 * Validation of a ModelStructure.
 */
isValidModelStructure: ModelStructure +> bool
isValidModelStructure(ms) ==
{
  -- @OnFail("2.2.8 ModelStructure has invalid unknowns")
  ( {list <> nil =>
        { isValidUnknown(u) | u in seq list } = {true}
      | list in set {ms.outputs, ms.derivatives, ms.initialUnknowns}
    } = {true}
  ),

  -- @OnFail("2.2.8 ModelStructure.InitialUnknowns are not of kind dependent or constant")
  ( ms.initialUnknowns <> nil =>
        forall iu in seq ms.initialUnknowns &
           iu.dependenciesKind <> nil =>
              forall dk in seq iu.dependenciesKind &
                 dk in set { <dependent>, <constant>, nil }
  )
} = {true};
```

The validation of the Unknown type checks that the dependencies and dependenciesKind are consistent: if dependencies is defined, then dependenciesKind need not be, but if it is then it has the same number of elements as dependencies; dependencies must not have duplicate entries; if there are no dependencies, then there must be no dependenciesKind.

The validation of the overall ModelStructure of Unknowns checks that each Unknown is valid, and that any dependencyKinds defined for the initial unknowns can only be “dependent” or “constant”.

3.7 Other Types

There are other types defined at the top level of FMIModelDescription, but their definitions and validation functions are not as complex as those covered in previous sections. Their @OnFail messages are listed below for completeness:

```
"4.3.1 CoSimulation source file names are not unique: %s"
"2.2.5 DefaultExperiment stop time must be later than start time"
"2.2.5 DefaultExperiment stepSize must be less than start-stop interval"
```

```
"2.2.4 LogCategory names are not unique: %s"
"3.3.1 ModelExchange source file names are not unique: %s"
"2.2.7 min %s is not <= max %s"
"2.2.7 start %s is not within min %s/max %s"
"2.2.7 ScalarVariables define more than one independent variable: %s"
"2.2.7 ScalarVariable names are not unique: %s"
"2.2.7 Invalid ScalarVariable aliasing"
"2.2.7 Multiple aliases of reference %s are settable: %s"
"2.2.7 Aliases of reference %s are settable and independent: %s"
"2.2.7 Too many aliases of reference %s have start set"
"2.2.7 Constant aliases of reference %s have different start values"
"2.2.7 Aliases of reference %s must all be constant or variable"
"2.2.7 Aliases of reference %s must all have same unit/displayUnits"
"2.2.7 ScalarVariables %s invalid"
"2.2.7 Causality/variability/initial/start %s/%s/%s/%s invalid"
"2.2.7 Independent variable must be Real"
"2.2.7 Variability/causality %s/%s invalid"
"2.2.7 Continuous variable must be Real"
"2.2.7 Initial/causality %s/%s invalid at %s"
"2.2.7 Initial/variability/start %s/%s/%s invalid"
"2.2.7 Variable min/max/start/nominal invalid"
"2.2.7 Real nominal must be >0.0"
"2.2.7 VendorAnnotations invalid"
"2.2.3 SimpleType %s, Real max %s not >= min %s"
"2.2.3 SimpleType %s, Integer max %s not >= min %s"
"2.2.3 SimpleType %s, EnumerationType item name/values do not form a bijection"
"2.2.3 TypeDefinitions names are not unique: %s"
"2.2.3 TypeDefinition %s invalid"
"2.2.2 UnitDefinitions names are not unique: %s"
"2.2.6 VendorAnnotations tool names are not unique: %s"
```

3.8 Automated FMU Checking in VDM

A VDM model is useful in itself, because it encodes a more precise and unambiguous description of the constraints in the FMI standard than either the XSD or informal documentation alone. But to be actively useful, the model and supporting VDM tools need to be able to analyse an FMU file directly.

To do this, the FMU package file first has to be unpacked to access the “modelDescription.xml” file within it. That XML then has to be converted into VDM-SL such that the result can be put together with the VDM-SL representation of the XSD types, and the top level `isValidFMIModelDescription` function can be called to check it.

This process can be completely automated. Unpacking an FMU package is a simple matter as the package is a ZIP format. The extracted XML file can be converted to VDM-SL with a fairly simple SAX parser (using standard Java libraries). Combining the main VDM model files with the generated file and then calling isValidFMIModelDescription is simple because the command line VDM tools accept input from multiple files, and a function to evaluate can be passed. The whole process is combined into a bash shell script called VDMCheck.sh, which either prints the result of isValidFMIModelDescription as "true" or lists the `@OnFail`’s that were raised and prints `false`. For example:

```
$ VDMCheck.sh
Usage: VDMCheck.sh [-v <VDM outfile>] <FMU or modelDescription.xml file>

$ VDMCheck.sh WaterTank_Control.fmu
true
```

```
$ VDMCheck.sh MixtureGases.fmu
2.2.7 start -1 is not within min 1/max 10000
2.2.7 Variable min/max/start/nominal invalid at line 1143
2.2.7 ScalarVariables["Medium2.fluidConstants[6].normalBoilingPoint"] invalid at
line 1143
2.2.1 ScalarVariables invalid
2.2.8 Derivatives declared, but no Real/derivative variables at line 3130
2.2.8 InitialUnknowns must include: {353, 354}
false
```

Note that several messages relate to the same area: the first two messages relate to the fields of the "normalBoilingPoint" variable identified in the third message, and the last two messages say that there are ModelStructure Derivatives but no derivative variables declared, and that the InitialUnknowns are missing two index entries.

The locations given are line numbers within the XML file. Each `@OnFail` message starts with a section reference in the FMI standard where the corresponding rule is defined. An alpha release of VDMCheck is available[5].

4 Empirical Evaluation of Static Conformance

The VDMCheck.sh tool described in Section 3.8 has been executed on all of the FMUs within the version 2.0 branch of the Modellica FMI Cross Check repository[6]. The preliminary results are as follows, though these "faults" have to be investigated to determine whether the VDM model is being too strict:

- There are 692 FMUs in this branch the repository (some are duplicated for different architectures and tools)
- 395 of them pass without any `@OnFail` messages (ie. isValidFMIModelDescription returns true)
- 118 of them have invalid InitialUnknowns (missing required unknowns, or too many unknowns)
- 79 of them have ModelStructure Derivatives indexes that do no match Real/derivative variables
- 67 of them have malformed floating point variables (eg. "1." or "4.e-10")
- 56 of them have aliased variables that do not all have the same units
- 33 of them have ScalarVariable attribute inconsistencies
- 27 have a Derivatives section but no derivative variables
- 24 have the reinit flag set for co-simulation models
- 18 have derivative variables defined but no Derivatives section
- 14 have Real units that are not declared in UnitDefinitions
- 13 of them have ModelStructure Outputs that do not include every "output" variable
- 4 of them have InitialUnknowns that are not sorted in ascending order

[5] `https://github.com/INTO-CPS-Association/FMI2-VDM-Model/releases`

[6] `https://github.com/modelica/fmi-cross-check/tree/master/fmus/2.0/`

We wish to emphasise that these results are preliminary and may be a reflection of faults in the VDM model rather than faults with the FMUs concerned. The same FMUs produce the following results with the current FMU Compliance Checker[7] (version 2.0.4):

- The same 692 FMUs were tested
- 653 of them pass without any error or warnings messages
- 27 of them have Derivatives that do not refer to a derivative variable
- 12 of them have inconsistent ScalarVariable causality/variability/initial/start settings

The most frequent discrepancy between the checking tools is regarding the population of the InitialUnknowns field of the Model Structure. The rules for this seem fairly clear (section 2.2.8, p60), but 17% of the models in the repository do not seem to follow these rules.

The next most common discrepancy is regarding the Derivatives section of Model Structure, which frequently does not agree with the ScalarVariable settings. The rules are defined in section 2.2.8, p58. There is a capability flag which indicates that Derivatives can be ignored for co-simulation, but they still apply for model exchange. Our interpretation is that, if this section is defined, the corresponding ScalarVariables must be Real and have the "derivative" flag set.

Similarly, the rules for the remaining descrepancies are stated in the FMI Standard, and yet many models do not follow them.

The issue with malformed floating point values is a simple matter of XML parsing. The examples in error violate the XML standard notation for decimal numbers by not having a digit after the decimal point[8].

4.1 Tailored XML Test Results

A set of 24 "modelDescription.xml" files have been produced in order to test each one of the `@OnFail` messages in the VDM model. The same set of XML files can be packaged into minimal FMU ZIP files and processed with the FMUChecker for comparison. The FMUChecker currently reports that only 10 of the files contain errors, so here again FMUChecker is identifying far fewer problems than the VDM model. We have yet to determine whether this is the VDM model being too strict.

5 Concluding Remarks and Future Work

This paper has demonstrated that producing a formal specification of the semi-formal FMI standard highlights a number of issues that are not sufficiently clearly described in the standard, nor checked by the FMUChecker tool. In particular this leads to misconfigured InitialUnknowns and inconsistent Derivatives declarations.

[7] `https://github.com/modelica-tools/FMUComplianceChecker/releases`

[8] `https://www.w3.org/TR/xmlschema-2/#decimal`

The work to try to determine the correct semantics, based on the FMUs in the Cross-Check repository and the behaviour of the FMUChecker tool, is ongoing. We hope that the FMI community at large will welcome the kinds of verification we are able to perform in this manner.

The current work can naturally be migrated to cover subsequent versions of the FMI standard, and the process of migrating the model may identify weaknesses in the supporting standard documentation.

We also plan to extend the scope of the work to cover the dynamic semantics of the orchestration of the co-simulation of a collection of FMUs that is defined in the FMI standard. This would then allow the behaviour of different orchestration algorithms to be explored, from a formal footing (e.g., as done in [8]), and enable verification of the modular version of the Maestro co-simulation engine [19].

Acknowledgements

We are grateful to the Poul Due Jensen Foundation, which has supported the establishment of a new Centre for Digital Twin Technology at Aarhus University, which will take forward the principles, tools and applications of the engineering of digital twins. We also acknowledge EU for funding the INTO-CPS project (grant agreement number 644047) which was the original source of funding for the INTO-CPS Application, and the Research Foundation - Flanders (Grant File Number 1S06316N). Finally, we thank the reviewers for their throughout feedback.

References

1. Bastian, J., Clauß, C., Wolf, S., Schneider, P.: Master for Co-Simulation Using FMI. In: 8th International Modelica Conference. pp. 115–120. Linköping University Electronic Press, Linköpings universitet, Dresden, Germany (Jun 2011)
2. Blochwitz, T., Otter, M., Arnold, M., Bausch, C., Clauss, C., Elmqvist, H., Junghanns, A., Mauss, J., Monteiro, M., Neidhold, T., Neumerkel, D., Olsson, H., Peetz, J.V., Wolf, S.: The Functional Mockup Interface for Tool independent Exchange of Simulation Models. In: 8th International Modelica Conference. pp. 105–114. Linköping University Electronic Press; Linköpings universitet, Dresden, Germany (Jun 2011)
3. Blockwitz, T., Otter, M., Akesson, J., Arnold, M., Clauss, C., Elmqvist, H., Friedrich, M., Junghanns, A., Mauss, J., Neumerkel, D., Olsson, H., Viel, A.: Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models. In: 9th International Modelica Conference. pp. 173–184. Linköping University Electronic Press, Munich, Germany (Nov 2012)
4. 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)
5. Fitzgerald, J., Gamble, C., Payne, R., Larsen, P.G., Basagiannis, S., Mady, A.E.D.: Collaborative Model-based Systems Engineering for Cyber-Physical Systems – a Case Study in Building Automation. In: Proc. INCOSE Intl. Symp. on Systems Engineering. Edinburgh, Scotland (July 2016)

6. 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
7. FMI: Functional Mock-up Interface for Model Exchange and Co-Simulation. Tech. rep., FMI development group (2014)
8. Gomes, C., Meyers, B., Denil, J., Thule, C., Lausdahl, K., Vangheluwe, H., De Meulenaere, P.: Semantic Adaptation for FMI Co-simulation with Hierarchical Simulators. SIMULATION 95(3), 1–29 (2018)
9. Gomes, C., Oakes, B.J., Moradi, M., Gamiz, A.T., Mendo, J.C., Dutre, S., Denil, J., Vangheluwe, H.: HintCO - Hint-Based Configuration of Co-Simulations. In: International Conference on Simulation and Modeling Methodologies, Technologies and Applications. p. accepted. Prague, Czech Republic (2019)
10. Gomes, C., Thule, C., Broman, D., Larsen, P.G., Vangheluwe, H.: Co-simulation: A Survey. ACM Computing Surveys 51(3), Article 49 (Apr 2018)
11. Gomes, C., Thule, C., Larsen, P.G., Denil, J., Vangheluwe, H.: Co-simulation of Continuous Systems: A Tutorial. Tech. Rep. arXiv:1809.08463 [cs, math], University of Antwerp (Sep 2018), `http://arxiv.org/abs/1809.08463`
12. Hasanagić, M., Tran-Jørgensen, P.W.V., Lausdahl, K., Larsen, P.G.: Formalising and Validating the Interface Description in the FMI standard. In: The 21st International Symposium on Formal Methods (FM 2016) (November 2016)
13. Jones, C.B.: Systematic Software Development Using VDM. Prentice-Hall International, Englewood Cliffs, New Jersey, second edn. (1990), iSBN 0-13-880733-7
14. Larsen, P.G., Fitzgerald, J., Woodcock, J., Fritzson, P., Brauer, J., Kleijn, C., Lecomte, T., Pfeil, M., Green, O., Basagiannis, S., Sadovykh, A.: Integrated Tool Chain for Model-based Design of Cyber-Physical Systems: The INTO-CPS Project. In: CPS Data Workshop. Vienna, Austria (April 2016)
15. Larsen, P.G., Fitzgerald, J., Woodcock, J., Lecomte, T.: Trustworthy Cyber-Physical Systems Engineering, chap. Chapter 8: Collaborative Modelling and Simulation for Cyber-Physical Systems. Chapman and Hall/CRC (September 2016), ISBN 9781498742450
16. Larsen, P.G., Fitzgerald, J., Woodcock, J., Nilsson, R., Gamble, C., Foster, S.: Towards Semantically Integrated Models and Tools for Cyber-Physical Systems Design. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation, Proc 7th Intl. Symp. Lecture Notes in Computer Science, vol. 9953, pp. 171–186. Springer International Publishing (2016)
17. Schweiger, G., Gomes, C., Engel, G., Hafner, I., Schoeggl, J.P., Posch, A., Nouidui, T.: An empirical survey on co-simulation: Promising standards, challenges and research needs. Simulation Modelling Practice and Theory 95, 148–163 (Sep 2019)
18. Thule, C., Lausdahl, K., Gomes, C., Meisl, G., Larsen, P.G.: Maestro: The INTO-CPS co-simulation framework. Simulation Modelling Practice and Theory 92, 45 – 61 (2019), `http://www.sciencedirect.com/science/article/pii/S1569190X1830193X`
19. Thule, C., Palmieri, M., Gomes, C., Lausdahl, K., Macedo, H.D., Battle, N., Larsen, P.G.: Towards Reuse of Synchronization Algorithms in Co-simulation Frameworks. In: Accepted for publication at the Co-Sim-19 workshop (September 2019)
20. Tomiyama, T., D'Amelio, V., Urbanic, J., ElMaraghy, W.: Complexity of Multi-Disciplinary Design. CIRP Annals - Manufacturing Technology 56(1), 185–188 (2007)
21. Van der Auweraer, H., Anthonis, J., De Bruyne, S., Leuridan, J.: Virtual engineering at work: The challenges for designing mechatronic products. Engineering with Computers 29(3), 389–408 (2013)

Migrating Overture to a different IDE

Peter W. V. Tran-Jørgensen and Tomas Kulik

Department of Engineering, Aarhus University, Denmark

Abstract. The popularity of Eclipse, which Overture is based on, is in rapid decline. According to RebelLabs' most recent developer productivity report, IntelliJ exceeds Eclipse by 21% as the Integrated Development Environment (IDE) of choice for Java developers. Although the popularity of IDEs varies over time, it would aid in the adoption of Overture if its core features were made available for other IDEs, as suggested by Fraser at the 16th Overture workshop. However, migrating language analysis components to a different IDE is a demanding task that (1) may require reimplementing these components to conform to the ecosystem of the targeted IDE and (2) entails writing code to expose these features to the users. To improve this situation, we report on lessons learned from migrating some of Overture's core features to a different IDE and discuss to which extent this is possible without having to reimplement everything from scratch. As part of this, we have developed Emacs packages for writing and analysing VDM specifications. These packages interact with Overture's command-line interface to create an IDE-like experience in Emacs. We estimate the efforts needed to develop these packages, and propose a way forward, based on the Language Server Protocol, that will further aid in the adoption of Overture.

Keywords: Overture · IDEs · VDM · Language Server Protocol · Emacs.

1 Introduction

According to RebelLabs' most recent developer productivity report, the popularity of Eclipse, which Overture is based on, is in rapid decline [12]. In particular, the report shows that IntelliJ is the most popular choice of Integrated Development Environment (IDE) among Java developers exceeding Eclipse by 21%. As pointed out by Fraser at the 16th Overture workshop, it would aid in the adoption of Overture if the tool's features were made available for other IDEs [7].

Implementing language components for an industrial strength programming or modelling language is a resource-demanding task. As an example, the Overture parser (as of version 2.7.0) consists of almost $9K$ Lines of Code (LoC), and the type checker and interpreter consists of $> 25K$ and $> 41K$ LoC, respectively[1]. Note that these numbers do not take the abstract syntax tree into account, which is generated using the AstCreator tool [2]. Migrating language components to a different IDE often entails reimplementing parts of the language components to

[1] These numbers are calculated using cloc and exclude comments and blank lines [3].

conform to the ecosystem of the targeted IDE (or editors that can be configured as so). In addition, one must write the front-end code necessary to expose the language features to the users. Naturally, this makes supporting multiple IDEs in this way a resource-demanding task.

To better understand the extent to which it is possible to migrate Overture's language components to a different IDE without having to reimplement them, we developed Emacs [20] packages for the Vienna Development Method (VDM) [5]. These packages use built-in and third-party Emacs packages (Section 2) to expose a basic set of Overture's features to provide an IDE-like experience in Emacs (Section 3). The packages interact directly with Overture's or VDMJ's command-line interface, hence both tools are supported. We chose Emacs, as it has several frameworks for integrating external language tools, but in principle we could have used any editor with similar extension capabilities (e.g. VSCode).

We asses the efforts needed to develop the Emacs packages using a LoC measure and discuss to which extent the migration enables full reuse of Overture's language components (Section 4). The results show that migrating the most basic features of Overture to a different IDE is possible using less than 400 LoC, which we find to be surprisingly few. The main reason for this is that Emacs offers several frameworks for integrating external language tools with minimal efforts. Similar frameworks are available for other popular editors (e.g. VSCode), and a comparable workload can therefore be expected for these. We further evaluate our work by comparing the features offered by the Emacs packages to those of the Overture IDE in order to clarify the extent of the migration.

Finally, we propose a way forward, based on the Language Server Protocol (LSP) that will significantly change the development and maintenance of VDM tools. This approach would be implemented as a single (highly performant) language server that provides VDM analysis features that modern IDEs are expected to have (e.g. code completion). Editors and IDEs can then connect to this language server and use it to create an IDE-like experience. Other language processing tools (including those maintained by the formal methods community) are starting to take a similar approach (Section 5). Going forward, we believe that an LSP-based approach will not only aid in the adoption of Overture itself, but also reduce the efforts needed to maintain the tools in the long run. We therefore hope that the findings presented in this paper can be the first step towards taking VDM tool development in this direction (Section 6).

2 Background

In this section, we introduce the technologies used to develop VDM support for Emacs. We further describe the LSP protocol, which can be used to expose language features as services to external tools such as IDEs.

2.1 Emacs

Emacs is a highly extensible, customisable, open source text editor that operates on data structures called *buffers* containing text [20]. The *point* is the current location of the cursor inside a buffer. Text is associated with text properties, which among several things, enable syntax highlighting. Each buffer is associated with a *major mode* (e.g. for a particular programming language) that determines the editing behaviour of that buffer and optional *minor modes* (e.g. for highlighting indentation levels) that also affect the behaviour of Emacs in some way.

Emacs extensions, or *packages*, are written using a dialect of Lisp, called Emacs Lisp. Packages that are generally useful may be published to package archives to make them easily accessible to other Emacs users. The most comprehensive package archive for Emacs is MELPA [21], which currently contains more than 4000 packages, including those developed as part of this work.

2.2 Enabling Emacs packages

We have used several built-in and third-party packages to develop VDM support for Emacs. In this section we describe these packages and how they have supported our development efforts.

`prog-mode` is a built-in major mode that other programming modes *derive* from in order to create syntax highlighting and setup indentation rules for a particular programming or modelling language. Examples of built-in modes that has `prog-mode` as parent mode include `java-mode`, `python-mode` and `c-mode`. Syntax highlighting for VDM is created in a similar way, namely by creating a derived `prog-mode` that defines the necessary syntax rules.

`comint-mode` is used for creating shell- or REPL-like (read–eval–print loop) buffers that interact with external processes. A typical example of using `comint-mode` is passing source code contained in a buffer to a REPL for evaluation in order to have the result printed to the REPL buffer. We have created a VDM REPL that derives from `comint-mode` in order to make Overture's command-line interface accessible from within Emacs.

`flycheck` is a modern syntax checking framework for Emacs [18] that enables creation of custom syntax checkers. Doing this is mostly a matter of defining the warning and error patterns produced by the external syntax checker, and calculating the command used for performing syntax validation. This enables `flycheck` to parse warnings and errors produced by the external syntax checker and present them to the user (for example using error or warning markers, like Overture does). We have used `flycheck` to support syntax validation of VDM through invocation of Overture's command-line interface. By default, `flycheck` will perform syntax checking when this mode is enabled, the current buffer is saved, or shortly after the last change. However, this behaviour may be redefined by the user (for example to only perform syntax checking when the buffer is saved).

`yasnippet` is a template system for Emacs [16] that enables expansion of an abbreviation into a template with "placeholders" that the user must fill out. Many programming languages have a selection of templates defined for different language constructs to improve the editing experience. As an example, an object-oriented language may define a template for classes that will expand an abbreviation such as “cls” into a class with a “placeholder” for the name of that class. We currently provide 41 templates files for a selection of VDM constructs such as classes, modules, functions and operations to make VDM editing more efficient.

`prettify-symbols-mode` is a minor mode that enables replacement of ASCII characters (e.g. lambda) with more aesthetically pleasing symbols (e.g. λ). We use `pretiffy-symbols-mode` to support VDM's mathematical syntax.

2.3 The Language Server Protocol

Several editors and IDEs enable creation of plugins in a way similar to that of Emacs. However, the development of these plugins is not standardised, which makes m editors and IDEs supporting n languages a problem of $m \cdot n$ complexity. LSP reduces the complexity of this problem to $m + n$ by proposing a common protocol for language features such as *go-to-definition*, *find-references*, *code completion* and *mouse hover tooltips* [14]. In this way, m LSP-enabled editors and IDEs can use the same n language servers to add programming language support[2]. Although we have not used LSP to develop our VDM extensions for Emacs, we discuss how this could be done in Section 4 as a way forward.

3 Emacs Packages

The Emacs extensions are implemented using a series of packages supporting the following features:

- Syntax highlighting and editing
- ASCII and mathematical-based syntax
- A notion of VDM projects for models consisting of multiple files
- On the fly syntax checking using `flycheck`
- VDM templates based on `yasnippet`
- REPL support based on `comint`
- Integration with VDMJ and Overture

The packages are open-source, published under the GNU General Public License version 3.0, and available via the references provided [26]. The landing page of the repository containing the Emacs packages includes screen recordings (GIF animations) that demonstrate some of the package features and instructions for getting started using them.

The Emacs packages have been accepted into MELPA. In consequence, the packages conform to the coding standards required by this package repository. That is, all the functionality must be fully documented at the code level and reviewed by Emacs Lisp experts as part of the submission process.

[2] A list that tracks the progress of LSP clients and servers is available at https://langserver.org/

3.1 Package overview

The relationships between the different Emacs packages and Overture are shown in Figure 1. Syntax highlighting is implemented as a major mode called `vdm-mode` that is derived from `prog-mode`. Essentially, this mode defines regular expressions needed to perform highlighting of VDM keywords, types, values and so on. Therefore, this package does not need Overture (or VDMJ) to work. To further enhance syntax highlighting, `vdm-mode` uses `prettify-symbols-mode` to replace ASCII syntax with more aesthetically pleasing symbols in order to resemble VDM's mathematical syntax to the best possible extent, essentially by defining a mapping between ASCII and Unicode characters. In this way, the user can easily switch between the ASCII-based and the mathematical syntax, by toggling `prettify-symbols-mode`. For comparison, this feature is not yet supported by the Overture IDE. We note that once a buffer, which has `prettify-symbols-mode` enabled, is saved, the corresponding file will only contain the original characters, i.e. `prettify-symbols-mode` does not affect the file content. Moreover, if you move the cursor to one of the replacement symbols (e.g. λ), it will immediately be converted back to ASCII format (e.g. `lambda`) to enable editing. One of the limitations of `prettify-symbols-mode` is that it only enables single token substitution. Therefore, it cannot handle syntax fragments such as `set of T`, which is represented as `T-set` using the mathematical syntax.

Utility functionality, common to most of the other packages, is contained in the package `vdm-mode-util`. To name a few examples, this package contains functionality to detect the current VDM dialect and collect files associated with the current VDM project. Regarding the latter, `vdm-mode` will by default only perform syntax checking of the current buffer. However, for models consisting of multiple files, `vdm-mode` uses a special file `.vdm-project` to group VDM files into projects (or multi-file models). Every time syntax checking is performed or the REPL is launched, `vdm-mode` locates the root of the project (assuming that it exists) and recursively finds all VDM files associated with the current project. These files are then passed as arguments to the underlying VDM tool. We note that the Emacs packages do not yet have a feature for importing the standard libraries (e.g. `IO` and `VDMUtil`) into projects. Instead libraries must currently be added manually to projects.

Syntax checking is implemented using a package called `flycheck-vdm`, which uses `flycheck` to define a syntax checker for VDM. In particular, `flycheck-vdm` is responsible for triggering type checking by invoking Overture, which causes warnings and errors to be highlighted in a way similar to that of Overture. This package therefore defines the necessary patterns used to parse errors and warnings produced by Overture.

VDM templates for efficient editing are implemented by the `vdm-snippets` package, which uses the `yasnippet` template system to define several templates for VDM. The implementation of `vdm-snippets` itself is simple. The primary work lies in creating the VDM templates. Currently, `vdm-snippets` ships with 41 VDM templates although more templates could easily be added, if desired.

REPL support inside Emacs is implemented by the `vdm-comint` packages, which uses `comint` to interact with Overture.

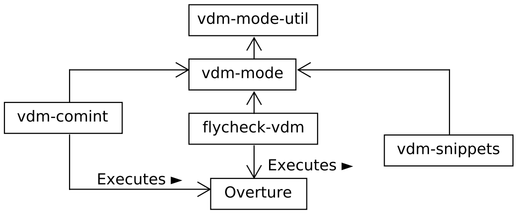


Fig. 1. Relationships between the Emacs packages and Overture.

3.2 Syntax highlighting

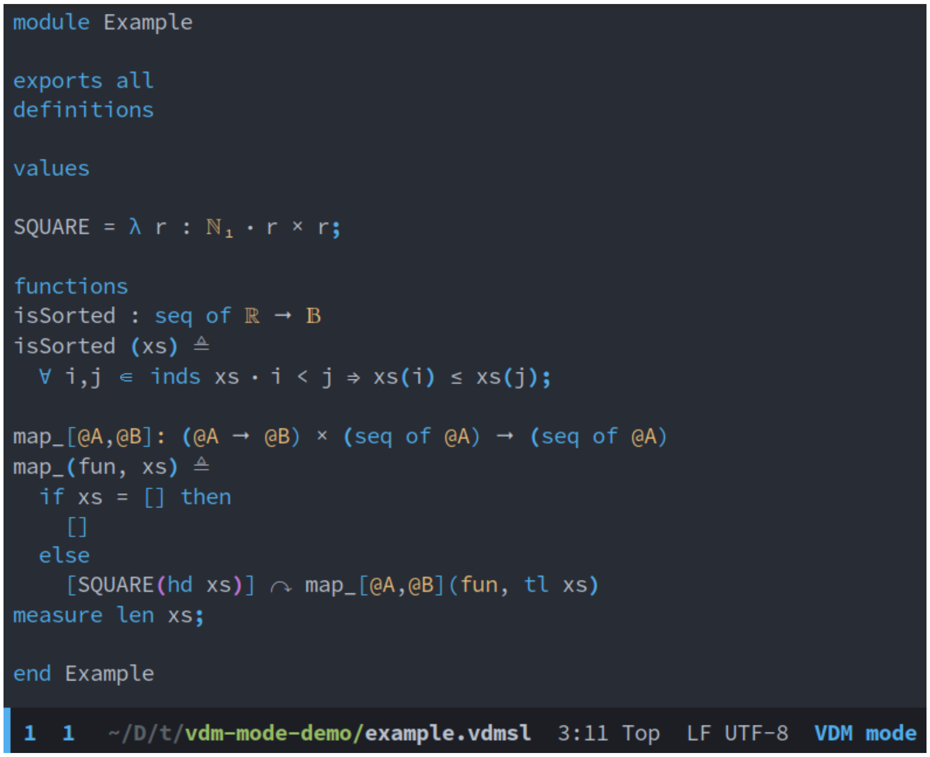
An example of what the syntax highlighting feature looks like is shown in Figure 2. Note that syntax highlighting will look slightly different on other Emacs instances that use different configurations (themes for colouring, fonts etc.). In this example, `prettify-symbols-mode` is enabled, which causes certain ASCII characters to be replaced with Unicode counterparts. To name a few examples, consider the `isSorted` function where the `forall` quantifier is replaced with the $\forall$ symbol, the `in set` operator with $\in$ and so on.

3.3 Syntax validation

The user can interact with warnings and errors produced by the syntax checker in a way similar to that of many modern IDEs. As an example, the user may have errors (or warnings) underlined and listed in the fringe (or "margin") and error descriptions printed in the echo area. An example of this is shown in Figure 3 where the error under point is caused by the mismatch between the declared return type (`bool`) and body of the function `fun`, which is type `nat1`. Alternatively, the user may choose to have `flycheck` show an interactive list of errors. In addition, there exists an extension of `flycheck` called `flycheck-pos-tip` that enables errors to be shown using popups [19], which is common in most IDEs.

3.4 The REPL

`vdm-comint` is used to interact with Overture in a REPL-like fashion. This is similar to running the command-line version of Overture in regular terminal, except that `vdm-comint` uses an Emacs buffer, hence editing and model animation is part of the same environment. This enables you to highlight parts of the model and send them directly to the REPL for evaluation. An example of a REPL session is shown in Figure 4. In this session, functions from the model shown in



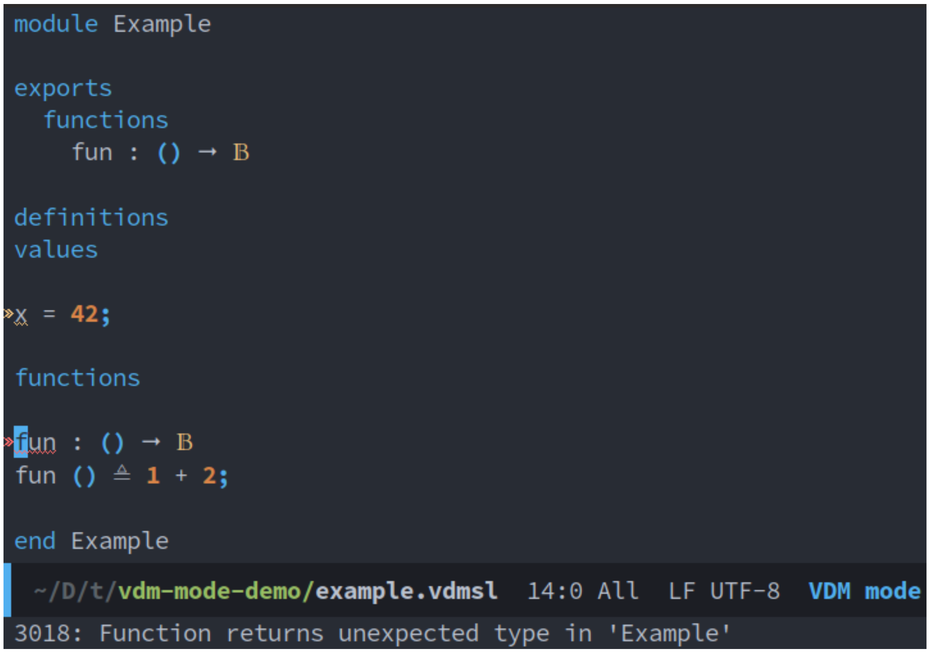
```
module Example

