PHD DISSERTATION

Enhancing Formal Modelling Tool Support with Increased Automation

by Kenneth Lausdahl

AARHUS
UNIVERSITY

Enhancing Formal Modelling Tool Support with Increased Automation

Enhancing Formal Modelling Tool Support with Increased Automation

PhD Thesis by

Kenneth Lausdahl

Aarhus University Department of Engineering, Denmark

AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

Aalborg

ISBN 978-87-93102-02-6 (e-book)

Published, sold and distributed by:
River Publishers
P.O. Box 1657
Algade 42
9000 Aalborg
Denmark

Tel.: +45369953197
www.riverpublishers.com

Abstract

The intrinsic complexity of even simple software systems makes their development challenging. This is especially true for heterogeneous embedded control systems that include the constraints of the physical world. Formal methods and modelling techniques allow software designs to be analysed and thus contribute to their reliability and robustness. However, industrial adoption of formal methods is limited by uncertainties related to their cost, effectiveness and the skills required. This thesis has three areas of focus: manual labour reduction through automation of various kinds of analysis, with focus on validation; language translation utilised to exploit existing automated analysis techniques; and a way to model and simulate embedded control systems that demand high-fidelity representation of their environment, based on a sound formal foundation. The result is a collection of contributions to a modern integrated development environment that can analyse software specifications and simulate hybrid embedded control systems with their environment.

Resumé

Udviklingen af selv små software systemer er en kompleks udfordring: især for heterogene indlejrede kontrolsystemer, som også tager højde for dynamikken i den fysiske verden. Formelle metoder og tilsvarende modelleringsteknikker muliggør analyse af software design. Det kan være medvirkende til at øge pålideligheden og derved sikre et mere robust system. Dog har den industrielle anvendelse af formelle metoder været begrænset, primært grundet usikkerhed i forhold til omkostningerne og den nødvendige tekniske ekspertise.

Denne afhandling har tre fokus områder: reduktion af manuelt arbejde gennem automatisering af forskellige former for analyser primært med fokus på validering; udnyttelse af eksisterende automatiserede analyse teknikker ved hjælp af sprog oversættelse; og simulering af indlejrede kontrolsystemer, der kræver en meget detaljeret og nøjagtig repræsentation af det dynamiske miljø systemet opererer i. Der præsenteres adskillige bidrag til analyse af software abstraktioner som alle bidrager til et moderne integreret udviklingsmiljø.

Acknowledgements

I would like to thank my supervisor Peter Gorm Larsen from the Department of Engineering, University of Aarhus, for providing valuable feedback during my PhD. It has been a great pleasure and great continuous source of inspiration to work with Peter, who always sees opportunities. His professionalism and dedication is second to none. I would also like to thank my co-supervisor Erik Ernst from the Computer Science Department, University of Aarhus, for his impressive sense for detail and feedback that he has provided during the development of this PhD thesis.

I give thanks to the DESTECS project for funding my PhD. I have had so many useful discussions with this group of talented people, who have also given me the opportunity to get valuable feedback from industrial partners. I would like to give a special thanks to my colleagues in the DESTECS project at Controllab Products Peter Visser and Frank Groen.

I would like to express my gratitude to Daniel Jackson and his software design group at the Computer Science and Artificial Intelligence Lab at Massachusetts Institute of Technology for giving me the opportunity to visit them in Cambridge, MA, USA, and to Aarhus university research foundation and Fondation Idella for providing the necessary funding this trip.

I would also like to thank all my close collaborators: Nick Battle, Joey W. Coleman and Augusto Ribeiro. The collaboration with these skilled people have helped me develop my competences as a researcher. I have enjoyed our many valuable discussions that have given served as inspiring variation. In particular, I admire their dedication to detail.

I would like to thank Claus Ballegaard Nielsen and Sune Wolff for their camaraderi and many small interactions as well as the feedback they have provided during the development of this PhD thesis. I would also like to thank Stefan Hallerstede for his valuable feedback and review of this PhD thesis.

I would like to thank my parents, Yvonne and Henning Lausdahl, for loving and supporting me unconditionally, and for giving me a good foundation for life. I also thank my sister Leni Lausdahl, for being the best sister

in the world. My family has been a source of constant encouragement and unconditional support.