exports all
definitions

values

SQUARE = λ r : ℕ₁ · r × r;

functions
isSorted : seq of ℝ → 𝔹
isSorted (xs) ≜
  ∀ i,j ∈ inds xs · i < j ⇒ xs(i) ≤ xs(j);

map_[@A,@B]: (@A → @B) × (seq of @A) → (seq of @A)
map_(fun, xs) ≜
  if xs = [] then
    []
  else
    [SQUARE(hd xs)] ⁀ map_[@A,@B](fun, tl xs)
measure len xs;

end Example
 1  1  ~/D/t/vdm-mode-demo/example.vdmsl  3:11 Top  LF UTF-8  VDM mode
```

Fig. 2. Syntax highlighting using `vdm-mode`.

Figure 2 are invoked and results output to the `comint` buffer. Besides getting several Emacs features for free, `vdm-comint` has the advantage over regular shells that it can load all files associated with the current VDM project automatically, essentially by constructing the command needed to load the model.

3.5 VDM templates

`vdm-snippets` reduces the amount of editing and navigation in Emacs when writing VDM by generating a “skeleton” of VDM syntax with “placeholders” that the user must fill out. As this feature is best demonstrated using a screen recording, the reader is encouraged to go to the landing page of the repository containing the Emacs packages, to watch the GIF demonstrating this feature [26]. An example of the expansion of the VDM function template is shown in Figure 5. In this example, the user is filling out the name of the function (`myFunction`), which causes the name of the function to appear the required places. After that, the user can cycle through the remaining “placeholders” in order to fill out the type parameters (`argTypes`), the return type (`resType`), the parameters list (`argNames`) and the body of the function (which is intentionally left blank).



```
module Example

exports
  functions
    fun : () → B

definitions
values

x = 42;

functions

fun : () → B
fun () ≜ 1 + 2;

end Example
 ~/D/t/vdm-mode-demo/example.vdmsl  14:0 All  LF UTF-8  VDM mode
3018: Function returns unexpected type in 'Example'
```

Fig. 3. Syntax validation using `vdm-flycheck`.

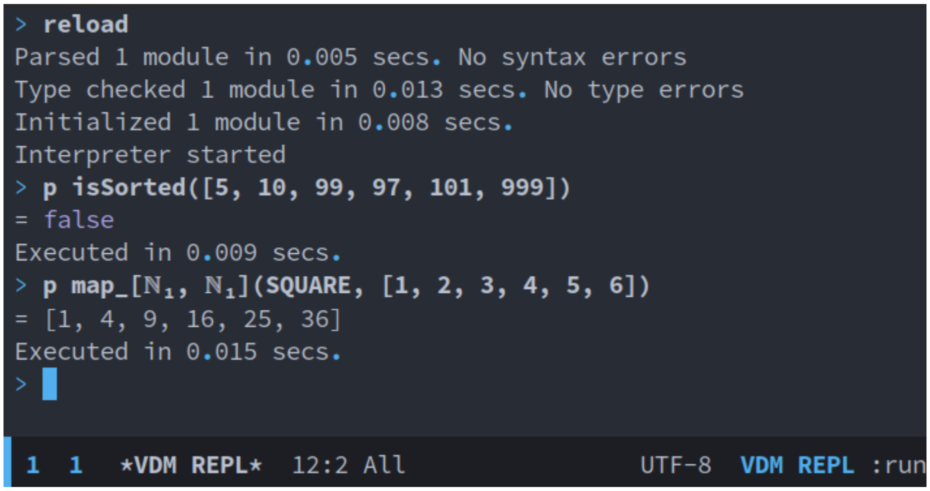
4 Evaluation

In this section we discuss to which extent it is possible to migrate Overture's language features to Emacs without having to rewrite everything from scratch. Finally, we discuss, as a way of going forward, how LSP can further help improve the uptake of Overture.

4.1 Assessing the viability of the approach

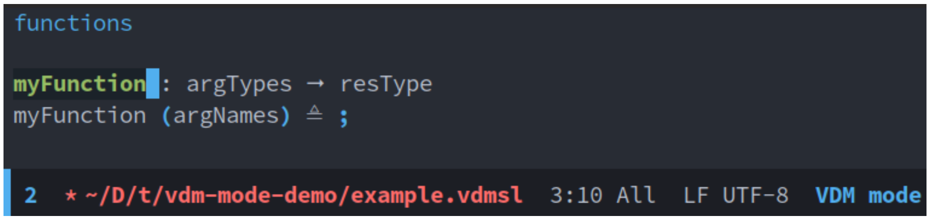
Adding support for basic VDM features in Emacs only requires 349 LoC, which we find to be surprisingly few. A more detailed overview of the lines needed to implement the individual Emacs packages are shown in Table 1. For comparison, the Overture IDE (version 2.7.0) consists of $> 119K$ LoC (not counting the core).

The few LoC needed to implement the packages can mainly be attributed to the many Emacs packages that enable integration of external language tools. As an example, creating a `flycheck` extension for VDM mostly entails defining the warning and error patterns produced by Overture and calculating the command needed to perform syntax validation. As another example, getting syntax highlighting to work is mostly a matter of creating a derived programming mode that defines regular expressions for VDM keywords, types, values and so on.



```
> reload
Parsed 1 module in 0.005 secs. No syntax errors
Type checked 1 module in 0.013 secs. No type errors
Initialized 1 module in 0.008 secs.
Interpreter started
> p isSorted([5, 10, 99, 97, 101, 999])
= false
Executed in 0.009 secs.
> p map_[ℕ₁, ℕ₁](SQUARE, [1, 2, 3, 4, 5, 6])
= [1, 4, 9, 16, 25, 36]
Executed in 0.015 secs.
> 

 1  1  *VDM REPL*  12:2 All                UTF-8  VDM REPL :run
```

Fig. 4. REPL session using `vdm-comint`.



```
functions

myFunction: argTypes → resType
myFunction (argNames) ≜ ;

 2  * ~/D/t/vdm-mode-demo/example.vdmsl  3:10 All  LF UTF-8  VDM mode
```

Fig. 5. Expansion of a template for VDM functions using `vdm-snippets`.

Table 1. LoC measures for the individual Emacs packages.

File	**Blank**	**Comment**	**Code**
`vdm-mode.el`	38	61	145
`vdm-comint.el`	30	35	102
`vdm-mode-util.el`	18	23	57
`flycheck-vdm.el`	15	32	32
`vdm-snippets.el`	14	25	13
SUM	115	176	349

The workflow Adding REPL support was achieved by wrapping the Overture interpreter in a `comint` buffer in order to make editing and model animation part of the same environment. For this reason, one may argue that the Emacs packages presented in this work support all Overture's features that are exposed via the command-line. However, since interaction is mainly achieved in a REPL-like fashion, the workflow is different from that of the Overture IDE that relies mostly on interaction through a graphical user interface. That being said, Overture does have a *quick interpreter* and the ability to launch models in a *console mode* to create a workflow similar to that of `vdm-comint`. However, the Overture IDE itself does first and foremost focus on interaction through its graphical user interface. As an example, presenting large collections of test results produced using combinatorial testing in the Overture IDE [9], is achieved using a tree-like view to improve inspection of test results.

The IDE experience One may rightfully claim that the features offered by the Emacs packages developed as part of this work do not alone comprise the feature set of a full-blown modern IDE. Although this is true, we argue that some IDE feature such as file management or project navigation are merely features of the IDE and not the language components themselves. The Emacs package repositories (e.g. MELPA) offer several packages that bring the Emacs experience closer to that of a modern IDE. Examples of such packages include

- `neotree` or `treemacs`, which are tree layout file explorers for Emacs that work in a way similar to that of Eclipse's Package Explorer,
- `projectile` for project management/navigation (e.g. switching between files),
- `counsel` or `helm`, which offer various search and completion functions for (say) simple refactoring.

The Emacs packages developed as part of this work would, however, benefit significantly from an extension that enables debugging in a way similar to that of modern IDEs. Currently, debugging is only possible using `vdm-comint`, i.e. in a command-line fashion. Although this is far from ideal, there do exist Emacs packages for integrating external debuggers [22,1]. However, it is currently unclear how much effort it will take to develop a front-end for Overture's debugger.

Besides debugging, further improvements can be made in terms of auto completion, which Overture has limited support for [10]. However, this feature is currently not exposed through Overture's command-line interface, hence not easily accessible from Emacs. However, it is still possible to get limited auto-completion support for VDM solely using Emacs code. For example using `company-mode`, which is an auto-completion framework for Emacs [17]. As an example, the default behaviour of `company-mode` will support text-based completion of constructs such as function-, operation-, type- and variable names, even across buffers. Although the calculations of the candidates themselves are not based on semantic analysis of the VDM model, `company-mode`'s default behaviour does cover the most basic needs and we do find it useful in practice. A feature by feature comparison between the Emacs packages and Overture is shown in Table 2.

Immediate improvements Improving support for VDM in Emacs beyond what is currently covered by the Emacs packages presented in this paper is expected to require significantly more work (compared to the 349 LoC contributed so far). In particular, if Overture's core features must be exposed using a custom-made graphical user interface inside Emacs.

Table 2. Feature comparison between Overture and Emacs. ✓and ÷ indicate whether a feature is supported or not, respectively. The "Emacs" column indicates the name of the package that adds support for a particular feature.

Features	**Overture**	**Emacs packages**
Syntax highlighting	✓	`vdm-mode`
Symbol prettyfication	÷	`vdm-mode`
Syntax validation	✓	`flycheck-vdm`
Evaluation	✓	`vdm-comint`
Debugging	✓	`vdm-comint`
POG	✓	`vdm-comint`
LaTeX report generation	✓	`vdm-comint`
Combinatorial testing	✓	`vdm-comint`
Code generation	✓	÷
Auto completion (limited)	✓	`company-mode` (simple, non-semantic)
Template expansion	✓	`vdm-snippets`
Standard library import	✓	÷ (must be added manually)

In addition, to the features supported so far, there are several immediate improvements that can be made to the Emacs packages. Among the more prominent features is code indentation, which `vdm-mode` does not yet define rules for. We envisage that defining indentation rules for a complex language such as VDM is a demanding task — at least if the indentation rules must cover anything beyond basic usage. Implementing a sophisticated source code formatter usually entails parsing the code in order to produce an abstract syntax tree from which formatted code is emitted. For this reason, we believe it is better to make source code formatting a service of Overture (which already includes most of the components needed to build this feature).

The Overture IDE also offers features that are not exposed through the command-line interface. Examples of such extensions include auto-completion based on semantic analysis and code-generation [8,25,27]. Supporting these features in Emacs would therefore require changing Overture in order to make them accessible to external tools and writing the code needed for the Emacs integration. If code-generation is made available via Overture's command-line interface then this feature will be available via `vdm-comint` as well. Regarding auto-completion, one could imagine the Emacs integration being implemented using a bespoke VDM backend based on `company-mode`.

4.2 The LSP-based approach

Developing and maintaining the Overture IDE is a demanding task, which is difficult to keep up with over time in an academic environment. Mainly for the reasons that changing the IDE is often time consuming and requires extensive knowledge of the Eclipse framework. Naturally, this is further complicated by the popularity of Eclipse being in rapid decline, which only makes it more difficult to increase the uptake of the tool. Migrating Overture to a more popular IDE such as IntelliJ is not a sustainable approach either, as editor popularity changes over time. For Overture development to be sustainable, the next generation of the tool should not favour any particular editor or IDE, but rather support the most popular ones using standardised ways to integrate language tools and IDEs. In this way, Overture developers can focus their efforts on language related features.

Going forward, we believe that a more viable way to increase the uptake of Overture is by exposing the tool's language features using a language server based on LSP. Since Overture is implemented using Java, one way to achieve this is using the LSP4J framework, which provides the necessary Java bindings for LSP [4]. Once Overture's language features are exposed using LSP, this language server can in principle be used to power any LSP-enabled editor of choice such as Emacs and VSCode. We believe that having the option to choose your favourite editor for VDM will play an important role in the industrial uptake of Overture.

We note that similar efforts are made elsewhere in the formal methods community to expose language features using LSP. In particular, language servers and clients are already available for other formal specification languages such as Alloy [15] and Isabelle [28], which we describe in more detail in Section 5.

5 Related Work

Support for multiple editors has become common in the field of software engineering, and similar work is now being conducted to achieve the same for formal methods tools. In this section, we describe some noteworthy attempts that aim to migrate features of formal methods tools to different development environments.

VDMTalk/VDMPad [11] is a lightweight IDE powered by VDMJ. This IDE has a feature called *LIVE tastes* that enables state manipulation, animation over modifications, visual presentation, continuous unit testing and permissive checking. VDMTalk/VDMPad is based on web technology and hence requires hosting on a web server. In comparison, our approach does not introduce a new editor, nor does it require a web server to be run. Instead our packages use Emacs in order to support a basic set of Overture's features.

Another contribution is the Overture Web IDE, which offers a web-based editor for VDM specification [13]. This IDE is a fully featured web extension of Overture that is hosted on a web server. Essentially, the Web IDE uses Overture as a back-end to provide basic editor functionality. Hence the Overture Web IDE shares many advantages and disadvantages with VDMTalk/VDMPad as the architectures of both IDEs are similar.

VDMTools [6] provides two Emacs packages, called `vdmde` and `vppde`, that are customised for VDM-SL and VDM++, respectively [24,23]. Essentially, these packages act as front-ends for the toolbox. Both packages expose a single command that, once invoked, will prompt the user for the VDM specification and load the interpreter in a new, separate window that the user can interact with in a REPL-like fashion. In this way, VDMTools' features can be accessed directly via a command-line interface that offers syntax validation, navigating between errors, debugging, code-generation and so on. Naturally, this creates an experience that is comparable to that of `vdm-comint`. In case the model has syntax errors or warnings, VDMTools' Emacs packages will highlight these in the window showing the specification. This process of having the user explicitly executing commands does, however, create a somewhat manual workflow. Although this might be beneficial for larger specifications where syntax checking is a resource-demanding task, this workflow is different from that of a modern IDE where syntax validation is usually performed incrementally. The main differences in terms of syntax validation, compared to our work, is that `flycheck-vdm` (1) is based on a modern framework (`flycheck`) that enables interacting with errors in different ways (error lists, error markers etc.) and (2) offers on-the-fly syntax checking by automatically invoking Overture (which includes collecting all files associated with the current VDM project).

Besides VDM, several other formal specification languages support multiple editors. As the current trend in enabling support for multiple editors is achieved by exposing tool features using a language server, we discuss some of these contributions in the context of formal specification.

The Alloy specification language has an LSP extension for VSCode, which enables one to run Alloy commands, viewing verification results (i.e. inspecting instances/counterexamples) using the Alloy viewer, syntax error highlighting, snippet expansion and Alloy Markdown files support [15]. Given that this extension already communicates with a language server, it should in principle be a simple task to achieve support for Alloy in other editors and IDEs.

The Isabelle formal specification language also has an LSP extension for VSCode, which supports features such as static syntax tables for Isabelle files, implicit dependency management of sources, error/warning/information message support and rich completion information for Isabelle symbols [28]. One of the notable limitations is the inability of VSCode to accept some Isabelle symbol abbreviations and lack of formal markup in prover and popup messages. These limitations are, however, attributed to the editor itself and may not be a problem for other editors. In our work we use built-in and third-party Emacs packages to implement VDM support, whereas the LSP approach taken by the authors of the Isabelle extension does not require additional packages. In particular, their approach can in principle be extended to support other editors that communicate with the same language server.

The contributions mentioned above show a trend within the FM community to move towards LSP-based implementations of formal specification languages. It is therefore expected that multi-editor support for formal methods will become more common in the future.

6 Conclusion and Future Plans

In this paper we investigate the efforts needed to migrate a basic set of Overture's core features to a different IDE and discuss to what extent this is possible without having to implement Overture's core features from scratch. As part of the investigation we developed Emacs packages for analysing VDM specifications that would leverage Overture in order to create an IDE-like environment for analysing VDM models inside Emacs. Syntax highlighting, syntax validation and template support are developed as bespoke Emacs modes by extending existing Emacs packages and modes. Features related to model animation are supported by a mode that runs the Overture interpreter in a dedicated Emacs buffer in order to make editing and model animation part of the same environment.

The efforts needed to develop the packages were estimated using a LoC measure. In total, the Emacs packages consist of 349 LoC (excluding the VDM templates) which we find to be surprisingly few. This is mainly due to the large number of modern frameworks that Emacs has for integrating external language tools. Similar frameworks are available for other editors and IDEs such as VS-Code. We therefore believe that the efforts needed to develop similar extensions for these editors will be comparable to our results.

Although the results show that adding support for basic VDM features in a modern editor or IDE can be done using a few hundreds LoC, the approach does rely on parsing the input from Overture, which is tedious and prone to errors. Going forward, we therefore believe that a better approve is to expose Overture's core features using a language server that implements LSP. In particular, we believe this approach will

- make it easier for other editors and IDEs to support VDM analysis,
- reduce the efforts needed to maintain VDM tools in the long run, and
- aid in the uptake of Overture, which could potentially power any LSP-enabled editor or IDE.

We therefore hope that the findings presented in this paper will help take VDM tools development in this direction.

References

1. Carter, M.: DBGp protocol frontend, a script debugger. https://github.com/ahungry/geben (2019)
2. Couto, L.D., Tran-Jørgensen, P.W.V., Lausdahl, K.: Principles for Reuse in Formal Language Tools. In: 31st ACM Symposium on Applied Computing (Apr 2016)
3. Danial, A.: Count Lines of Code (cloc). https://github.com/AlDanial/cloc (2019)
4. Eclipse Foundation: Eclipse LSP4J. https://github.com/eclipse/lsp4j (2019)
5. Fitzgerald, J.S., Larsen, P.G., Verhoef, M.: Vienna Development Method. Wiley Encyclopedia of Computer Science and Engineering (2008)
6. Fitzgerald, J., Larsen, P.G., Sahara, S.: VDMTools: Advances in Support for Formal Modeling in VDM. ACM Sigplan Notices **43**(2) (Feb 2008)
7. Fraser, S.: Integrating VDM-SL into the continuous delivery pipelines of cloud-based software. In: Pierce, K., Verhoef, M. (eds.) The 16th Overture Workshop. pp. 123–138. Newcastle University, School of Computing, Oxford (Jul 2018)
8. Jørgensen, P.W.V., Couto, L.D., Larsen, M.: A Code Generation Platform for VDM. In: Proceedings of the 12th Overture workshop (Jun 2014)

9. 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). https://doi.org/10.1109/SEFM.2010.32, http://dx.doi.org/10.1109/SEFM.2010.32, ISBN 978-0-7695-4153-2
10. Larsen, P.G., Lausdahl, K., Tran-Jørgensen, P.W.V., Coleman, J., Wolff, S., Couto, L.D., Bandur, V., Battle, N.: Overture VDM-10 Tool Support: User Guide. Tech. Rep. TR-2010-02, The Overture Initiative (May 2010)
11. Oda, T., Araki, K., Larsen, P.G.: VDMPad: a Lightweight IDE for Exploratory VDM-SL Specification. In: Proceedings of FormaliSE 2015. Florence (May 2015)
12. RebelLabs: RebelLabs Developer Productivity Report 2017: Why do you use the Java tools you use? https://jrebel.com/rebellabs/developer-productivity-report-2017-why-do-you-use-java-tools-you-use/ (2017)
13. 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)
14. Rodriguez-Echeverria, R., Izquierdo, J.L.C., Wimmer, M., Cabot, J.: Towards a language server protocol infrastructure for graphical modeling. In: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. pp. 370–380. MODELS '18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3239372.3239383, http://doi.acm.org/10.1145/3239372.3239383
15. Sahebolamri, A.: Alloy LSP. https://github.com/s-arash/org.alloytools.alloy (2018)
16. ao Távora, J.: YASnippet: A template system for Emacs. https://github.com/joaotavora/yasnippet (2019)
17. The Company team: A Modular in-buffer completion framework for Emacs. http://company-mode.github.io/ (2019)
18. The Flycheck team: Flycheck: Modern on-the-fly syntax checking extension for GNU Emacs. https://github.com/flycheck/flycheck (2019)
19. The Flycheck team: The `flycheck-pos-tip` package. https://github.com/flycheck/flycheck-pos-tip (2019)
20. The Free Software Foundation: GNU Emacs: An extensible, customizable, free/libre text editor — and more. https://www.gnu.org/software/emacs/ (2019)
21. The MELPA team: Milkypostman's Emacs Lisp Package Archive (MELPA). https://melpa.org (2019)
22. The RealGUD team: RealGUD: A modular front-end for interacting with external debuggers. https://github.com/realgud/realgud (2019)
23. The VDMTools team: VDMTools User Manual (VDM++). Tech. rep., Kyushu University (2016)
24. The VDMTools team: VDMTools User Manual (VDM-SL). Tech. rep., Kyushu University (2016)
25. Tran-Jørgensen, P.W.V.: Enhancing System Realisation in Formal Model Development. Ph.D. thesis, Aarhus University (Sep 2016)
26. Tran-Jørgensen, P.W.V.: Emacs packages for writing and analysing VDM specifications. https://github.com/peterwvj/vdm-mode (2019)
27. Tran-Jørgensen, P.W.V., Larsen, P.G., Leavens, G.T.: Automated translation of VDM to JML-annotated Java. International Journal on Software Tools for Technology Transfer pp. 1–25 (Feb 2017). https://doi.org/10.1007/s10009-017-0448-3, http://dx.doi.org/10.1007/s10009-017-0448-3
28. Wenzel, M.: Isabelle LSP. https://isabelle.in.tum.de/repos/isabelle/file/tip/src/Tools/VSCode (2016)

Migrating the INTO-CPS Application to the Cloud

Mikkel Bayard Rasmussen[1], Casper Thule[1], Hugo Daniel Macedo[1], Peter Gorm Larsen[1]

DIGIT, Department of Engineering, Aarhus University, Aarhus, Denmark,
`mbrbayard@live.dk`, `casper.thule@eng.au.dk`, `hdm@eng.au.dk`,
`pgl@eng.au.dk`

Abstract. The INTO-CPS Application is a common interface used to access and manipulate different model-based artefacts produced by the INTO-CPS tool chain during the model-based development of a cyber-physical system. The application was developed during the INTO-CPS project. It uses web-technologies on top of the Electron platform, and it requires a local installation and configuration on each user local machine. In this paper, we present a cloud-based version of the INTO-CPS Application which was developed while researching the potential of cloud technologies to support the INTO-CPS tool chain environment. The proposed application has the advantage that no configuration or installation on a local machine is needed. It makes full usage of the cloud resource management, and its architecture allows for a local machine version, keeping the current local approach option open.

1 Introduction

In Cyber-Physical Systems (CPSs), computing and physical processes interact closely. Their effective design, therefore, requires methods and tools that bring together the products of diverse engineering disciplines. Without such tools, it would be difficult to gain confidence in the system-level consequences of design decisions made in any one domain, and it would be challenging to manage trade-offs between them.

The INTO-CPS project created a tool chain supporting different disciplines such as software, mechatronic, and control engineering. All have notations and theories that are tailored to their needs, and it is undesirable to suppress this diversity by enforcing uniform general-purpose models [4,5,9,11,12]. The goal is to achieve a practical integration of diverse formalisms at the semantic level and to realise the benefits in integrated tool chains. In order to demonstrate that the technology works industrially, it has been applied in very different application domains (e.g., [6,11,7,3,14,15]).

Practice shows that the integration of the diverse tools in the INTO-CPS tool chain forces users to install several software packages and to configure their desktops to satisfy the multiple requirements from the different tools. Moreover, as desktop specifications vary, variations in performance are often observed, thus frustrating the user experience.

As a possible solution to alleviate the installation burden, and to obtain improved performance, the INTO-CPS Application may migrate to the cloud providing a preconfigured and elastic resources solution. This paper reports on works in progress towards such a migration.

Hosting the INTO-CPS Application in the cloud is ideal over the traditional hosting solution, as the cloud represents a network of computing units that are capable of sharing their resources dynamically. A traditional hosting solution only offers a single server with a fixed amount of resources, which would not be beneficial for the INTO-CPS Application as multiple active users carrying out co-simulations would challenge the performance of the application.

Another aspect is that the cloud is capable of redistributing its resources on-demand, resulting in dynamic upscaling and downscaling, while resources are only paid for when needed. All can be handled from the cloud providers interface and allows for cloud users to measure and see their use of cloud resources. The cloud allows the INTO-CPS Application to offer INTO-CPS as a service.

The migration of the INTO-CPS Application followed a cloud migration process targeting desktop applications [16]. This resulted in a cloud solution hosted using Amazon Web Services' EC2 service. This service offers a virtual Linux server that is capable of running the technologies that were used to implement the cloud version of the INTO-CPS Application.

The rest of this paper first provides background information in Section 2. The main part of the paper explains the INTO-CPS Application in Section 3. Finally, Section 4 and Section 5 contain a discussion of the results, the conclusions, and the envisaged future work.

2 Background

This Section briefly introduces the features, software architecture, and the base technologies involved in the INTO-CPS Application.

2.1 The INTO-CPS tool chain

The INTO-CPS Application combines the capabilities of several tools that comprise the INTO-CPS tool chain. The combination is illustrated in Figure 1 where the components that have been migrated are labelled with a small cloud symbol.

At the core is the INTO-CPS Application itself, which provides a User Interface (UI) to the tool chain, where the user is able to configure and launch co-simulations and visualise the results, perform Design Space Exploration (DSE), among other tasks.