Last, but not least, I would like to thank my beautiful wife Tenna Lausdahl for her love, support and can-do spirit. I would especially like to thank her for giving my two wonderful daughters, Marie and Laura, the apples of my eye and a constant source of smiles. I would also like to thank the three of them for joining me on multiple occasions when research has lead me abroad. This thesis is dedicated to my wife and my two daughters.

Contents

Part I

Overview

1

Introduction

People are surrounded by devices that contain software, and daily life is becoming increasingly more dependent upon the services offered by software in general. However, a common practice in some areas of the software industry is to release incomplete software, or software with known faults to gain a market advantage. This is made even worse by the constantly increasing complexity of software, while at the same time expectations are that it performs flawlessly. Unfortunately, this is not the case, and most software is faulty and expensive to develop partially due to the increase in complexity. Faulty software poses a significant risk in safety critical systems where either machinery or humans might be damaged or hurt by errors. The consequence of the increase in complexity and expectations is that a large number of software projects fail and either get cancelled or run over budget, and still deliver fewer features than initially claimed [136, 59].

The software engineering discipline is relatively new compared to other disciplines such as mechanical engineering that dates back to the medieval times and was deeply influenced by the work of Archimedes (287 BC) and later Heron of Alexandria (c. 10-70 AD) who created the first steam engine. The mechanical engineering discipline uses modelling techniques to accurately predict the behaviour of a final system. This ability to accurately design and later build the same solution still remains unsolved for software engineering.

The term *software engineering* was introduced in the 1968 NATO Software Engineering Conference [106], 45 years ago; where concepts were introduced for software development. Unfortunately, the concepts introduced at that conference are not universally used in software development. The modelling concept allows engineers to formally describe software systems based on their informal requirements and thus unambiguously capture interactions with devices external to the system; the result is a formal specification that thus can be analysed. The specification can then be *validated* against the

informal requirements by simulation, thus checking that the specification represents the intended system. A specification can also be *verified* with respect to its internal consistency by e.g. checking for contradictions in the form of type incomparability, conflicting invariants etc. These types of analysis are challenging to perform by hand and, therefore, automated analysis is a significant advantage since it reduces human errors and in most cases speeds up analysis by an order of magnitude [129].

1.1 Modelling of Software Systems

Modelling is the process of capturing and describing knowledge about a system in an abstract form. The result is a *specification* that capture the requirements for a given system and its behaviour. A specification is described using a modelling language that focuses on certain types of analysis. Various modelling languages exist which can be used to describe a system; the languages can either be graphical using diagrammatic techniques (e.g. the Unified Modelling Language), or textual using standardized keywords in a natural language or with terms that make the language interpretable by computers. These interpretable languages can be formal modelling languages with a concrete syntax and well defined semantics that is built on mathematics and therefore provides a base of analysis.

The nature of a system may favour some modelling languages over others, e.g. a software ticketing system might require a language capable of describing relations, while an embedded reactive control system might require the ability to describe the complex behaviour of the controller in the system but also the ability to faithfully represent the physical environment it operates within. The first case of a simple software system may be modelled by any of the languages like: UML, Alloy, Z, B or VDM from the discrete-event (DE) domain; however, the second category requires a more complex language that not only has the capability to describe elements of the DE domain but also the ability to describe physical elements of the environment, that belong to the continuous-time (CT) domain. To describe such systems a language that spans multiple domains is needed either in the form of a single language or two languages that integrate with each other.

The motivation behind the creation of a specification of a system is to describe the system in a clear and concise way and then through analysis check that the specification is consistent and unambiguous and reflects the requirements. A critical factor is the ability to automatically check such specifications for internal consistency and validation against external criteria e.g.

requirements. This is supported by formal modelling languages due to their mathematical foundations.

1.2 Formal Modelling Languages

This section gives a brief description of a few of the existing formal modelling languages (Z, B, Event-B, Alloy and VDM) which have been used both in academia and industry. The formalisms are presented with a description of the available tools that can be used for various kinds of analysis. The section presents an extended description of VDM which is the language used throughout this thesis.

1.2.1 Z-notation

The Z-notation was developed at Oxford University in the 1980's; it was originally proposed by Jean-Raymond Abrial in 1977, based on the publication [2]. The Z-notation was later ISO standardised [56], a number of books have been written about the Z-notation [58, 122, 156, 158]. It is based on the standard mathematical notation used in axiomatic set theory, lambda calculus and first-order predicate logic. The logic is augmented with structure (*schemas*) to make it easy to describe software systems, i.e. *schema calculus*. Specifications are built as a collection of schemas. The most popular version of Z is the version from Mike Spivey's book [134]. Z, unlike B, does not have built-in refinement, and thus many users view Z as a system modelling language and have no intent of proving the conformance of code to the specification. However, a well-established theory to refinement of Z exists, as introduced in Woodcock's book [156]

Tools for Z

The tools available for Z focuses on theorem proving. The most widely used proof tools are ProofPower from Lemma 1 Ltd., and Z/Eves, a front-end to the Eves theorem prover from ORA Canada. Eves can calculate preconditions and perform domain checks (checking that partial functions are not applied outside their domain) as well as general theorem proving. The tool is automated but still requires expert assistance. Mark Utting has developed an animator for Z named Jaza[1]. It can execute operations written in an explicit style, evaluate expressions, check state against invariants, and so on. Finally,

[1] `http://www.cs.waikato.ac.nz/~marku/jaza/`

the Community Z Tools (CTZ) project[2] aims to combine tools for Z. The tool currently includes infrastructure required to parse, type check, edit and animate Z (ALive).

1.2.2 B-Method and Event-B

The B-Method [1] is a method for software development based on the B notation, it is based around an abstract machine notation for e.g. embedded software development. The method was originally developed by Jean-Raymond Abrial and is a proof-based refinement method. The B notation is related to Z, but B is closer to the implementation level, with focus on formal refinement to code compared to just formal specification. The formal method Event-B [3] is an evolution of B and has a simpler notation. It focuses more on system-level modelling, and is thus less suited for embedded software. Event-B is also a refinement method, and it has been used in the DEPLOY project[3] [130, 129]. Furthermore, Event-B forms the base in a current research project Advanced Design and Verification Environment for Cyber-physical System Engineering (ADVANCE[4]).

Tools for B and Event-B

The B-method is supported by the commercial tool *Atelier B*[5] from ClearSy, a French company. Event-B is supported by the open-source tool *Rodin*[6]. Furthermore, an extension *ProB* provides model checking and animation to Event-B specifications [96].

1.2.3 Alloy

Alloy [57] is a declarative formal specification language for describing software abstractions; it was developed to adapt a declarative language like Z to bring in fully automatic analysis. At the core, Alloy is based on relations over atoms with a logic that is small, simple and expressive. It is based on a relational logic that combines the quantifiers of first-order logic with operators of relational calculus. It is easy to learn and understand if one already is familiar with basic set theory. The Alloy language is more than just logic; it provides

[2] http://czt.sourceforge.net
[3] http://www.deploy-project.eu/
[4] http://www.advance-ict.eu/
[5] http://www.atelierb.eu/en/
[6] http://sf.net/projects/rodin-b-sharp

ways to organise a model, build larger models based on smaller ones and a way to factor out components for reuse. The language also provides a number of commands needed to communicate with the Alloy Analyzer. And, finally, the language includes modules, polymorphism, parametrized functions etc., though some features are unique to Alloy including *signatures* and the notion of *scope* that is used to describe structure and define the scope of analysis.

Tools for Alloy

The Alloy Analyzer is a bounded model finder that has proven to be useful for validating specifications in the Alloy language [57]. The analyzer can *find* instances of Alloy specifications, as well as checking user defined assertions. The analyzer can provide immediate visual feedback when an instance is found or present a *core* containing the top level formulas if no instance could be found. The analyzer is built on top of a boolean satisfiability (SAT) solver, which is used to find an instance by converting the Alloy signatures to boolean formulas, and then find an assignment to all variables so these formulas are satisfied.