Co-simulations are performed by the Co-simulation Orchestration Engine (COE) called Maestro, which is used by the INTO-CPS Application [17]. The INTO-CPS Application is capable of optimising a CPS design using DSE to carry out multiple co-simulations to search for optimal designs [8].

Modelio is a tool within the toolchain that supports the Systems Modeling Language (SysML) and is capable of creating a range of different diagrams describing the connectivity between the FMUs. The tool generates a configuration file (`.mm`) known as the Multi-Model (MM) describing the connections between the FMUs variables, and local instances. This can be read by the INTO-CPS Application to generate a co-simulation configuration file (`.coe`) that offers a configuration of the co-simulation, specifying how it should be performed by the COE. This file can be modified using the INTO-CPS

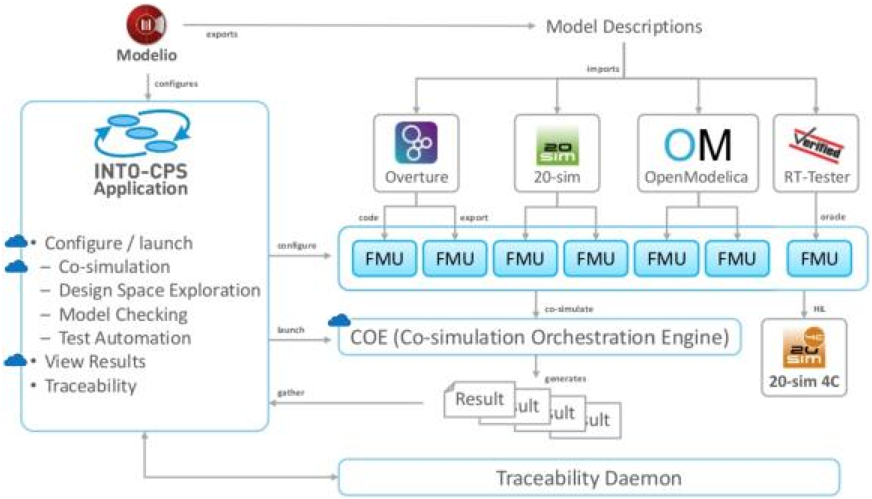


Fig. 1. Illustrates an overview of the INTO-CPS tool chain (taken from [10]). The cloud icon labels the features that were migrated during our research work.

Application, for instance specifying changes to co-simulation parameters defaults (e.g. logging, live streaming, step size,. . .).

Modelio also produces a description of a model that can be imported by the other tools within the INTO-CPS tool chain[1]. Overture, 20-sim, OpenModelica, and RT-Tester import the model description, to produce, for instance, a skeleton of the FMU definitions to be completed by the user, enabling them to interconnect using the Function Mock-up Interface (FMI) standard to generate compatible co-simulation Functional Mockup Units (FMUs) [2] to be used by the INTO-CPS Application[2].

2.2 The INTO-CPS Application

The application provides a UI to individual projects developed using the INTO-CPS tool chain. To read a project contents, the application uses a specific file structure, which is illustrated to the left in Figure 2. A project has two essential folders: the FMU folder and the Multi-model (MM) folder. The FMU folder contains the project developed FMU's files (`.fmu`), and the MM folder includes the (`.mm`) and (`.coe`) files mentioned in subsection 2.1. The cloud migration of UI and project structure poses no major difficulty.

The project files are presented it in the application left menu, as illustrated within the dotted lines in the middle of Figure 2. Using the left menu, users can navigate to different files within an INTO-CPS project, and, for instance, by clicking on a model, which is made using one of the tools of the INTO-CPS tool chain, the application launches the

[1] `https://www.modelio.org/about-modelio/features.html`

[2] Note that currently none of the modelling and simulation tools have been migrated, but it is envisaged the forthcoming HUBCAP project will establish a collaboration platform in the cloud where this could be possible.

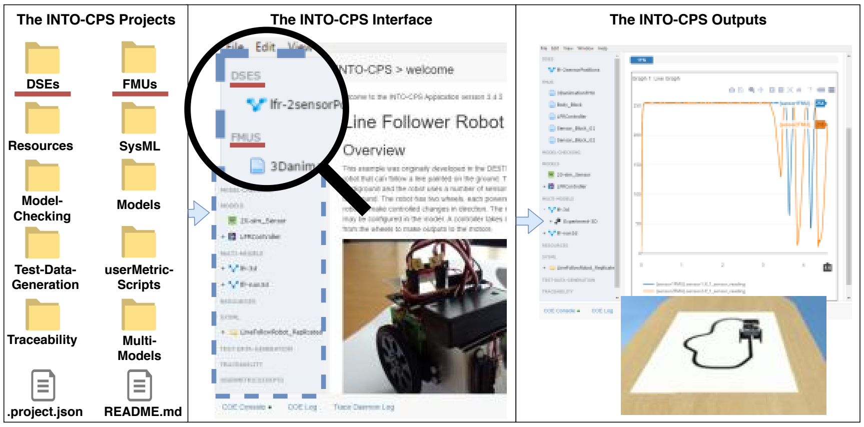


Fig. 2. Illustrates an INTO-CPS project structure and the INTO-CPS Application's representation of a project. Additionally, it illustrates the outputs that the INTO-CPS and the COE, is capable of producing.

corresponding tool with the corresponding model loaded. Although in a cloud app the first is easy implement, the later is more cumbersome.

Additionally, the INTO-CPS Application includes a download manager which supports fetchning the tools of the INTO-CPS tool chain, including the DSE and the COE, which must be downloaded to carry out simulations and DSEs. In a cloud version, a COE and DSE scripts would be expected to be available by default, and download links may be provided as well.

The INTO-CPS Application, in combination with the COE, generates a set of simulation results stored in the MM folder. These results can be presented to a user in multiple ways, as they are based on raw data files that describe the FMUs interactions over time. A typical use case is to present live streaming of the simulation using 2D plots or 3D animations. Both outputs are illustrated to the right in Figure 2, depending on the use of 20-Sim, which in this case provides the 3D capability. The 20-Sim requirement posed an additional challenge during the cloud migration research, and our work focused on 2D plots only.

Technologies of the INTO-CPS Application. The INTO-CPS Application is a complex web app, making usage of several technologies. Figure 3 provides an overview of the different technologies that power the INTO-CPS Application (in the application frame), the software development environment, and the third-party applications used by the INTO-CPS Application based on the Angular framework.

Architecture of the Desktop Version of the INTO-CPS Application. The TypeScript class architecture of the pre-existing desktop version of the INTO-CPS Application caters for all INTO-CPS project features, thus we chose to reverse engineer the architecture, which we generated from the codebase, as a top-down architecture for the INTO-CPS GUI. This is still work in progress, and our research focuses on a subset (project management plus launch and visualise co-simulation) of the desktop version

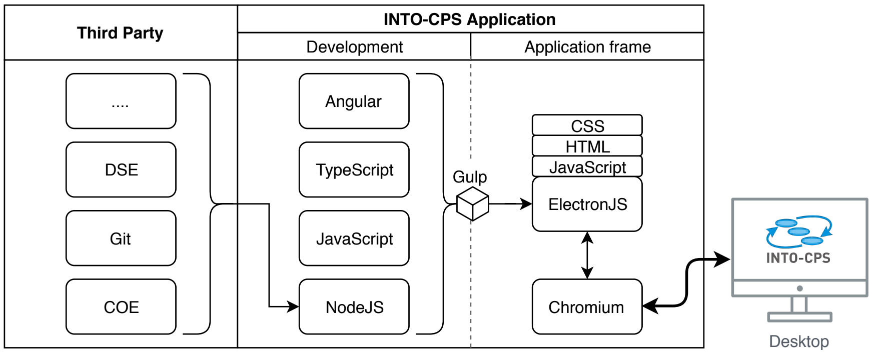


Fig. 3. The diagram illustrates the technologies used in the desktop-version of the INTO-CPS Application and their interactions.

features. Figure 4 is generated from the code base and illustrates its monolithic approach. The diagram also shows the nested relationship found within a fraction of the INTO-CPS Application.

Allowing multiple users to commonly access and use the desktop application poses a problem, due to the usage global classes and variables. Users could configure settings on the cloud, potentially affecting other users' experience, thus a cloud version requires an architectural redesign.

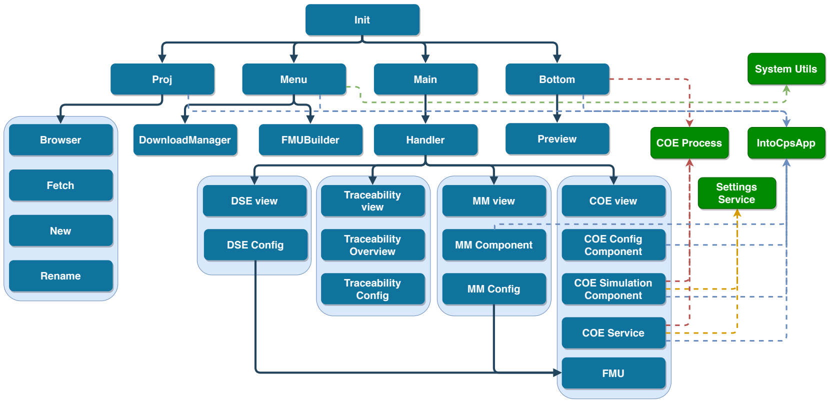


Fig. 4. Illustrates an extract of an architecture diagram of the INTO-CPS Application codebase. Note that the dotted lines illustrate relations with classes that are shared across the entire application, the coloured lines are only for distinguishing.

3 The INTO-CPS Cloud Application

The migration of the INTO-CPS Application from its desktop state to a cloud-based version was accomplished during the project reported in [16]. In the cloud version users are able to create accounts, create and manage projects, and to configure co-simulations by modifying the MM (`.mm`) and the corresponding co-simulation configuration (`.coe`) files only.

The migration of the INTO-CPS Application resulted in a new visual representation, illustrated in Figure 5, with a more modern expression that is capable of carrying out its task from all devices with a browser and an Internet connection as the more computer intensive tasks are carried out in the cloud.

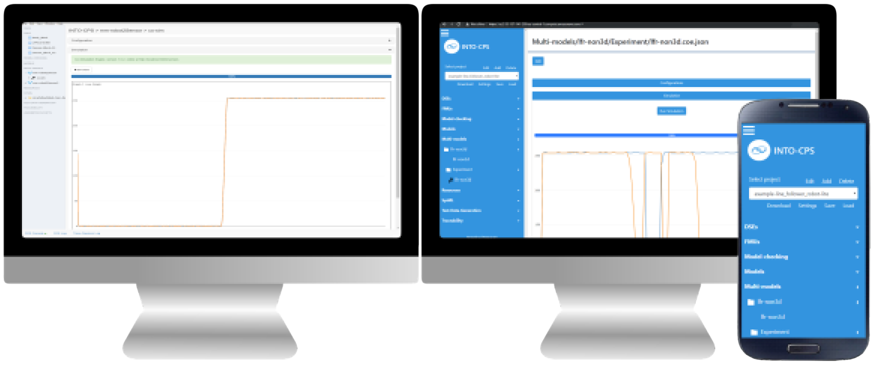


Fig. 5. Illustrates a comparison between the new and the old user interface, with the old one illustrated on the left and the new one illustrated on the right.

3.1 Adding Multi-User Support and Cloud Based Co-simulation

The cloud-based version of the INTO-CPS Application, allows users to carry out the tasks illustrated in Figure 6. This Figure illustrates that a user must signup to access projects, which can be either uploaded to the cloud or an existing project can be selected and configured. The two types of configuration can be carried out and saved, and lastly, it allows the users to start a simulation if a COE is available, and view the results as a co-simulation is executing.

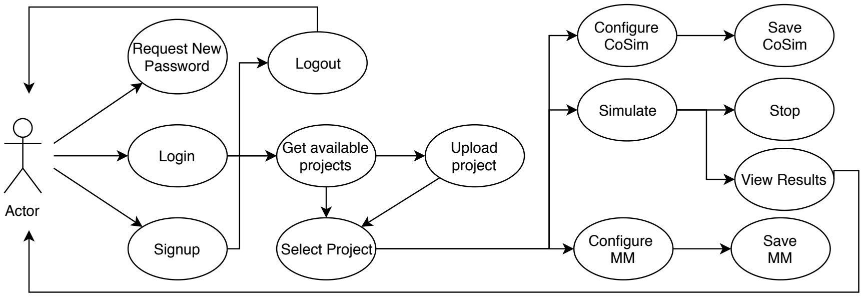


Fig. 6. Illustrates the use-case diagram of the users interacting with the cloud-based version of the INTO-CPS Application.

The desktop version of the INTO-CPS Application does not include authentication and user registration, nor does it provide the ability to upload and store files or projects in a cloud server for long term use. These functionalities were added and allow users to be recognised by their unique ID and map the users to their uploaded files.

Furthermore, the addition enabled the management of COE servers, letting a user reserve a COE for the time it takes to carry out a co-simulation. Additionally, it enables keeping track of the users that are using the cloud solution, and provides each user with an encapsulated experience, which offers users a trusted platform with a certain standard of privacy, that is capable of handling confidential data.

3.2 Technologies of the Cloud-based INTO-CPS Application

Figure 7 provides an overview of the different technologies that take part in the cloud-based INTO-CPS Application, and may compared with Figure 3, where the desktop version technologies are depicted.

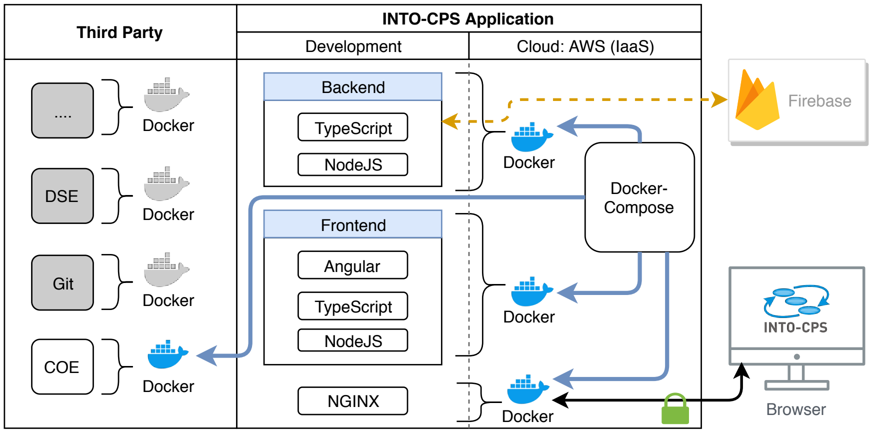


Fig. 7. Illustrates the technologies used in the cloud-based INTO-CPS Application and their interaction. Artefacts marked in grey illustrates technologies did not migrate during the first migration iteration, but are expected to be executed using Docker.

The cloud solution uses Docker[3] to easily package the different application components in the form of containers, which also easens the burden of executing the application in a suitable (libraries and dependencies equipped) environment.

The cloud solution consists of four Docker containers: the frontend app, the backend app, the COE, and the NGINX[4] web server.

- The frontend uses the Angular framework to provide the web application with the visual elements of the INTO-CPS Application, and it has a web socket connection to the backend, allowing the backend to handle more extensive tasks[5].
- The backend is a server developed using NodeJS, providing a REST API for the offering the functions and operations made available to users via the frontend.

[3] `https://docs.docker.com/get-started`

[4] `https://www.nginx.com/resources/glossary/nginx`

[5] See `https://angular.io/`.

- The COE container deploys a pool of co-simulation orchestrators.
- The NGINX web-server provides a proxy to securely interface the micro-services with the internet. Users access the cloud application using a browser to connect to the NGINX instance, which, by default, redirects the user to the frontend.

The different docker containers are managed and started by the Docker-compose tool, which orchestrates the build and launch of the Docker containers.

To authenticate users, the backend uses an API provided by Firebase, which offers a database and a preconfigured authentication module, that allows users to register for the platform and login. Firebase, therefore, stores the user credentials, and handles the user registration and access, providing each user with a unique identifier, which is used by the INTO-CPS Application to distinguish between users.

In our research experiments, all the docker components were hosted on an IaaS offered by Amazon Web Service[6] to scale resources to the necessary amount on demand.

3.3 Architecture of the Cloud Version of the INTO-CPS Application

This Section describes the details of the architectural structure of the two micro-services and the COE container that were developed for the cloud solution. The micro-services are based on the monolithic architecture, that was illustrated in Figure 4 and we embedded the desktop COE into a container.

Micro-service - Frontend The frontend of the cloud-based INTO-CPS Application must provide the different views found in the desktop version of the INTO-CPS Application. Angular is used for some of the visuals elements found on the desktop version of the INTO-CPS Application. It was therefore chosen as the frontend framework of the cloud-based solution. The Angular framework is component-based development, and the frontend components are illustrated in Figure 8.

The components and services included in the frontend are combined within the *frontend server* found at the top of Figure 8, which acts as a starting point for the micro-service. The different components carry out different tasks:

- The *management component* contains the logic associated with user management (signup,login,. . .), thereby offering an authentication unit.
- The *menu component*, contains the components that allow a user to select between uploaded projects and present their content.
- The *context component* allows the configuration files to be presented as GUI offering users a useful configuration method. Additionally, allowing users to start a co-simulation and visualise the results.
- The *communication component* allows the frontend to interact with the backend, as a service. It offers a connection to the REST API offered by the backend, and bridges the components depicted in Figure 8 and Figure 9.

[6] `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html`

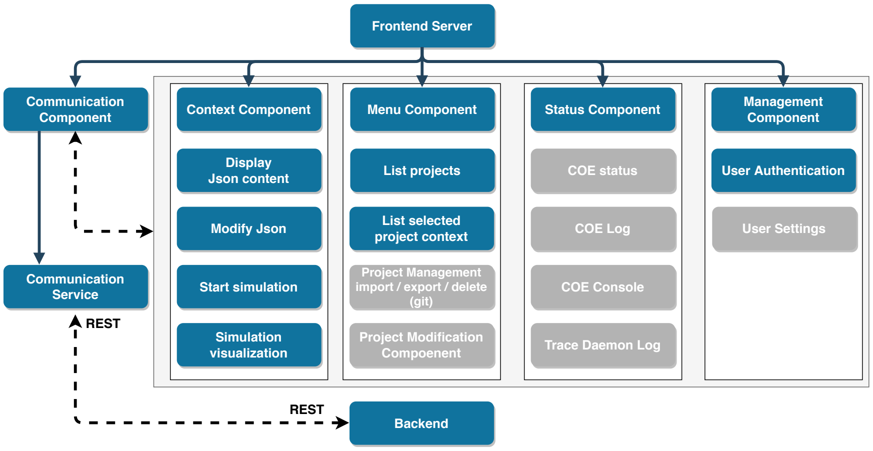


Fig. 8. Illustrates the architecture diagram of the frontend of the cloud solution, the grey boxes were excluded during the first migration iteration.

Micro-service - Backend The backend of the cloud-based INTO-CPS Application contains different handlers each connected to the server element that offers the REST API for the frontend. Figure 9 illustrates the components implemented, including the handlers, and the desktop version components yet to migrate in grey.

The backend contains three handlers: the *COE handler*, *storage handler*, and the *authentication handler*:

- The *authentication handler* manages user access, resourcing to Firebase, which is a service provided by Google, which stores the user credentials securely, and allows for an easy authentication integration.
- The *storage handler* manages the files that are uploaded by the individual users. It prevents users from accessing unauthorised files and enables the user to upload files and entire projects to the cloud.
- The *COE handler* manages the reservation of the COEs and provides the communication with each COE. The handler registers users and checks if a COE is available. If a COE is available, it adds the user and virtual IP of that COE to a ledger indicating that the COE is occupied. The handler takes care of the simulation status and data, and provides the necessary information back to the frontend. Once the backend receives a notification from the COE asserting that a simulation has ended, it removes the user and the COE from the ledger freeing the COE for other users.

Cloud challenges to the COE handler. If all COEs are busy, users are required to wait, as the co-simulation only starts if a COE is available. Our solution is not ideal, yet the cloud allows several solutions to this challenge. A cloud approach is to offer subscription based dedicated COEs, while keeping the free COEs for new users that want to try the INTO-CPS application.

If a user's COE breaks during a co-simulation, the COE remains reserved, preventing the user to reserve another one. This issue can be addressed using the feedback the backend receives from the COE to remove the user from the ledger.

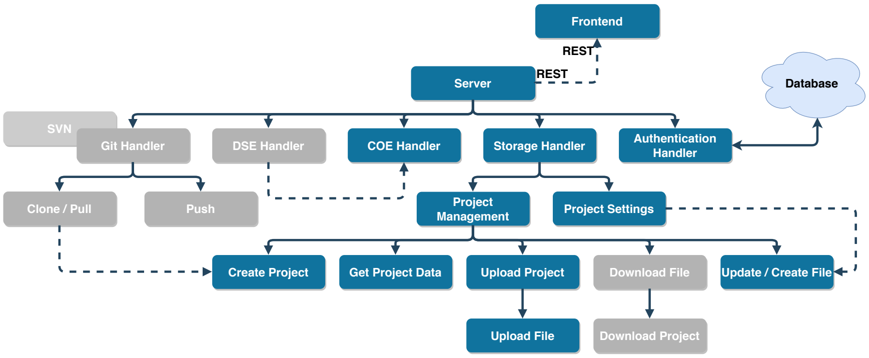


Fig. 9. Illustrates the components that exist on the backend of the cloud solution, the grey boxes were excluded in the first migration iteration.

COEs Container and Pooling The COE container wraps the COE in a suitable java runtime and a shell script is used to launch a pool of COEs for registered users to reserve and carry out co-simulations.

The pool launching is done using the features offered by the local network creation features of Docker-compose, providing each COE with a unique virtual IP, rather than a fixed IP with a different PORT number, allowing better encapsulation and security management.

During the launch of the container, Docker-compose defines a folder on the cloud environment that the backend and the pool of COEs share. This folder is used to avoid any network transactions between the two units, thereby offering the COEs and the backend a direct connection and sharing of co-simulation required files.

4 Discussion

We implement the migration during the project reported in [16] and the software development was carried out over 80 days including the migration of existing components (project management UI plus co-simulation launching and visualization UI) and we developed from scratch the many cloud specific components (authentication, encrypted web connection, shared COEs pool,. . .) absent in the desktop version.

The migration deals with the complexity of local tool configuration problems for the INTO-CPS Application, COE, and their dependencies (Java runtime, . . .). Given the elasticity of the cloud resources, the uniform user experience in terms of performance was also achieved with the migration. Nevertheless, our research developed a cloud based solution that requires a deployment in a cloud provider.

Our research results also included a study on cloud deployment options [16]. We considered at least two possibilities to migrate a desktop application to the cloud: a **Private Server** (local machine installed with cloud like services) or a **Cloud** service (provided by any of one of the major vendor).

To choose among the options we elicited several challenges associated with the migration and evaluated the pros and cons of each option. Nine common challenges were identified and worth considering before migrating any regular application to the cloud. A summary of the several options and challenges are listed in Table 1, and the advantages and disadvantages of each solution is discussed below. As a reference for comparison, we also include a column for the **Desktop** version.

Challenges	**Cloud**	**Private Server**	**Desktop**
OS development	1	1	Many
Version Control	Always the latest	Always the latest	Varies
Users	Many	Many	One
Resource scaling	Limitless	Finite	Little to none
Paying for resources	Dynamically	Fixed	None
Security management	Cloud provider	Server owner	End-user
Data storage & handling	Cloud provider	Server owner	End-user
Downtime handling	Cloud provider	Server owner	None
Server location	Many	Few	None

Table 1. Illustrates the advantages and disadvantages of the use of a cloud, local server and a desktop application solution.

Additional challenges are expected during a migration. There are several challenges in terms of using the cloud rather than the existing desktop application and furthermore in the private server server solution.

OS development: A cloud environment offers multiple candidate OSs, making it capable of hosting different versions of programming languages. A cloud solution provides versatility in the software development process allowing to target each multi-platform solution, but only requiring to develop and maintain the application in one OS. A private server share the same ideology as the cloud in this matter, where the development is only targeting one OS. A desktop application often has to work on all of Mac-OS, Windows, and Linux proving an application to as many platforms and end-users versions as possible.

Version Control: In a cloud environment or a private server solution, the end-users are always offered the latest version of the software they are using, as developers now handle the versioning and update the cloud application as soon as a new version is ready. A desktop application often requires the user to update manually, and the latest functionalities can therefore not be guaranteed on all machines.

Users: In a cloud environment or a private server solution, the code must be capable of handling multiple users and the resources they require. The cloud-based INTO-CPS Application did so by implementing a reservation system for the COE, allowing users to reserve a COE from a pool of available COEs, thereby keeping track of the resources and the users utilising them, minimising resources and cost. A desktop application does not require the same amount of control, and resources are limited to the operating machine.

Resource scaling: In a cloud environment resource scaling is effectively limitless as resources are accessed dynamically, and are capable of scaling on demand. Using a private server solution the server owner has to upgrade the hardware manually as resources are finite. On a desktop application, this can be avoided as it depends on the end-users machine.

Paying for resources: In a cloud environment, the payment depends on resource usage, meaning applications that have fewer active end-users require fewer resources and are, therefore, cheaper than an application with many active end-users. In a private server solution, the cost is fixed as all the server resources are paid for and are available for end-users to reserve. The desktop application has no such payment as the resources are dependant on the end-users machine.

Security management: In a cloud environment security is the biggest drawback due to the limited control offered, the cloud providers handle the servers and the security on a hardware level, offering little security configuration to the application owner. A private server provides full control of the entire system and application, making security better as the server owner can configure security precautions. On a desktop application, all this can be avoided as security of the system is expected to be trusted and handled by the end-user.

Data storage & handling: In a cloud, environment storage is offered by the cloud provider. However, it does provide effectively limitless scalability in terms of storage. A private server solution storage must be added as required, making developers adding more storage which need additional effort compared to a cloud solution. Desktop users have all their local storage available and have to handle upgrading themselves.

Downtime handling: In a cloud, environment downtime does happen if the cloud experiences a downtime. It means the application and the cloud server is in a state where the application or the server can not be accessed. It is recommended to research how often downtimes occur as they can cost money as no end-users are using the application hosted within the cloud. A private server solution might experience this, but the server owner has full control of the environment and is capable of preparing counter measured in this regard. A desktop user does not experience downtime as the local machine does not require an Internet connection to function.

Server location: In a cloud environment, cloud providers offer multiple locations worldwide for hosting, which can be used to make an application better for nearby areas. A private server solution can provide this. However, it requires the server owner to find the locations and configure the servers in the places manually. A desktop application is always local to the hosting machine.

An alternative approach to offer the previously existing desktop version of application as a cloud application would be to provide a cloud running virtual machine and a remote desktop access to users. This approach requires additional resources due to the overhead of the virtualised OS, as each user would require an isolated OS without a connection to other users' files for both operational and data security reasons. This becomes a challenge in terms of resources needs and costs.

5 Concluding Remarks and Future Work

The INTO-CPS Application was migrated from a locally installed desktop application to a cloud based application that users can access and interact with using a web browser. Nevertheless, the cloud-based INTO-CPS Application is still missing some functionalities compared to the desktop version, as detailed in Section 2.1. However, we claim it does provide a reasonable proof-of-concept, and this research clarified the feasibility boundaries of the migration.

The cloud-based INTO-CPS Application is only the first tool of the INTO-CPS tool chain to migrate to the cloud. It illustrates that such a migration is possible, even though only a fraction of the application was migrated, it offers users a configuration free experience and offers the developers to focus on developing for a single OS rather than multiple OSs.