1.2.4 The Vienna Development Method

The Vienna Development Method (VDM) [10, 61, 62, 31] was originally developed at the IBM laboratories in Vienna in the 1970's and, as such, it is one of the longest established formal methods. The VDM Specification Language is a language with a formally defined syntax, and both static and dynamic semantics [119, 81]. Models in VDM are based on data type definitions built from simple abstract types using booleans, natural numbers, characters and type constructors for product, union, map, (finite) set and sequences. Type membership may be restricted by predicate invariants meaning that run-time type checking is also required from an interpreter perspective. Persistent state is defined by means of typed variables, again restricted by invariants. Operations that may modify the state can be defined implicitly, using standard pre- and post-condition predicates, or explicitly, using imperative statements. Such operations denote relations between inputs, outputs and states before and after execution; note that such relations allow non-deterministic behaviour. Functions are defined in a similar way to operations, but may not refer to state variables. Recursive functions can have a **`measure`** defined for them to ensure termination [124]. Arguments passed to functions and operations are always passed by value, apart from object references.

Three different dialects exist for VDM: The ISO standard VDM Specification Language (VDM-SL) [32], the object oriented extension VDM++ [33] and a further extension of that called VDM Real Time (VDM-RT) [146, 52].

None of these dialects are generally executable since the languages permit the use of type bindings with infinite domains, or implicitly defined functions and operations, but the dialects all have subsets that can be interpreted [77]. In addition, some commonly used implicit definitions can be executed in principle [42]. A full description of the executable subset of the language can be found in [80].

The dialects VDM++ and VDM-RT allow concurrent *threads* to be defined. Such threads are synchronised using *permission predicates* that are associated with any operation that limits their allowed concurrent execution. Where pre-conditions for an operation describe the condition the caller must ensure before calling it, the permission predicate describes the condition that must be satisfied before the operation can be activated, and until that condition is satisfied the operation call is blocked. The permission predicates can refer to instance variables as well as *history counters* which indicate the number of times an operation has been requested, activated or completed for the current object. In VDM-RT, the concurrency modelling can be enhanced by deploying objects on different CPUs with busses connecting the CPUs. Operations called between CPUs can be asynchronous, so that the caller does not wait for the call to complete.

VDM-RT has a special **`system`** class where the modeller can specify the hardware architecture, including the CPUs and their bus topology; the dialect provides two predefined classes for the purpose, `CPU` and `BUS`. CPUs are instantiated with a clock speed and a *scheduling policy*, either *First-Come, First-Served (FCFS)* or *Fixed Priority (FP)*. The initial objects defined in the model can then be deployed to the declared CPUs using the CPU's `deploy` operation. Busses are defined with a transmission speed and a set of CPUs which they connect. Object instances that are not deployed to a specific CPU (and not created by an object that is deployed), are automatically deployed onto a *virtual* `CPU`. The virtual CPU is connected to all real CPUs through a *virtual* `BUS`. Virtual components are used to simulate the external environment for the model of the system being developed.

The semantics of VDM-RT has been extended with the concept of discrete time, so that all computations a thread performs take time, including the transmission of messages over a bus. Time delays can be explicitly specified by special **`duration`** and **`cycles`** statements, allowing the modeller to explicitly state that a statement or block consumes a known amount of time.

This can be specified as a number of nanoseconds or a number of CPU cycles of the CPU on which the statement is evaluated. All virtual resources are infinitely fast: calculation can be performed instantaneously consuming no time, though if an explicit duration statement is evaluated on a virtual CPU, the system time will be incremented by the duration.

Tools for VDM

Early tools for VDM, such as Adelard's SpecBox [127] were largely confined to basic static checking and pretty-printing of specifications. Currently, only two tools for VDM are actively maintained, VDMTools [28, 35] and Overture [P74].

VDMTools [35] originated with the Danish company IFAD, but is now maintained and further developed by the Japanese corporation SCSK Systems Inc. This is a closed-source product which includes syntax- and type-checking facilities, an interpreter to support testing and debugging of models, test coverage, proof obligation generators that produce formal specifications of integrity checks that cannot be performed statically, and code generators for C++ and Java. A CORBA-based Application Programmer Interface (API) allows specifications to be executed on the interpreter, but accessed through a graphical user interface, so that domain experts unfamiliar with the specification language can explore the behaviour described by the model by playing out scenarios or other test cases. The interpreter has a dynamic link library feature allowing external modules to be incorporated. VDMTools supports round-trip engineering of VDM++ specifications to UML class diagrams.

A newer tool that is under active development is the Overture tool [P74], described in detail in Section 2.1. It has the advantage over VDMTools that it is open-source and designed to be highly extensible; therefore it provides a good platform for research. The Overture project is a community-based initiative[7] that this PhD project is very much involved with, and many of the contributions in this PhD thesis are associated with different aspects of Overture.

1.3 Modelling of Physical Systems

The modelling and analysis of real system phenomena that one would like to control typically spans multiple disciplines. Models use representations of the various logical and physical laws of a subject system, however, logical

[7] Overture: `www.overturetool.org`

and physical laws use different mathematical frameworks to create useful specifications. The underlying mathematics of physical laws is usually described using differential equations, resulting in Continuous Time (CT) specifications whereas logical laws and computational specifications are usually represented using discrete mathematics in Discrete Event (DE) models. The latter is described in languages as in Section 1.2 and may be interpreted.

The CT languages can be simulated by integrating the differential equations that embodies the specification and provide values of the derivatives at any particular point in time. The term used to describe the tools which perform simulation of such differential equations is called a numerical solver; they simulate by integrating the equations in time steps. A number of different integration methods exist which primarily differ on how they calculate the size of the step, and can generally be divided into two groups: fixed step and variable step.

The fixed step methods always take a predefined step in time without taking into account the dynamics of the system. This is also known as sampling in the computer science domain. The accuracy of the method depends on the step size; the smaller the step size the more accurate the solution is, but the penalty is a slower simulation. Variable step methods use the dynamics of the system to determine how large the step size can be so that a certain accuracy is guaranteed. The advantage is that, usually, fewer steps have to be taken over the full time range. Only at points where the dynamics of the system dictate a certain accuracy, step sizes are decreased. The disadvantage is that it is not possible to know in advance what the next output time will be. Only a maximum step-size can be specified within the integration method, but this maximum is only used if the dynamics of the physical system allow it. Various simulation tools exists of which MathLab[8] Simulink[9] is probably the best known tool. In the Design Support and Tooling for Embedded Control Software (DESTECS)[10] [12, 34] an FP7 research project an alternative tools named 20-sim[11] [14] is used which provides some features similar to that of the MathLab Simulink combination. The DESTECS project provides guidelines and tools for modelling embedded software [116, 115]; the project uses a co-simulation approach that combines a discrete-event formalism, like all the formalisms presented in Section 1.2, with CT formalisms.

[8] MathLab: `http://www.mathworks.com/products/matlab`
[9] Simulink: `http://www.mathworks.com/products/simulink`
[10] DESTECS: `http://destecs.org`
[11] 20-Sim: `http://www.20sim.com`

The project also focuses on how faults can be modelled in such multi-domain systems [117].

1.4 Motivation

The adoption of formal methods, and the underlying specifications languages, by industry is highly dependent on available tool support since it affects risk and cost during development [50]. While not all believe that tools are needed as in [19] where it is mentioned that tools are neither necessary nor sufficient for an efficient formal method application. However, the survey [157] states that it is almost inconceivable that an industrial application would proceed without tools. It is also reported that the tools investigated in the survey are not felt in general to be of sufficient quality for wide-scale applications. These tools must also be available to various platforms, and integrated with features that allow for version control and co-operative development. In [153, 66, 152] the authors suggest that integration of tooling for formal methods into existing development tools may improve the industrial adoption. A key factor for the application of formal methods is that automation is capable of reducing otherwise time consuming tasks while still being flexible [153]. A successful example of how one person-month is reduced to 17 minutes of computation illustrates how automation is able to drastically reduce the time spent performing analysis of specifications [129]. Thus the overall motivation behind this PhD project is to attempt to improve the tool automation of the VDM formal method enabling more widespread industrial take up.

1.5 Research Method

The approach that was taken in this PhD project focused on how the analysis of VDM specifications could be enhanced to automate procedures that otherwise would be manual.

The method followed was based on the identification of areas where the existing support was not yet automated or capable of performing analysis. The work was driven by a number of external industrial case studies from the DESTECS project: a self-balancing scooter, a document inserting system and a dredging excavator [13].

These case studies led to the discovery of a number of limitations of the ability to either express or analyse parts of the case studies with respect to the objectives set for this PhD project. These limitations led to the identification

of contributions that could make it possible to model and perform analysis of the case studies. The contributions that were identified range from the ability to deterministically simulate discrete-event systems but also included the ability to describe and simulate systems that require high-fidelity environment representations. Other contributions focused more on the development of the discrete-event systems and understanding and communication of its behaviour.