Future Work. The cloud solution misses fundamental additional functionalities to be made available for users. As the cloud is billed depending on the used resources, resource measurement and a payment wall must be implemented (or contracted to a cloud provider) to ensure users are paying for their resources. Moreover, we need to develop a secure platform that encrypts data and only allows authorised users to decrypt it, as co-simulations often involve sensitive data.

The INTO-CPS Application developed a new architecture, which allows the application to offer a frontend, that can be expanded allowing new functionalities. We expect to see improvements in visualisation capabilities with the opening of new research projects on new alternatives for co-simulation configuration inside the INTO-CPS Application as the one in [13].

Another possible extension is the inclusion of FMU static checkers as the one proposed in [1], to prevent running co-simulations, and thus usage and billing of resources, that are known in advance to be bound to non-compliant FMUs.

Another advantage of the new architecture is that the cloud application was developed with the possibility for deployment both in the cloud and in a desktop. Such work is to be taken in the future, as it requires overhead for developers, but it can be handled using a builder tool such as Gulp, and it allows us to keep the previous desktop based approach as an option.

Migrating Missing Desktop Application Features. In the near future, we expect to port more features from the desktop version into the cloud, for instance porting the grey boxes in Figure 7, Figure 8, and Figure 9. In [16] a, three month time-span was proposed as a good time estimate for completion of the migration, within an ideal resource environment. In the following we provide a list of features to be added to the application.

Logging: The COE produces two logs, a status of the simulation and a status of the COE. Furthermore, the Trace Daemon also produces a log. These logging must be presented for the users when connected to either of the technologies.

Download Manager: Allowing the users to download the tools found within the INTO-CPS toolchain, this requires changes to offer the files for download using the user's browser. However, these changes are not significant because a link to a provided file initialises a download using a browser.

Single File Upload: Users are currently only capable of uploading entire projects, uploading individual files must be handled to allow users to append an FMU or other files to an existing project. This can reuse the current upload functionality. However, it does require more implementation to the frontend allowing users to upload to any position in an open project.

DSE & Traceability & Test Case Generation: All introduces new technologies, which must be discovered. Additionally, this would lead to further development on the backend and frontend and their Docker containers to get the technologies to work. Migrating each requires three weeks, used on source analysis determining the resources required, the remaining time is used for implementation and migrating.

Git: used for users to download git repositories using the tool, handling multiple users and their git credentials become an issue in a cloud, and most of this estimation regards security and handling of user authorisation to Git.

Acknowledgements We would like to thank all stakeholders that have been involved in the development of the INTO-CPS Application. We acknowledge EU for funding the INTO-CPS project (grant agreement number 644047) which was the original source of funding for the INTO-CPS Application. We are also grateful to the Poul Due Jensen Foundation, which has supported the establishment of a new Centre for Digital Twin Technology at Aarhus University, which will take forward the principles, tools and applications of the engineering of digital twins. Finally we would like to. thank Nick Battle and the anonymous reviewers for valuable feedback on earlier versions of this paper.

References

1. Battle, N., Thule, C., Gomes, C., Macedo, H.D., Larsen, P.G.: Towards Static Check of FMUs in VDM-SL. In: The 17th Overture Workshop: New Capabilities and Applications for Model-based Systems Engineering
2. Brosse, E., Quadri, I.: SysML and FMI in INTO-CPS: Integrated Tool chain for model-based design of Cyber Physical Systems p. 37 (12 2017)
3. Couto, L.D., Basagianis, S., Mady, A.E.D., Ridouane, E.H., Larsen, P.G., Hasanagic, M.: Injecting Formal Verification in FMI-based Co-Simulation of Cyber-Physical Systems. In: The 1st Workshop on Formal Co-Simulation of Cyber-Physical Systems (CoSim-CPS). Trento, Italy (September 2017)
4. 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)

5. Fitzgerald, J., Gamble, C., Payne, R., Larsen, P.G., Basagiannis, S., Mady, A.E.D.: Collaborative Model-based Systems Engineering for Cyber-Physical Systems – a Case Study in Building Automation. In: Proc. INCOSE Intl. Symp. on Systems Engineering. Edinburgh, Scotland (July 2016)
6. Fitzgerald, J., Gamble, C., Payne, R., Larsen, P.G., Basagiannis, S., Mady, A.E.D.: Collaborative model-based systems engineering for cyber-physical systems, with a building automation case study. INCOSE International Symposium 26(1), 817–832 (2016)
7. Foldager, F., Larsen, P.G., Green, O.: Development of a Driverless Lawn Mower using Co-Simulation. In: 1st Workshop on Formal Co-Simulation of Cyber-Physical Systems. Trento, Italy (September 2017)
8. Gamble, C.: Design Space Exploration in the INTO-CPS Platform: Integrated Tool chain for model-based design of Cyber Physical Systems. Aarhus University (Oct 2016)
9. Larsen, P.G., Fitzgerald, J., Woodcock, J., Fritzson, P., Brauer, J., Kleijn, C., Lecomte, T., Pfeil, M., Green, O., Basagiannis, S., Sadovykh, A.: Integrated Tool Chain for Model-based Design of Cyber-Physical Systems: The INTO-CPS Project. In: CPS Data Workshop. Vienna, Austria (April 2016)
10. Larsen, P.G., Fitzgerald, J., Woodcock, J., Gamble, C., Payne, R., Pierce, K.: Features of integrated model-based co-modelling and co-simulation technology. In: Bernardeschi, Masci, Larsen (eds.) 1st Workshop on Formal Co-Simulation of Cyber-Physical Systems. LNCS, Springer-Verlag, Trento, Italy (September 2017)
11. Larsen, P.G., Fitzgerald, J., Woodcock, J., Lecomte, T.: Trustworthy Cyber-Physical Systems Engineering, chap. Chapter 8: Collaborative Modelling and Simulation for Cyber-Physical Systems. Chapman and Hall/CRC (September 2016), iSBN 9781498742450
12. Larsen, P.G., Fitzgerald, J., Woodcock, J., Nilsson, R., Gamble, C., Foster, S.: Towards Semantically Integrated Models and Tools for Cyber-Physical Systems Design. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation, Proc 7th Intl. Symp. Lecture Notes in Computer Science, vol. 9953, pp. 171–186. Springer International Publishing (2016)
13. Legaard, C.M., Thule, C., Larsen, P.G.: Towards Graphical Configuration in the INTO-CPS Application. In: The 17th Overture Workshop: New Capabilities and Applications for Model-based Systems Engineering
14. Neghina, M., Zamrescu, C.B., Larsen, P.G., Lausdahl, K., Pierce, K.: A Discrete Event-First Approach to Collaborative Modelling of Cyber-Physical Systems. In: Fitzgerald, Tran-Jørgensen, Oda (ed.) The 15th Overture Workshop: New Capabilities and Applications for Model-based Systems Engineering. pp. 116–129. Newcastle University, Computing Science. Technical Report Series. CS-TR- 1513, Newcastle, UK (September 2017)
15. Pedersen, N., Lausdahl, K., Sanchez, E.V., Larsen, P.G., Madsen, J.: Distributed Co-Simulation of Embedded Control Software with Exhaust Gas Recirculation Water Handling System using INTO-CPS. In: Proceedings of the 7th International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH 2017). pp. 73–82. Madrid, Spain (July 2017), iSBN: 978-989-758-265-3
16. Rasmussen, M.B.: A Process for Migrating Desktop Applications to the Cloud. Master's thesis, Aarhus University, Department of Engineering (June 2019)
17. Thule, C., Lausdahl, K., Gomes, C., Meisl, G., Larsen, P.G.: Maestro: The INTO-CPS co-simulation framework. Simulation Modelling Practice and Theory 92, 45 – 61 (2019), http://www.sciencedirect.com/science/article/pii/S1569190X1830193X

Exploring Human Behaviour in Cyber-Physical Systems with Multi-modelling and Co-simulation

Ken Pierce[1], Carl Gamble[1], David Golightly[2], and Roberto Palacin[2]

[1] School of Computing, Newcastle University, United Kingdom
{kenneth.pierce,carl.gamble}@newcastle.ac.uk
[2] School of Engineering, Newcastle University, United Kingdom
{david.golightly,roberto.palacin}@newcastle.ac.uk

Abstract. Definitions of cyber-physical systems often include humans within the system boundary, however design techniques often focus on the technical aspects and ignore this important part of the system. Multi-modelling and co-simulation offer a way to bring together models from different disciplines together to capture better the behaviours of the overall system. In this paper we present some initial results of incorporating ergonomic models of human behaviours within a cyber-physical multi-model. We present two case studies, from the autonomous aircraft and railway sectors, including initial experiments and discussion of future potential.

Keywords: cyber-physical systems, ergonomics, human behaviour, multi-modelling

1 Introduction

Cyber-Physical Systems (CPSs) are systems constructed of interacting hardware and software elements, with components networked together and distributed geographically [23]. Importantly, humans are a key component of CPS design, for example, Rajkumar et al. call for "systematic analysis of the interactions between engineering structures, information processing, humans and the physical world" [34, p.734]. Humans may act as operators, acting with or in addition to software controllers controller; or as users, interpreting data from or actions of the CPS.

Model-based design techniques offer opportunities to achieve this systematic analysis. When considering the diverse nature of disciplines however, and therefore diverse modelling techniques and even vocabulary, creating models that sufficiently capture all aspects of a CPS present a challenge. Multi-modelling techniques present one solution, where models from appropriate disciplines are combined into a multi-model and are analysed, for example, through co-simulation [15, 19]. Multi-modelling has demonstrated its utility in a range of scenarios (for example, building automation [9], smart agriculture [12] and manufacturing [28]).

An open challenge in CPS design is how to accurately reflect human capabilities and behaviours. Without considering such an important part of a CPS, observed performance in an operation context may differ from that predicted by models [10, 16]. For example, train systems not achieving optimal performance due to drivers' not following eco-driving advice [33], or optimal performance requiring unrealistic demands on operators (e.g. challenging peaks or reduced wellbeing) [26].

There is a wealth of modelling human behaviours within the field of ergonomics —the study of people's efficiency in their working environment— which is discussed in Section 2. Multi-modelling would seem to offer an ideal way for these existing models to be incorporated into system-level models of CPSs. This paper reports on two initial studies where ergonomics models were incorporated into multi-models with cyber and physical components. The first looks at the effect of driving style on energy use within a urban rail system, and the second looks at operator loading in drone inspection of infrastructure. The studies focus on the use of VDM and Overture as a vehicle for initial experimentation, with references provided to papers on the ergonomic implications published within that domain.

In the remainder of the paper, Section 2 presents technical background on multi-modelling and brief survey of ergonomic modelling techniques, including related work describing their limited use in multi-modelling scenarios. Sections 3 and 4 describe two case studies in which ergonomic models were employed within a multi-modelling context, including simulation results. Finally, Section 5 concludes the paper and describes avenues for future work in this area.

2 Background

This section provides background material necessary to understand the case studies presented in Sections 3 and 4. It begins by describing the technologies used: the INTO-CPS technologies for multi-modelling, based on FMI (the Functional Mock-up Interface); VDM-RT for discrete-event modelling using the Overture tool; and the 20-sim for modelling physical phenomena. It then outlines a range of models from the ergonomics domain that could help in analysis of CPSs, and highlights some related work where ergonomic models have been used within a multi-modelling context.

2.1 Multi-modelling Technologies

The FMI standard is an emerging standard for co-simulation of multi-models, where individual models are packaged as Functional Mockup Units (FMUs). FMI defines an open standard that any tool can implement, and currently more than 30 tools can produce FMUs, with the number expected to surpass 100 soon, taking into account partial or upcoming support[3]. INTO-CPS is a tool chain based on FMI for the modelling and analysis of CPSs [19]. At the core of the tool chain is Maestro [36], an open-source and fully FMI-compliant co-simulation engine supporting variable- and fixed-step size Master algorithms across multiple platforms. Maestro includes advanced features for simulation stabilisation and hardware-in-the-loop simulation. INTO-CPS also provides a graphical front end for defining and executing co-simulations.

[3] `http://fmi-standard.org/tools/`

The Vienna Development Method (VDM) [20] is a family of formal languages based on the original VDM-SL language for systematic analysis of system specifications. The VDM-RT language allow for the specification of real-time and distributed controllers [37], including an internal computational time model. VDM-RT is an extension of the VDM++ object-oriented dialect of the family, which itself extends the base VDM-SL language. VDM is a state-based discrete-event (DE) language, suited to modelling system components where the key abstractions are state, and modifications of that state through events or decisions. Overture[4] is an open-source tool for the definition and analysis of VDM models, which supports FMU export.

The 20-sim tool[5] supports modelling and simulation of physical formula based on differential equations. 20-sim can represent phenomena from the mechanical, electrical and even hydraulic domains, using graphs of connected blocks. Blocks may contain further graphs, code or differential equations. The connections represent channels by which phenomena interact; these may represent signals (one-way) or bonds (two-way). Bonds offer a powerful, compositional and domain-independent way to model physical phenomena, as they carry both effort and flow, which map to pairs of familiar physical concepts, e.g. voltage and current. 20-sim is a continuous-time tool which solves differential equations numerically to produce high-fidelity simulations of physical components.

2.2 Ergonomics Modelling

There are a variety of modelling techniques which attempt to predict human behaviour and performance that could be used to enhance the analysis of CPS designs through inclusion in multi-models. The examples presented below highlight the potential of this area and are not intended to be an exhaustive list.

Some models suggest algebraic relationships from which predictions can be made. For example, Fitts' Law predicts that the time taken to reach a target is a ratio of the distance to and size of the target [25], while models of response time processing can predict the impact of operator delay on a real-time system [35]. Models of operator attention can predict choice of tasks and likelihood of completion [39], while combinations of workload, fatigue and task complexity can make predictions for optimum configurations of control [17].

The Yerkes-Dodson arousal model suggests that poorer performance occurs as both the lowest and highest levels of demand [6], implying that humans can be both underloaded (bored) and overloaded (stressed). This model has been applied to performance under varying levels of demand [6] and predicts an inverted U-shaped curve of performance, with poorer performance at both lower and upper bounds of demand. Empirical evidence suggests these bounds to be around 30% and 70% of occupancy [7, 30].

A range of bespoke modelling tools incorporate these types of models computationally. These include discrete-event simulation of train operator availability under given schedules [30], Monte Carlo simulations of human behaviour in Microsaint [21], and

[4] `http://overturetool.org/`

[5] `http://www.20sim.com/`

cognitive architectures such as ACT-R [2] and Soar [29] that include cognition, perception and movement performance models. Tools such as Jack [4] can also analyse complex physical models of individual human performance, while Legion supports agent-based modelling of groups of individuals in social situations such as evacuations [32].

There are limited examples of ergonomics within multi-modelling, despite the promise of bringing together work in the CPS and human factors domains. Examples include making models of human thermal comfort [1, 27] and user interactions [8] FMI compliant, and integrating agent-based models with building energy models using FMI [5, 24]. Other examples of FMI in ergonomics primarily focus on bringing the human into the loop to study nuclear reactors [38], urban mobility [3], medical devices [31], and demands on telecommunications networks [22].

While performance modelling is a mature field, a significant a gap remains between the CPS and ergonomics domains. These applications suggest a clear opportunity to improve CPS design by bringing ergonomics modelling into the CPS domain. The next sections provide two case studies which show the potential of multi-modelling as one way to achieve this.

3 Case Study 1: Operator Loading in Drone Searching

The first case study demonstrating ergonomics using multi-modelling is in the area of Unmanned Aerial Vehicle (UAV) control. The low price and capabilities of multi-rotor UAVs make them an enticing option for inspection and searching tasks [18]. UAVs fitted with high-resolution cameras are routinely used for visual inspection of infrastructure such as railway lines [11], where physically sending inspection workers carries risk of injury and fatality.

While we can envision highly-autonomous UAV systems using software flight control and image recognition to perform inspections, the current state of practice includes humans in the loop, in particular to intervene and inspect images of assets as they are relayed from UAVs. This human element means that there is an affect on system performance based on the relationship between tasks and operator availability. An existing multi-model of UAV searching [40] was enhanced with a model of operator availability to demonstrate the potential of understanding human performance within CPS design[6].

3.1 Scenario

The baseline multi-model [40] had UAVs systematically searching a wide area in a zigzag pattern, able to travel at approximately six metres per second. In this new scenario, four UAVs are tasked with visiting waypoints along a railway line, setting off from a central launch site, to perform a visual inspection by relaying images back to an operator. Each UAV has three distinct waypoints to visit with maximum distance of 1500m and mission time of approximately six minutes, Figure 1.

At each waypoint, the UAV must wait until the inspector has checked the image before moving to the next waypoint. The model represents human performance in three ways:

[6] See also Golightly et al. [13] which is aimed at an ergonomics audience.

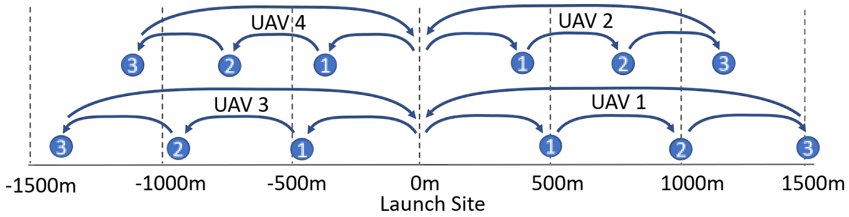


Fig. 1. Waypoints visited by each of the four UAVs

- **Task activity** The operator must realise that UAV requires attention (duration = T_{SA}), check the images (duration = T_{dec}), and interact with the UAV (duration = T_{dec}). The total time (duration = T) suggested by ergonomics models is 28 seconds [13].
- **Operator occupancy** The operator is not available to attend a UAV while occupied with another. Given the operator's capacity, a rolling window of operator occupancy was calculated over the previous 100 simulated seconds.
- **Task switching** The operator must decide which of the waiting UAVs to attend. This was achieved by always attending to the UAV that has been waited longest.
- **Dynamic performance** In addition to the above (static) calculations, a real-time model allows for feedback to affect the performance model based on the evolving conditions in the simulation. In this case, the Yerkes-Dodson arousal model was used, with time penalties (increase in T_{SA}) added for a bored operator losing track of UAV status (under 30%), and an overloaded worker dealing with too many tasks (above 70%).

There are three scenarios presented in the results section: the baseline static scenario including task activity, occupancy, and task switching); a dynamic scenario including dynamic performance; and a dynamic scenario including wind in the physical model, representing external perturbations of the system by the environment.

3.2 Multi-model

The multi-model comprises three FMUs, as shown in Figure 2. An optional 3D visualisation FMU is not shown, which connects to n instances of the 20-sim FMU. The **UAV** is realised as a continuous-time model in 20-sim. It contains a model of a quadcopter, which accepts inputs to control flight in three dimensions (throttle, pitch, roll, and yaw) and reports its position. An optional 3D visualisation FMU is not shown, which connects to these outputs to to show the UAVs visually in a 3D environment. The **Controller** is realised as a VDM model, it contains both a low-level loop controller, which moves the UAV within 3D space, and a high-level waypoint controller, which visits a sequence of waypoints using the loop controller. A single UAV instance is paired with a single Controller instance, and these pairs are replicated to realise multiple UAVs. This part of the multi-model is an updated version after Zervakis et al. [40].

To this baseline model, an **Operator** model was added to consider human performance element of the inspection system. Again VDM was used as this is the most

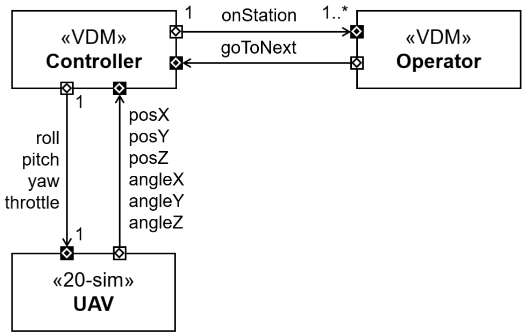
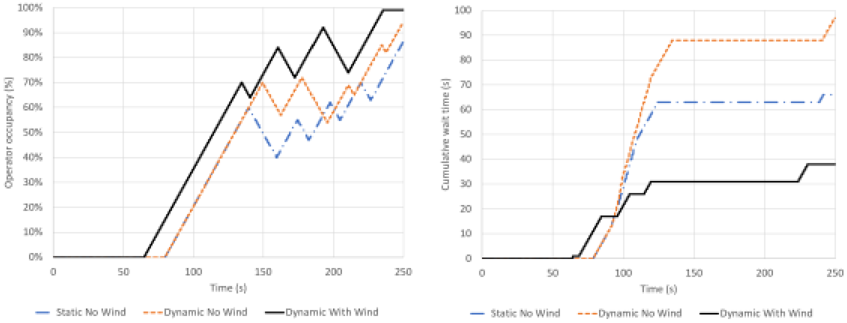


Fig. 2. The FMUs in the UAV control multi-model and their relationships

appropriate modelling language from which FMUs can be readily produce. The **Operator** is aware of each Controller instance and receives a signal when the UAV is “on station” (read for the image to be inspected). Once the inspection is complete, a return signal is sent to allow the UAV to continue to the next waypoint. The Operator encodes the Yerkes-Dodson model of occupancy described above by selecting a waiting UAV, then simulating processing of the data by occupying time before releasing the UAV to the next waypoint. When dynamic performance is switched on, the baseline time to process input is increased if the workload is below a the boredom threshold (under 30%) or above the overloaded threshold (over 70%) as seen in the listing for the `updateErgonomicTimings()` operation:

```
updateErgonomicTimings() ==
(
  if inspector_workload < work_load_n_curve_lower then
  (
    sa_time := sa_time_normal + workload_bored_sa_modifier;
  )
  else
  (
    sa_time := sa_time_normal;
  );

  if inspector_workload > work_load_n_curve_upper then
  (
    d_time := d_time_normal + workload_bored_sa_modifier;
  )
  else
  (
    d_time := d_time_normal;
  )
)
```



(a) Operator occupancy (%) over time (s)

(b) Cumulative wait time (s) over time (s)

Fig. 3. Operator occupancy and cumulative UAV waiting time during three scenarios

3.3 Results

Figure 3 shows output from the three scenarios for two metrics. The scenarios are as described above: static occupancy calculation without wind (blue, dot-dashed line), dynamic occupancy calculation without wind (orange, dashed line), and finally dynamic occupancy calculation with wind (black, solid line). The first graph, Figure 3a, shows operator occupancy as a percentage during the three scenarios. The peaks and troughs show the operator receiving and clearing inspection tasks. It can be seen that the more realistic dynamic occupancy model indicates that the operator is close to saturation by the end of the simulation. When wind is included, the altered arrival times happen to push the operator to saturation, indicating a likely reduction in wellbeing and performance.

The second graph, Figure 3b, shows the cumulative wait time of the UAVs during the simulation. This is clearly a monotonically increasing value, so it is the rate of accumulation that is indicative of system performance. Again, it can be seen that the more-realistic dynamic occupancy model shows a significantly increased wait time for the UAVs, which will impact on battery life and performance. Interestingly, when wind is introduced, the cumulative wait time is significantly reduced, indicating a more favourable arrival time due to some UAVs getting a speed boost with a tailwind, while are slowed down with a headwind. Although this indicates a favourable scenario for battery performance, it is clear that operator wellbeing is impacted. This shows the potential of multi-models to reveal hidden interactions between human, cyber and physical aspects of CPSs which otherwise might not be discovered until deployment.

4 Case Study 2: Driver Behaviour in Urban Rail

The second case study demonstrating inclusion of ergonomics models within multi-models is in the area of de-carbonisation in urban rail. De-carbonisation can be achieved principally through reducing overall energy usage, and reducing energy wastage through recovering energy from braking. While some urban rail lines are autonomous, most are still driven by human operators. Driving style has a significant effect on energy usage [33], because electrical current has a squared relationship to power, therefore accelerating and braking aggressively uses more energy. Drivers who follow defensive techniques —accelerating and braking gradually— should use less power and reduce carbon footprint, though this driving style must be traded-off against potentially increased journey times. A multi-model was developed to demonstrate the effect of driving style and more efficient rolling stock (trains) on energy usage[7].

4.1 Scenario

This study was based upon the Tyne and Wear Metro network within Newcastle upon Tyne, UK. The Metro cars weigh 40 metric tons and are powered by 1.5kV overhead lines throughout. The chosen scenario, illustrated in Figure 4 is an 800m section between two stations on the busiest part of the network, South Gosforth station and Ilford Road station. Peak throughput is 30 trains per hour. This section has also been studied previously [33], which provides baseline data to validate the co-simulation against.

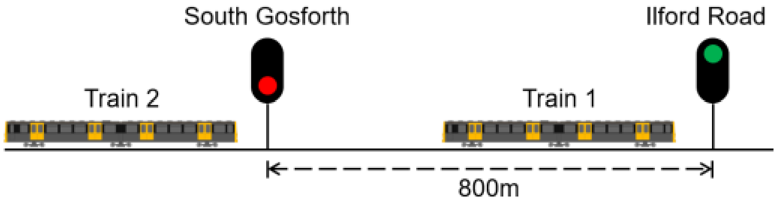


Fig. 4. The trains, stations and signals in the train control scenrio

Each station is within a track section, each section must only contain a single train at any one time. A *movement authority* controls access to each section using stop-go (two-aspect) signals. Drivers must stop at when the signal is red, and may go when the signal is green. The movement authority changes the signals when trains enter and leave sections.

In the simulated scenario, there are two trains. One train begins at Ilford Road station (Train 1), and the other at South Gosforth station (Train 2). Train 2 cannot leave South Gosforth until Train 1 departs from Ilford Road, therefore the signal for Train 2 is red. The signal for Train 1 is green, so it may leave as soon as the simulation begins. Once it has left, the signal for Train 2 goes green, so it may accelerate to leave its station, drive, then brake to stop at the next station. This gives a full accelerate, drive and brake cycle to study driver behaviour.

There are two types of drivers, aggressive drivers who use full throttle and full brake, and defensive drivers who use half throttle and half brake. There are also two types of rolling stock, the baseline Metro cars as they exist today, and a hypothetical

[7] See also Golightly et al. [14] which is aimed at an urban rail audience.

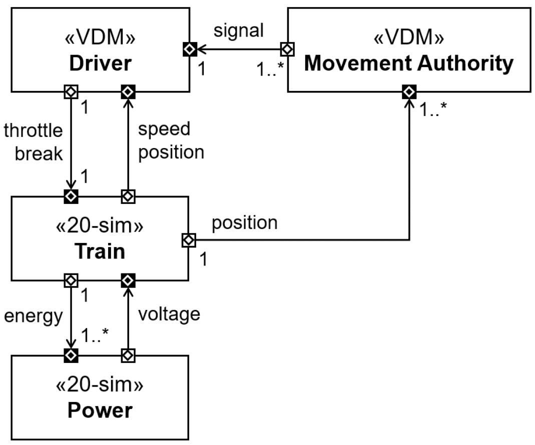


Fig. 5. The FMUs in the train control multi-model and their relationships

lighter rolling stock with 30% energy recovery from regenerative braking. This gives four scenarios: aggressive drivers with existing rolling stock, aggressive drivers with lightweight rolling stock, defensive drivers with existing rolling stock, and defensive drivers with lightweight rolling stock.

4.2 Multi-model

The multi-model comprises four FMUs, as shown in Figure 5. The **Train** is realised as a continuous-time model in 20-sim. It accepts a throttle and brake signal and computes the position, speed and energy usage of the train. Two different FMUs were produced for the baseline and lightweight rolling stock, which were swapped between scenarios. A **Power** model was also realised in 20-sim, which provides a voltage to each train and receives back the energy consumed, calculating the overall power usage of the scenario. In this experiment the model is ideal and does not voltage drop or line losses.

The **Movement Authority** is informed of the positions of all trains and updates the state of the stop-go signals as trains enter and leave track sections. These enter and leave events mean that VDM was the most appropriate modelling language for this component. Each Train instance is paired with a **Driver** instance, which observes the speed and position of their train and the state of the next signal ahead on the track. The throttle and brake in Metro cars are notched, and can only take three values (off, half, and full), therefore VDM is again an appropriate choice. The aggressive and defensive styles of driving are given as a parameter to the Driver model, with the listing below showing what happens when `throttle()` operation is applied:

```
private throttle: () ==> ()
throttle() ==
(
  if mode = <AGGRESSIVE> then
  (
    throttleActuator.setValue(AGGRESSIVE_THROTTLE);
    brakeActuator.setValue(0)
  )
  elseif mode = <DEFENSIVE> then
  (
    throttleActuator.setValue(DEFENSIVE_THROTTLE);
    brakeActuator.setValue(0)
  )
)
```

4.3 Results

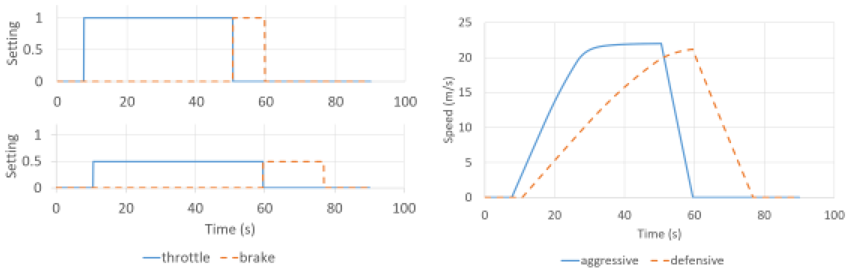
Figure 6 shows the driver outputs for Train 2 as it moves from being stationary at South Gosforth to being stationary at Ilford Road. Figure 6a shows that the aggressive driver uses full power to reach maximum speed quickly and full braking to stop quickly, while the defensive driver uses half power and brake. Figure 6b shows the speed profiles of the aggressive and defensive driver. Note that the defensive driver reaches the station later, so lower energy usage must be traded-off against journey times and timetables updated accordingly.

Figure 7 shows the total energy usage of the two trains in the four scenarios described above. The existing rolling stock with aggressive driving (dashed line) uses the most energy, while defensive driving with lightweight rolling stock (dotted line) uses the least. The downward curve here represents energy being recovered from the brakes and returning to the power system. Finally, using defensive driving with existing rolling stock or simply acquiring lightweight rolling stock provide a similar reduction in energy usage. This suggests a potential trade-off between retraining drivers and enforcing defensive driving versus investing in new stock and accept current driver performance.

5 Conclusions and Future Work

This paper argued that CPSs necessarily include humans within the system boundary, as operators or users of parts of the system. Until such time as we achieve full autonomy within these systems, it is necessary to take into account the needs and performance characteristics of humans when designing CPSs. The paper argues further that multi-modelling —combining domain models from across disciplines into system models— offers a promising avenue for exploring human performance within analysis of CPSs, supported by a brief survey of existing modelling approaches that could be incorporated, and promising related work in a similar vein.

The paper then described two initial studies in which human performance characteristics were included within multi-models. The first study considered a human operator



(a) Driver output for aggressive (top) and defensive (bottom) driving

(b) Speed profiles for for aggressive (solid) and defensive (dashed) driving

Fig. 6. Driver outputs and corresponding speed profiles for aggressive and defensive driving

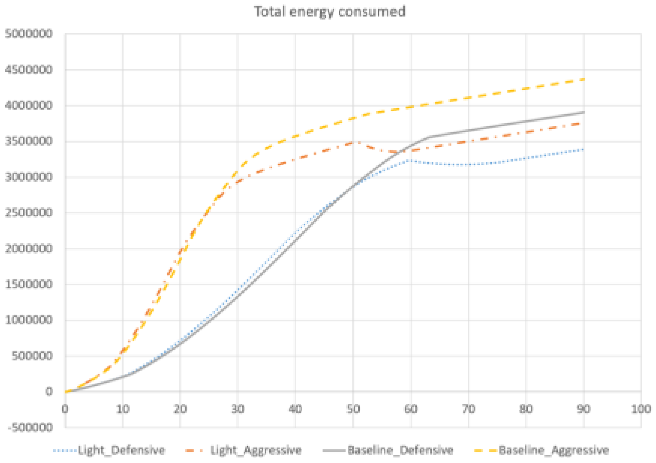


Fig. 7. Total energy usage for combinations of driving style and rolling stock

responding to data from UAVs performing inspections of a railway line. The human performance aspects included the time taken for operators to process images and switch tasks, and included performance penalties for boredom and overloading. The second study considered the effect of driver behaviour on energy usage in an urban rail system. The human performance aspects here considered aggressive drivers using maximum throttle and brake versus defensive drivers using half throttle and break. In both case studies, inclusion of human performance in the multi-model enabled trade-offs between human aspects (e.g. workload, wellbeing) and physical aspects (e.g. energy usage, system performance).

In terms of concrete next steps, both studies can be expanded to validate the results against observed data from experiments and improve the human performance models to incorporate state-of-the-art ergonomics models. Both studies were built using VDM-RT and generated as FMUs using Overture, based on appropriate performance calculations form ergonomics literature. While this was an appropriate choice for experimentation, a clear avenue for future work is to add FMI functionality to specific ergonomics modelling tools. Alternatively, dedicated FMUs could be produced to allow others to incorporate human performance factors without being ergonomics experts.

Applying the techniques to other domains should also yield interesting insights. We envision incorporating physiological models (such as Jack) to multi-models of smart manufacturing [28] to account for physical effort and movement time while interacting with an assembly line, for example. Finally, incorporating human-in-the-loop simulation, potentially using virtual reality, will allow us to refine the parameters used in the multi-models and potentially serve as training tools for operators.

Acknowledgements

The work reported here is supported in part by the Rail Safety and Standard Board (RSSB) project “Digital Environment for Collaborative Intelligent De-carbonisation” (DECIDe, reference number COF-IPS-06).

References

1. Alfredson, J., Johansson, B., Gonzaga Trabasso, L., Schminder, J., Granlund, R., Gårdhagen, R.: Design of a distributed human factors laboratory for future airsystems. In: Proceedings of the ICAS Congress. International Council of the Aeronautical Sciences (2018)
2. Anderson, J., Bothell, D., Byrne, M., Douglass, S., Lebiere, C., Qin, Y.: An integrated theory of the mind. Psychology Review 111(4), 1036–1060 (2004)
3. Beckmann-Dobrev, B., Kind, S., Stark, R.: Hybrid simulators for product service-systems: Innovation potential demonstrated on urban bike mobility. Procedia CIRP 36, 78–82 (2015), 25th CIRP Design Conference on Innovative Product Creation
4. Blanchonette, P.: Jack human modelling tool: a review. Tech. Rep. DSTO-TR-2364, Defence Science and Technology Organisation (Australia) Air Operations Division (2010)
5. Chapman, J., Siebers, P.O., Robinson, D.: On the multi-agent stochastic simulation of occupants in buildings. Journal of Building Performance Simulation 11(5), 604–621 (2018)
6. Cummings, M., E. Nehme, C.: Modeling the impact of workload in network centric supervisory control settings. In: Steinberg, R., Kornguth, S., Matthews, M.D. (eds.) Neurocognitive and Physiological Factors During High-Tempo Operations, chap. 3, pp. 23–40. Taylor & Francis (2010)
7. Cummings, M., Guerlain, S.: Developing operator capacity estimates for supervisory control of autonomous vehicles. Human Factors 49(1) (2007)
8. Filippi, S., Barattin, D.: In-depth analysis of non-deterministic aspects of human-machine interaction and update of dedicated functional mock-ups. In: Marcus, A. (ed.) Design, User Experience, and Usability. Theories, Methods, and Tools for Designing the User Experience. pp. 185–196. Springer (2014)
9. Fitzgerald, J., Gamble, C., Payne, R., Larsen, P.G., Basagiannis, S., Mady, A.E.D.: Collaborative Model-based Systems Engineering for Cyber-Physical Systems – a Case Study in Building Automation. In: Proc. INCOSE Intl. Symp. on Systems Engineering. Edinburgh, Scotland (July 2016)

10. Flach, J.M.: Complexity: learning to muddle through. Cognition, Technology & Work 14(3), 187–197 (Sep 2012)
11. Flammini, F., Pragliola, C., Smarra, G.: Railway infrastructure monitoring by drones. In: 2016 International Conference on Electrical Systems for Aircraft, Railway, Ship Propulsion and Road Vehicles International Transportation Electrification Conference (ESARS-ITEC). pp. 1–6 (2016)
12. Foldager, F., Larsen, P.G., Green, O.: Development of a Driverless Lawn Mower using Co-Simulation. In: 1st Workshop on Formal Co-Simulation of Cyber-Physical Systems. Trento, Italy (September 2017)
13. Golightly, D., Gamble, C., Palacín, R., Pierce, K.: Applying ergonomics within the multi-modelling paradigm with an example from multiple UAV control *(to appear)*. Ergonomics (2019)
14. Golightly, D., Gamble, C., Palacín, R., Pierce, K.: Multi-modelling for decarbonisation in urban rail systems *(submitted)*. Urban Rail Transit (2019)
15. Gomes, C., Thule, C., Broman, D., Larsen, P.G., Vangheluwe, H.: Co-simulation: a Survey. ACM Comput. Surv. 51(3), 49:1–49:33 (May 2018)
16. Hollnagel, E., Woods, D.D.: Joint Cognitive Systems: Foundations of Cognitive Systems Engineering. CRC Press, 1 edn. (2005)
17. Humann, J., Spero, E.: Modeling and simulation of multi-UAV, multi-operator surveillance systems. In: Annual IEEE International Systems Conference (SysCon 2018). pp. 1–8 (2018)
18. Kingston, D., Rasmussen, S., Humphrey, L.: Automated uav tasks for search and surveillance. In: 2016 IEEE Conference on Control Applications (CCA). pp. 1–8 (2016)
19. Larsen, P.G., Fitzgerald, J., Woodcock, J., Gamble, C., Payne, R., Pierce, K.: Features of integrated model-based co-modelling and co-simulation technology. In: Bernardeschi, Masci, Larsen (eds.) 1st Workshop on Formal Co-Simulation of Cyber-Physical Systems. LNCS, Springer-Verlag, Trento, Italy (September 2017)
20. 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)
21. Laughery Jr., K.R., Lebiere, C., Archer, S.: Modeling Human Performance in Complex Systems, chap. 36, pp. 965–996. John Wiley & Sons, Ltd (2006)
22. Leclerc, T., Siebert, J., Chevrier, V., Ciarletta, L., Festor, O.: Multi-modeling and co-simulation-based mobile ubiquitous protocols and services development and assessment. In: Sénac, P., Ott, M., Seneviratne, A. (eds.) Mobile and Ubiquitous Systems: Computing, Networking, and Services. pp. 273–284. Springer (2012)
23. 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`
24. Li, C., Mahadevan, S., Ling, Y., Choze, S., Wang, L.: Dynamic bayesian network for aircraft wing halth monitoring digital twin. AIAA Journal 55(3), 930–941 (2017)
25. MacKenzie, I.S.: Fitts' law as a research and design tool in human-computer interaction. Human-Computer Interaction 7(1), 91–139 (1992)
26. de Mattos, D.L., Neto, R.A., Merino, E.A.D., Forcellini, F.A.: Simulating the influence of physical overload on assembly line performance: A case study in an automotive electrical component plant. Applied Ergonomics 79, 107–121 (2019)
27. Metzmacher, H., Wölki, D., Schmidt, C., Frisch, J., van Treeck, C.A.: Real-time assessment of human thermal comfort using image recognition in conjunction with a detailed numerical human model. In: 15th International Building Simulation Conference. pp. 691–700 (2017)
28. Neghina, M., Zamrescu, C.B., Larsen, P.G., Lausdahl, K., Pierce, K.: Multi-Paradigm Discrete-Event Modelling and Co-simulation of Cyber-Physical Systems. Studies in Informatics and Control 27(1), 33–42 (March 2018)

29. Newell, A.: Unified Theories of Cognition. Harvard University Press, Cambridge, MA (1990)
30. Nneji, V.C., Cummings, M.L., Stimpson, A.J.: Predicting locomotive crew performance in rail operations with human and automation assistance. IEEE Transactions on Human-Machine Systems 49(3), 250–258 (2019)
31. Palmieri, M., Bernardeschi, C., Masci, P.: A flexible framework for FMI-based co-simulation of human-centred cyber-physical systems. In: Mazzara, M., Ober, I., Salaün, G. (eds.) Software Technologies: Applications and Foundations. pp. 21–33. Springer (2018)
32. Pelechano, N., Malkawi, A.: Evacuation simulation models: Challenges in modeling high rise building evacuation with cellular automata approaches. Automation in Construction 17(4), 377 – 385 (2008)
33. Powell, J., Palacín, R.: A comparison of modelled and real-life driving profiles for the simulation of railway vehicle operation. Transportation Planning and Technology 38(1), 78–93 (2015)
34. Rajkumar, R., Lee, I., Sha, L., Stankovic, J.: Cyber-physical systems: The next computing revolution. In: Proceedings of the Design Automation Conference, Ahaheim, California 2010. ACM (2010)
35. Teal, S.L., Rudnicky, A.I.: A performance model of system delay and user strategy selection. In: SIGCHI Conference on Human Factors in Computing Systems. pp. 295–305. ACM (1992)
36. Thule, C., Lausdahl, K., Gomes, C., Meisl, G., Larsen, P.G.: Maestro: The into-cps co-simulation framework. Simulation Modelling Practice and Theory 92, 45 – 61 (2019), `http://www.sciencedirect.com/science/article/pii/S1569190X1830193X`
37. Verhoef, M., Larsen, P.G.: Enhancing VDM++ for Modeling Distributed Embedded Real-time Systems. Tech. Rep. (to appear), Radboud University Nijmegen (March 2006), a preliminary version of this report is available on-line at `http://www.cs.ru.nl/~marcelv/vdm/`
38. Vilim, R., Thomas, K.: Operator support technologies for fault tolerance and resilience. In: Advanced Sensors and Instrumentation Newsletter, pp. 1–4 (2016)
39. Wickens, C.D., Gutzwiller, R.S., Vieane, A., Clegg, B.A., Sebok, A., Janes, J.: Time sharing between robotics and process control: Validating a model of attention switching. Human Factors 58(2), 322–343 (2016)
40. Zervakis, G., Pierce, K., Gamble, C.: Multi-modelling of Cooperative Swarms. In: Pierce, K., Verhoef, M. (eds.) The 16th Overture Workshop. pp. 57–70. Newcastle University, School of Computing, Oxford (July 2018), TR-1524

ViennaDoc: An Animatable and Testable Specification Documentation Tool

Tomohiro Oda[1], Keijiro Araki[2], Yasuhiro Yamamoto[3], Kumiyo Nakakoji[1,3], Hiroshi Sako[4], and Han-Myung Chang[5] Peter Gorm Larsen[6]

[1] Software Research Associates, Inc. (tomohiro@sra.co.jp)
[2] National Institute of Technology, Kumamoto College (araki@kyudai.jp)
[3] Future University Hakodate (yxy@acm.org, kumiyo@acm.org)
[4] Designer's Den (sakoh@ba2.so-net.ne.jp)
[5] Nanzan University (chang@nanzan-u.ac.jp)
[6] Aarhus University, DIGIT, Department of Engineering, (pgl@eng.au.dk)

Abstract. An obstacle to apply formal specification techniques to industrial projects is that stakeholders with little engineering background may experience difficulty comprehending the specification. Forming a common understanding of a specification is indeed essential in software development because a specification is consulted by many kinds of stakeholders, including those who do not necessarily have an engineering background.

This paper introduces ViennaDoc, a specification documentation tool that interleaves animation of a formal specification into informal texts written using natural language. ViennaDoc helps readers to understand the behaviour of the specified system by providing opportunities to verify their understanding by executing the specification in the context of the informal explanation. ViennaDoc also helps maintainers of the specification by enabling unit testing that asserts equality between values embedded in the informal specification with formal expressions.

1 Introduction

A development team needs a common understanding of the system to be developed. Ambiguity, looseness and inconsistency in a specification may result in disagreement in understanding among the development stakeholders. If the product owner and user representatives understand the specification differently from the specifier's intension, the development team may produce a useless system.

VDM-SL [5] is a formal specification language with rigorous syntax and semantics, and therefore gives an unambiguous meaning to the specification. However, the mathematical meaning of a formal specification is not enough to avoid disagreement of understanding because the specification of a software system should also be situated in the context of the real world as well. In industrial projects, informal documents in a natural language are created along with formal specifications. Such informal documents typically explain the operational usage of the formal specification to stakeholders unfamiliar with formal notations, and illustrate the concrete situations that the formally specified system is expected to be used inside. In this paper, we call such informal descriptions *specification documents* to distinguish them from formal specifications.

Specification documents are literature to explain the meaning of the specification. Conventional IDEs for VDM, such as VDMTools [3] and the Overture tool [4], support the VDM-SL's literate format that embed VDM-SL modules in a LaTeX document. Although LaTeX is a practical tool to print documents such as books, its main target is to print on paper and to produce PDF files with static contents, and it does not afford dynamic representations that are suitable for explaining dynamic behaviour of the system in particular scenarios.

Specification animation is a technique to simulate behaviour of the system to be realised [6,8]. Animation has been used mainly by formal engineers in practice. Integrated Development Environments (IDEs) for VDM provide animation mechanism including interpreters and transpilers so that the specification engineers can confirm the specification *means* as intended. Those IDEs also provide unit testing mechanisms to automatically test whether or not the operational meaning of the specifications is kept as intended.

The authors have been developing a specification environment called ViennaTalk that supports stakeholders with little engineering skills to understand the specified system in contexts of specific domains [9]. One of the major objectives of ViennaTalk is to enable more features for animation techniques for a wider range of stakeholders. A stakeholder, in general, uses animation to check whether the stakeholder's understanding is correct or not. The stakeholder typically has a particular situation in mind, and reproduces it in the animation. If the animation provides the expected behaviour, the stakeholder gains confidence on the validity of the specification and also the correctness of the stakeholder's understanding. Animation techniques can thus be used in the context of comprehension support.

It is, however, often hard for stakeholders without sufficient training in formal specifications to articulate an expression to evaluate. Some of the difficulties are of the VDM-SL language features such as its formal syntax and semantics. ViennaTalk addresses the language difficulty by collaborative approaches. Lively Walk-Through is a User Interface (UI) prototyping environment provided in ViennaTalk to develop shared understanding of the formal specification between formal engineers and UI designers. Other stakeholders, such as product owners and end user representatives, can also use Lively Walk-Through to experience the specified system. Those systems support stakeholders to understand formal specifications without knowledge of VDM-SL.

This paper introduces ViennaDoc, a web publishing tool for VDM-SL specification documents with live animation and testing features. In ViennaDoc, formal expressions expressed using VDM-SL are placed along with use scenarios of the system. Stakeholders with little formal background can understand the scenario exemplified with predefined animations without writing any formal expressions. ViennaDoc can also help maintenance of the specification document by unit testing. ViennaDoc is developed as a part of ViennaTalk and its source code is available under the MIT license[7].

In Section 2, overall design objectives of ViennaDoc will be presented. In Section 3, implementation and features of ViennaDoc are explained and discussed. Section 4 will introduce related work, and Section 5 will conclude the discussion and describes possible future work.

[7] `https://github.com/tomooda/ViennaTalk/`

2 Informal Documents Augmented with Formal Specifications

Formal specification often complements informal specification documents for many purposes. A specification document is not only for stakeholders with little engineering background, but also for formal engineers to understand how the specified system will be used in the real world. Situating the formal specification is a critical role of its documentation for stakeholders to share a common understanding of the system to be developed. Formal engineers can use interpreters to confirm the specified system exhibits the intended behaviour in particular situations. However, the use of animation was mainly limited to formal engineers due to difficulty of articulating appropriate formal expressions.

Inconsistency between a specification and its documentation may become one of the barriers for stakeholders to have a shared understanding. For example, a piece of specification that appears in a specification document should be the right version. The effects of an action are often explained informally in a specification document. Such an explanation must naturally agree with the behaviour of the specification. Nevertheless, it is not a trivial task to maintain a documentation consistent with the formal specification. A specification document needs to be revised for many reasons, such as a change of the formal specification, a change of the expected uses of the system, and inaccurate discourse of the explanation. Animation has been used to test formal specifications in practice, and case studies report that those tests were effective [2]. However, the use of animation in testing was limited to the formal specification.

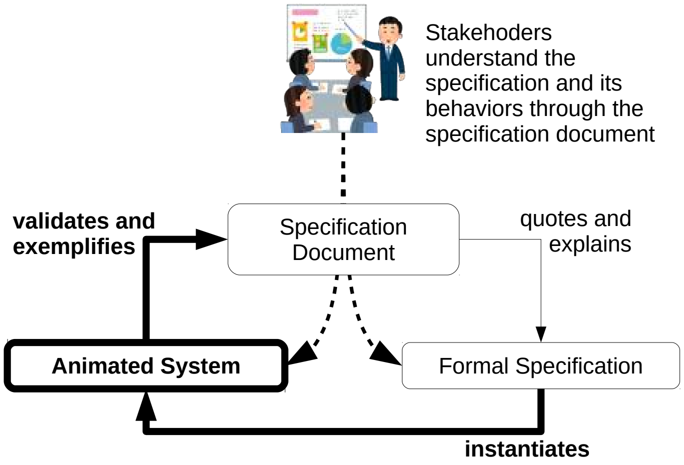

Stakehoders understand the specification and its behaviors through the specification document
validates and exemplifies
Specification Document
quotes and explains
Animated System
Formal Specification
instantiates

Fig. 1. Specification, Specification Document and Animation

This paper proposes to apply specification animation techniques to informal specification documents. Figure 1 describes the relationship among a formal specification, its documentation and the animated system. This paper discusses roles of animation mechanisms in the documentation environment, drawn in the thick box and the thick arrows. ViennaDoc is a simple animation tool for HTML documents on a VDM-SL specification. ViennaDoc provides the following three features to support documentation:

- Source expansion to include a VDM-SL source;
- animation to illustrate the system's behaviour; and
- assertion to validate values in the document.

The source expansion feature helps the author of a specification document to keep the embedded source up to date. It is a simple and effective way to maintain document without spoiling agility of the specification phase.

The animation feature is to explain the system's behaviour in a particular scenario. When a user scenario is explained in natural language, the user can see how the system will behave in the scenario by executing an operation with the given contextual state of the system. The user can even explore different series of actions by performing animation in an arbitrary order. Animation can be used for exercises to check whether the reader's understanding is correct. This flexibility cannot be provided by a static PDF file.

The assertion is a declaration of a certain property of the system or a part of it, and software testing is one of its major applications. The assertion in ViennaDoc is to test the specification document whether or not a VDM-SL value displayed in a document is equal to the result of evaluating the given expression in the given state of the system. If they are not equal, the document is inconsistent with the formal specification.

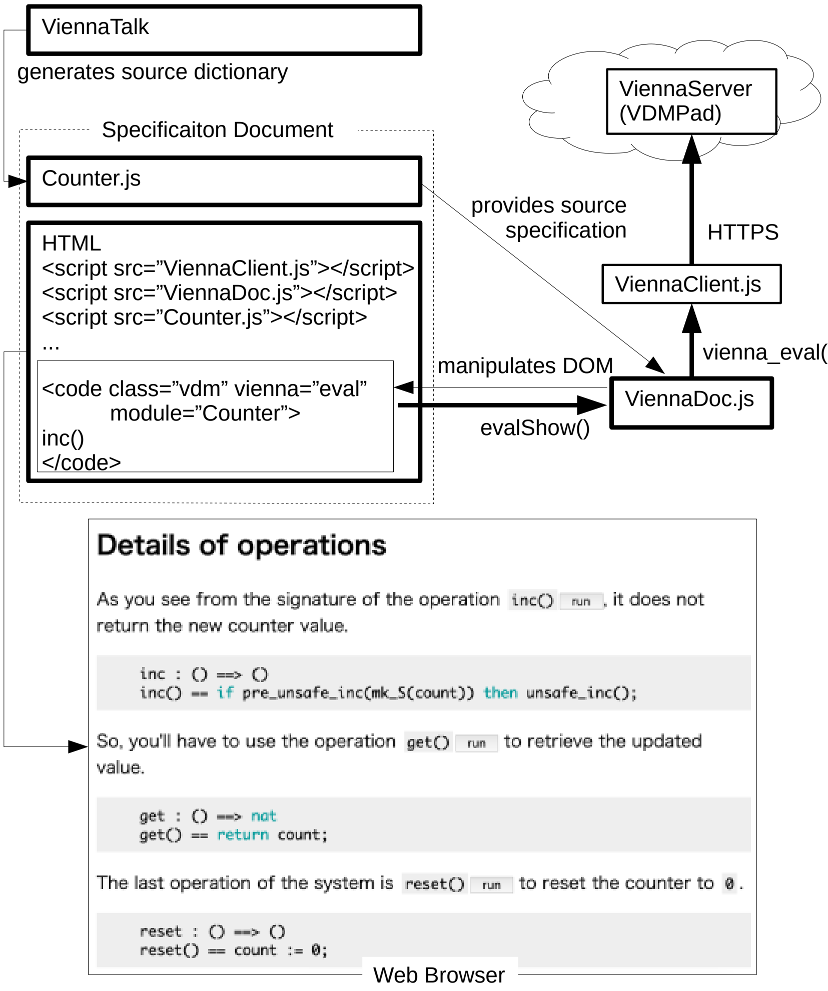
Section 3 will describe how ViennaDoc is implemented and how it works inside a web browser.

3 Implementation

ViennaDoc is implemented in JavaScript enabling HTML documents to animate a VDM-SL specification. Figure 2 illustrates the overview of ViennaDoc and related components. The `ViennaDoc` object provided by the `ViennaDoc.js` script has two responsibilities: 1) to perform animation and assertion and 2) to scan the Document Object Model (DOM) to insert the specification sources, animation mechanisms and assertions into the ViennaDoc specific DOM elements. The HTML file also loads a specification specific script (`Counter.js` in Figure 2) generated by ViennaTalk [9]. The specification specific script defines a dictionary of source code for each module, type, value, function, state and operation.

The `ViennaDoc.js` script uses the public VDMPad [9] server[8] via the functions provided by the `ViennaClient.js` script. The `ViennaClient.js` script extends the `String` object with the `vienna_eval()` method. To animate a specification, the source specification, the state before evaluation, the expression to evaluate and the namespace (module) must be specified. The `vienna_eval()` method uses the receiver string as the VDM-SL source specification. The expression to evaluate is given

[8] `https://vdmpad.viennatalk.org/`



Details of operations

As you see from the signature of the operation `inc()` run , it does not return the new counter value.