Whenever a limitation was identified, the existing literature was searched for similar problems and corresponding solutions. If a solution was found it was adapted and extended to fit the needs, or otherwise a new theoretical solution was constructed. These were, in most cases, followed by a proof of concept implementation of the theory, and evaluation of the support on concrete VDM specifications, including the case studies. Depending on the outcome, theories were adjusted, and the prototype was improved. When a solution that could be considered as a new contribution was reached, it was reported in the form of publications to either workshops, conferences or journals.

1.6 Research Objectives

The thesis objective is to increase the applicability of existing analysis techniques for formal methods-based models, in particular those based on industrial cases and those that involve a higher degree of complexity than the usual textbook case studies. As complex specifications are often modelled using multiple tools and notations, this thesis also investigates the connections that are possible between the multiple notations used in these models.

Given that a formal method approach results in software with fewer faults, the hypothesis is that a higher degree of automated analysis will allow formal methods-based approaches to be a viable alternative for software development to the traditional approaches used in industry.

To accomplish this, it is necessary to enhance the fidelity of formal specifications in relation to the final system's environment, and also to improve and extend the analysis already available within the scope of validation and translation, while maintaining a formal semantic foundation. VDM is used throughout this work as the base formal method, and it has been shown elsewhere to be a suitable formal method for industrial applications [157, 76].

1.7 Evaluation Criteria

The developed automation has been evaluated with regards to the properties described below. The properties are defined primarily based on the industrial statements about tool requirements described in Section 1.4, and the industrial case studies from the DESTECS project.

The individual contributions made in this PhD project are numbered explicitly in Chapter 2 and 3. There is a total of 16 contributions, each of which have been evaluated using the following criteria:

Semantic foundation: The languages that automated analysis are developed for must have a well defined semantic foundation.

Language completeness: The analysis of a specification must cover the complete specification language either in the form of a single type of analysis or by combining different types of analysis. Likewise, any limitations must be clearly identified.

Reduction of manual labour: The workload of the user must be reduced by automated analysis; this includes reduction of the required user guidance for any particular kind of analysis.

Fidelity of embedded control systems: To describe and perform analysis on specifications of embedded control systems that rely heavily on high-fidelity representations of the target physical environment; it is required that support is available for modelling of both the embedded system as well as the environment.

Integration with other tools: Integration between tools supporting different languages is important if it brings value to the different stakeholders or improves the communication between stakeholders. Therefore, a translation should be enabled if the desired view or analysis already has tool support in another language.

The evaluation of the contributions is described in Section 4.3 where the level of fulfilment of all contributions are compared to each criteria. The comparison is illustrated using a chart as shown in Figure 1.1; the covered area indicates to what extent the criteria are fulfilled. The chart can be used to see the relation between contributions and fulfilment of criteria.

The contributions are motivated by the external industrial case studies and have been posed as a solution to the objectives but are not generalizable

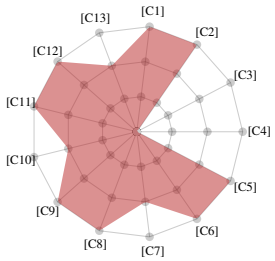


Figure 1.1: Example comparison chart.

for every possible application. However, the contributions have been incorporated into tools used by the industrial partners that enabled them to carry out their case studies with success.

1.8 Published Work

This section presents the work published during this PhD project, it mainly focuses on the topics *tool automation*, and the formal *semantics* of VDM.

1.8.1 Publications

The publications listed here are all included in this thesis in Part II.

[P74] Peter Gorm Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl, and Marcel Verhoef. *The Overture Initiative – Integrating Tools for VDM*. SIGSOFT Softw. Eng. Notes, 35(1):1–6, January 2010.

[P89] Kenneth Lausdahl, Hans Kristian Agerlund Lintrup, and Peter Gorm Larsen. *Connecting UML and VDM++ with Open Tool Support*. In Ana Cavalcanti and Dennis R. Dams, editors, Proceedings of the 2nd World Congress on Formal Methods, volume 5850 of Lecture Notes in Computer Science, pages 563–578, November 2009. Springer-Verlag. ISBN 978-3-642-05088-6.