```
inc : () ==> ()
inc() == if pre_unsafe_inc(mk_S(count)) then unsafe_inc();
```

So, you'll have to use the operation `get()` run to retrieve the updated value.

```
get : () ==> nat
get() == return count;
```

The last operation of the system is `reset()` run to reset the counter to `0`.

```
reset : () ==> ()
reset() == count := 0;
```

Fig. 2. Overview configuration of a typical ViennaDoc document

as the first argument. The state before evaluation and the module are optionally given as the second and the third arguments in order. The return value of the `vienna_eval()` method is triple of the return value, the post-state, and the error message.

The `ViennaDoc.evalShow()` method is called by an event on a DOM element, such as a clicked event on a `code` element. The `ViennaDoc.evalShow()` method manipulates the requesting DOM element to show the animation result. The `String.vienna_eval()` method is used to get the animation result.

The `ViennaDoc.js` script scans the HTML document for ViennaDoc specific elements, that are `code` elements with both the `vdm` class and `vienna` attribute. The `ViennaDoc.js` script inserts DOM elements to inject functions to run animation and show the results. The `ViennaDoc.js` script also manages DOM elements that need active updates at every animation step.

Table 1. ViennaDoc attributes on the `code` element

`vienna` **attribute**	**Other attributes**	**Description**
`"source"`	`src` = module name or global name	include source code specified by the `src` attribute
`"eval"`	`prestates` = variable list (optional) `poststates` = state variables (optional) `module` = module name (optional)	insert a `run` button that evaluates the content and prints the return value and pre/post states
`"watch"`	`module` = module name (optional)	always print the latest value of the variable specified in the content
`"assert"`	`eval` = expression `prestates` = state variables (optional) `module` = module name (optional)	evaluate the given expression and insert a warning message if the the result is not equal to the content

Table 1 summarises the ViennaDoc specific elements. The `ViennaDoc.js` script scans for `code` elements with the `vienna` attribute when the HTML has been loaded to the browser. ViennaDoc currently supports four values of the `vienna` attribute; `source`, `eval`, `watch` and `assert`. The rest of this section will explain each kind of `code` elements one by one.

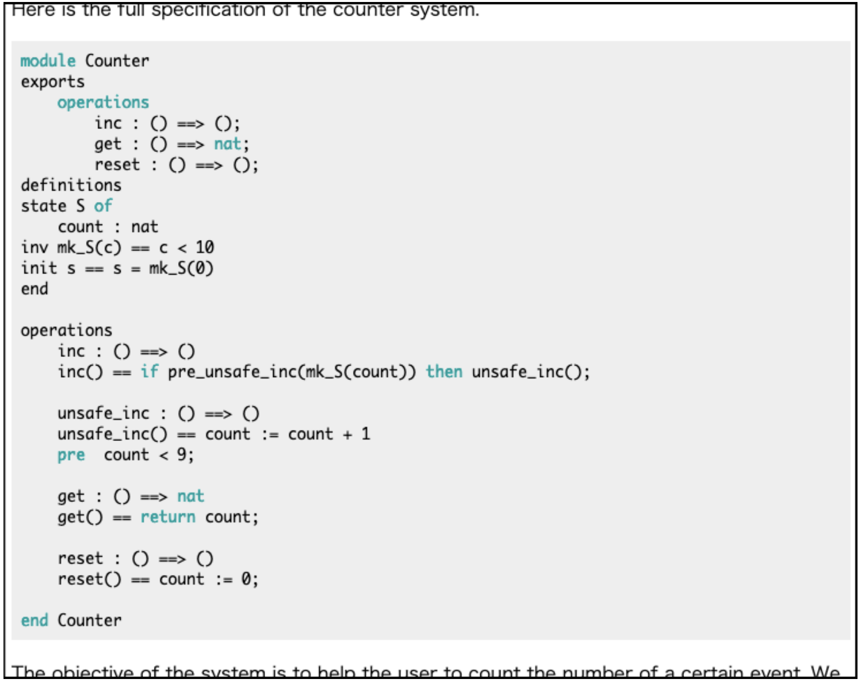
3.1 Embedding Specification in Documents

ViennaDoc provides access to source listings of modules and definitions. Figure 3 illustrates how a `code` element for source listing is specified in ViennaDoc and how it appears inside a browser.

A `code` element with the `vienna="source"` attribute fills the element with the corresponding source specified by the `src` attribute. In Figure 3, the `Counter` module is specified by the `src` attribute. Using the source embedding feature of ViennaDoc, the source listings shown in the document are kept identical to the ones used for the animation.

```
<code class="vdm" vienna="source" src="Counter"></code>
```

⇓ rendered on a browser



Here is the full specification of the counter system.

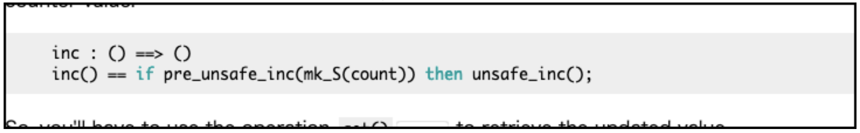
```
module Counter
exports
    operations
        inc : () ==> ();
        get : () ==> nat;
        reset : () ==> ();
definitions
state S of
    count : nat
inv mk_S(c) == c < 10
init s == s = mk_S(0)
end

operations
    inc : () ==> ()
    inc() == if pre_unsafe_inc(mk_S(count)) then unsafe_inc();

    unsafe_inc : () ==> ()
    unsafe_inc() == count := count + 1
    pre  count < 9;

    get : () ==> nat
    get() == return count;

    reset : () ==> ()
    reset() == count := 0;

end Counter
```

The objective of the system is to help the user to count the number of a certain event. We

Fig. 3. Screenshot of a ViennaDoc page that includes a module source

```
<code class="vdm" vienna="source" src="Counter`inc"></code>
```

⇓ rendered on a browser



counter value.

```
inc : () ==> ()
inc() == if pre_unsafe_inc(mk_S(count)) then unsafe_inc();
```

So, you'll have to use the operation get() to retrieve the updated value.

Fig. 4. Screenshot of a ViennaDoc page that includes an operation's source

The value of the src attribute is either a module name or a global name. In Figure 4, the Counter`unsafe_inc operation is specified by the src attribute. Please note that a global name should be specified to include a source listing of an operation, function or any other definition. A global name should be in the form of $\langle module\ name \rangle$ ` $\langle identifier \rangle$, and the source is stored at ViennaDoc.sources[$\langle global\ name \rangle$]. ViennaDoc needs the source listings from an external source for source embedding as well as animation and assertions. ViennaTalk's VDM browser generates a JavaScript source code to supply the source listings.

ViennaDoc automatically enclose the code element with a pre element so that newline and indentations are preserved when rendered on the browser. Although ViennaDoc does not have any feature to decorate source code, the HTML file can use an external engine to highlight the source specifications inserted by ViennaDoc. For example, the source listings in Figure 3 and Figure 4 are highlighted by the highlight.js[9] package.

3.2 Animated Specification Documents

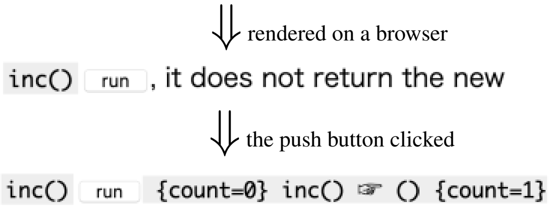
ViennaDoc employs animation to exhibit concrete behaviour of the specified system. ViennaDoc provides user interfaces for animation in a controlled manner comparing to those in IDEs. For example, VDMPad enables the user to evaluate an arbitrary expression in an arbitrary state by providing entry fields for state variables and the expression to evaluate. VDMPad allows the user as much freedom as possible so that the user can explore a wider range of use scenarios. On the other hand, ViennaDoc does not allow the user to change the expression to evaluate. The objective of the animation capability inside ViennaDoc is to help the user's comprehension of the specification document in a natural language by showing simulated behaviour of the system. Agreement between the explanation in natural language and animation in a formal language is crucial and therefore the expression to evaluate is hardcoded.

ViennaDoc provides two kinds of user interfaces for animation. One is a push button. ViennaDoc.js inserts a push button after a code element with the vienna="eval" attribute. Figure 5 shows how a push button is placed in a ViennaDoc document and how it can present the animation result. The code element shown in the upper box is rendered on a web browser while the middle box shows the rendered element. A push button labeled run is placed after the code node (inc() in a grey background). When the user clicks on the run button, a brief text in the gray background is appended after the run button as shown in the lower box. The appended text is in the form of $\{bindings\ before\ evaluation\}expr$☞$ret\ value\{bindings\ after\ evaluation\}$.

The animation state is managed by the ViennaDoc.states variable. At every animation, ViennaDoc first reads the state from the ViennaDoc.states, evaluates the given expression, and updates the ViennaDoc.states. If the prestates attribute has bindings indicated by =, the value will be used instead of the corresponding value in the ViennaDoc.states. In Figure 6, the value of the state variable count is set to 5 before evaluating inc().

[9] See https://highlightjs.org/

```
<code class="vdm" vienna="eval"
  prestates="count"
  poststates="count"
  module="Counter">
inc()
</code>
, it does not return the new
```



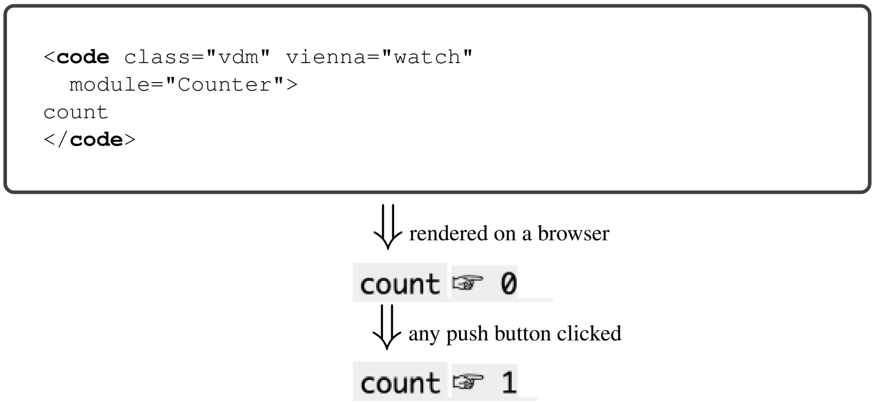
inc() run , it does not return the new

inc() run {count=0} inc() ☞ () {count=1}

Fig. 5. Animation of a ViennaDoc page with a run button

```
<code class="vdm" vienna="eval"
  prestates="count=5"
  poststates="count"
  module="Counter">
inc()
</code>
```

Fig. 6. An HTML source for animation with a prestate value specified



```
<code class="vdm" vienna="watch"
  module="Counter">
count
</code>
```

Fig. 7. Animation of a ViennaDoc page with a watch variable

Another user interface for animation is a watch variable. `ViennaDoc.js` inserts a brief text after a `code` element with the `vienna="watch"` attribute. Figure 7 shows how a watch variable is shown in a ViennaDoc. The `code` element shown in the upper box is rendered on a web browser as shown in the middle box. The text ☞`0` in a grey background is placed after the `code` node (`count` also in the grey background) to indicate that the value of the state variable `count` is now set to `0`. Whenever the user clicks on any of the `run` buttons in the document, the value after the `hand` symbol is updated with the new value of the variable `count`.

In Figure 7, the module `Counter` is specified in the `code` element and its content is in a short name of the variable inside that module. A global name in form of *module name*`*state variable name* can be specified in the content without the `module` attribute.

The state variables are managed as a JavaScript object on the page. If the user reloads the page, the state will be initialised by the specification. The state will also be initialised when the web page transitions by hyperlinks. ViennaDoc has been designed to employ the volatile state that a ViennaDoc page always opens with a fresh initial state.

3.3 Testable Specification Documents

Software testing is a widely used technique to find a deviation from the expected behaviour. ViennaDoc provides testing feature on a HTML document to find a deviation from the actual behaviour of the formal specification. ViennaDoc performs a test on a `code` element with the `vienna=assert` attribute.

Figure 8 shows a successful case of the assertion. ViennaDoc evaluates the given expression `mk_(reset(), count).#2` with `count=5`, which returns the value of the `count` variable after the `reset()` operation call. ViennaDoc will get `0` as the

```
<code class="vdm" vienna="assert"
  module="Counter"
  prestates="count=5"
  eval="mk_(reset(),␣count).#2">
0
</code>
```

⇓ rendered on a browser

0.

Fig. 8. A successful assertion and its appearance on a browser

result which is identical to the value in the content of the code element. The assertion is successful and the web browser shows the code element without modification.

```
<code class="vdm" vienna="assert"
  module="Counter"
  prestates="count=5"
  eval="mk_(inc(),␣count).#2">
0
</code>
```

⇓ rendered on a browser

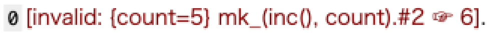

0 [invalid: {count=5} mk_(inc(), count).#2 ☞ 6].

Fig. 9. An assertion failure and its appearance on a browser

On the other hand, Figure 8 shows a failure case. ViennaDoc evaluates the given expression mk_(inc(), count).#2 with count=5, which returns the value of the count variable after the inc() operation call instead of the reset() operation call. ViennaDoc will get 6 as the result which is different from 0 in the content of the code element. The assertion fails and the web browser warns that the value in the document does not agree with the actual result.

Assertion failures may happen due to various causes. One possible cause of failure is that the document is not updated after a change of the VDM-SL specification. Another possible cause is that the author misunderstands the formal specification. ViennaDoc's assertion enables specification documents tested its consistency with the formal specification on the readers' web browsers just in time.

4 Related Work

This section introduces four software tools and explains their differences from this work.

4.1 Lively Walk-Through

Lively Walk-Through [9] is a UI prototyping tool to support formal methods engineers and UI designers to share a common understanding of the specified system. Lively Walk-Through can be also used as a prototyping tool for stakeholders with little engineering background to experience the expected use of the system in the specific domain. ViennaDoc and Lively Walk-Through share the same objective to provide a dynamic medium for stakeholders without formal engineering skills to understand what the formal specification means through animation.

A major difference between ViennaDoc and Lively Walk-Through is flexibility of animation. ViennaDoc provides a restrictive UI for animation while Lively Walk-Through provides full flexibility to evaluate arbitrary expressions. ViennaDoc is a documentation tool and therefore takes a more instructive approach to explain the formal specification. Lively Walk-Through does not provide textual explanations in natural languages, and thus expects emergent understanding in action.

4.2 PVSio-web

PVSio-web [7] is a prototyping environment that combines formal specification in PVS and human interface. One major application field of PVSio-web is embedded systems such as medical devices. With PVSio-web, the user can virtually manipulate the target device with feedback by animated specification.

A major difference between ViennaDoc and PVSio-web is the way the presentation is made. While PVSio-web is a simulator with photo realistic visual interface of the target device, ViennaDoc is documentation tool with textual descriptions.

4.3 Pillar

Pillar [1] is a documentation tool built on top of the Pharo Smalltalk system. Pillar has its own lightweight markup language with many dynamic features such as importing the source code from a live Smalltalk system, programmatically taking a snapshot image of the GUI, evaluating Smalltalk expressions, and testing assertions attached with Smalltalk expressions. Pillar can publish a document in various formats including LaTeX, PDF, HTML and Smalltalk's Text objects. Pillar has already been used for publishing numerous books and online documents shared among the Pharo community.

Agile Visualization[10] is a book authored with Pillar and published in paperback, eBook and HTML.

The conceptual design of ViennaDoc was highly inspired by Pillar. Both ViennaDoc and Pillar can incorporate source code into the document. Pillar also checks consistency between a book content and the actual behaviour of the system using assertion mechanisms in a way similar to that of ViennaDoc.

The major difference is that Pillar processes dynamic elements in a document source within Pillar itself. For example, a document is generated after assertions in the document source have been all checked. The readers of the book do not see its validation process. ViennaDoc, on the other hand, provides dynamic features such as the `run` button and watch variables on the browser. Animations performed on those elements are commenced by the reader, and the animation results are exposed to the reader. Assertions are also checked on the reader's browser, and the results are rendered inline on the browser.

4.4 Jupyer Notebook

Jupyter Notebook is a web-based interpreter interface initially for Python. Jupyter Notebook basically provides a REPL (Read, Eval, Print Loop) interface in a form of a webpage, and the server runs the python interpreter to execute the commands given by the user. Code fragments given by the user and the resulting output of the python interpreter are recorded on the webpage. The user can review those command lines and outputs to summarise the exploratory process and store it for later reference. The resulting page can be seen as a documentation of the exploration process.

Both Jupyter Notebook and ViennaDoc provides an execution engine on the web and help the user understand code fragments. Although they have common technical elements such as dynamic web content, evaluation engine on server side, and mixed use of natural languages and computational language, their objectives and supposed users are different. Jupyter Notebook is a tool for exploration. The user is working on an experimental code and review the entire process after the exploration. The author and the reader of the page is the same person. Jupyter Notebook can be considered a personal medium.

ViennaDoc is, on the other hand, a tool for explanation. The author of a ViennaDoc page have already explored possible ways of modelling the system. The author documents the specification by providing explanation. The reader comprehend the document with help of animation. The author and the reader of the ViennaDoc page are different users. ViennaDoc is inherently a communication medium.

5 Concluding Remarks

Lightweight formal methods inherently need to work with informal notations and methods. ViennaDoc is an attempt to utilise the specification animation techniques in informal documents. Although we have been focused on specification documents, we believe this approach could be applied to many other formal specification tools.

[10] `http://agilevisualization.com/`

One possible future work direction is using this inside VDM-SL tutorials. Many online tutorials for programming language learners provide functionality to evaluate a piece of code. ViennaDoc's animation feature can provide such a playground for VDM. Tutorials also need to be maintained along with growth of the languages and libraries. ViennaDoc's assertion mechanism could help authors of tutorials to keep them conformant to the latest version of the language and libraries.

Another possible application is for other open source communities. There are many emerging open source efforts associated with VDM. ViennaDoc is an HTML-based technology and thus could be an affordable medium for open source communities. The authors expect ViennaDoc could support more use of VDM in open source projects.

Acknowledgments

The authors gratefully acknowledge Stèphane Ducasse for his inspiring comments on the initial idea of this research. We would also thank anonymous reviewers for their thoughtful and constructive feedback. A part of this research was supported by JSPS KAKENHI Grant Number JP 18K18033.

References

1. Arloing, T., Dubois, Y., Ducasse, S., Cassou, D.: Pillar: A versatile and extensible lightweight markup language. In: Proceedings of the 11th edition of the International Workshop on Smalltalk Technologies. p. 25. ACM (2016)
2. Kurita, T., Chiba, M., Nakatsugawa, Y.: Application of a Formal Specification Language in the Development of the "Mobile FeliCa" IC Chip Firmware for Embedding in Mobile Phone. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008: Formal Methods. pp. 425–429. Lecture Notes in Computer Science, Springer-Verlag (May 2008)
3. Larsen, P.G.: Ten Years of Historical Development: "Bootstrapping" VDMTools. Journal of Universal Computer Science 7(8), 692–709 (2001)
4. 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
5. 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)
6. 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
7. Masci, P., Couto, L.D., Larsen, P.G., Curzon, P.: Integrating the PVSio-web modelling and prototyping environment with Overture. In: Ishikawa, F., Larsen, P.G. (eds.) Proceedings of the 13th Overture Workshop. pp. 33–47. 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

8. 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
9. Oda, T., Araki, K.: ViennaTalk: An Integrated Specification Environment Focused on the Early Stage of the Formal Specification Phase. Computer Software 34(4), 4_129–4_143 (November 2017), `https://www.jstage.jst.go.jp/article/jssst/34/4/34_4_129/_article/-char/ja/`