[P83] Kenneth Lausdahl. *Translating VDM to Alloy*. In Einar Broch Johnsen and Luigia Petre, editors, Integrated Formal Methods, volume 7940 of *Lecture Notes in Computer Science*, pages 46–60. Springer Berlin Heidelberg, 2013. 10th International Conference, IFM 2013.

[P87] Kenneth Lausdahl, Peter Gorm Larsen, and Nick Battle. *A Deterministic Interpreter Simulating a Distributed Real Time System using VDM*. In Shengchao Qin and Zongyan Qiu, editors, Formal Methods and Software Engineering, volume 6991 of Lecture Notes in Computer Science, pages 179–194, 2011. Springer-Verlag. ISBN 978-3-642-24558-9.

[P79] Peter Gorm Larsen, Kenneth Lausdahl, and Nick Battle. *Combinatorial Testing for VDM*. In Proceedings of the 2010 8th IEEE International Conference on Software Engineering and Formal Methods, SEFM'10, pages 278–285, Washington, DC, USA, September 2010. IEEE Computer Society. ISBN 978-0-7695-4153-2.

[P107] Claus Ballegaard Nielsen, Kenneth Lausdahl, and Peter Gorm Larsen. *Combining VDM with Executable Code*. In John Derrick, John Fitzgerald, Stefania Gnesi, Sarfraz Khurshid, Michael Leuschel, Steve Reeves, and Elvinia Riccobene, editors, Abstract State Machines, Alloy, B, VDM, and Z, volume 7316 of Lecture Notes in Computer Science, pages 266–279, 2012. Springer-Verlag. ISBN 978-3-642- 30884-0.

[P125] Augusto Ribeiro, Kenneth Lausdahl, and Peter Gorm Larsen. *Run-Time Validation of Timing Constraints for VDM-RT Models*. In Sune Wolff and John Fitzgerald, editors, Proceedings of the 9th Overture Workshop, number ECE-TT-2 in Technical Report Series, pages 4–16, June 2011.

[P86] Kenneth Lausdahl, Joey W. Coleman, and Peter Gorm Larsen. *The Execution Semantics of VDM Real-Time in a Co-Simulation Environment*. Submitted for publication to the International Journal on Software Tools for Technology Transfer, June 2013.

[P18] Joey W. Coleman, Kenneth Lausdahl, and Peter Gorm Larsen. *Semantics for Generic Co-simulation of Heterogenous Models*. Submitted for publication to the Formal Aspects of Computing journal, April 2013.

1.8.2 Other Publications

The publications listed here have not been selected for inclusion in this thesis but are all available from the respective publishers.

[P75] Peter Gorm Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl, and Marcel Verhoef. *The Overture Initiative – Integrat-*

ing Tools for VDM. In Min Zhang and Volker Stolz, editors, Harnessing Theories for Tool Support in Software, pages 9–19, November 2010.

[P108] Claus Ballegaard Nielsen, Kenneth Lausdahl, and Peter Gorm Larsen. *Using the Overture Tool as a More General Platform*. In Franco Mazzanti, editor, iFM 2012 & ABZ 2012 - Proceedings of the Posters & Tool demos Session, pages 1–34. CNR-ISTI, June 2012.

[P90] Kenneth Lausdahl and Augusto Ribeiro. *Automated Exploration of Alternative System Architectures with VDM-RT*. In Sune Wolff and John Fitzgerald, editors, Proceedings of the 9th Overture Workshop, number ECE-TT-2 in Technical Report Series, pages 17–31, June 2011.

[P125] Augusto Ribeiro, Kenneth Lausdahl, and Peter Gorm Larsen. *Run-Time Validation of Timing Constraints for VDM-RT Models*. In Sune Wolff and John Fitzgerald, editors, Proceedings of the 9th Overture Workshop, number ECE-TT-2 in Technical Report Series, pages 4–16, June 2011.

[P91] Kenneth Lausdahl, Marcel Verhoef, Peter Gorm Larsen, and Sune Wolff. *Overview of VDM-RT Constructs and Semantic Issues*. In Ken Pierce, Nico Plat, and Sune Wolff, editors, Proceedings of the 8th Overture Workshop, number CS-TR-1224 in Technical Report Series, pages 57–67, September 2010.

[P85] Kenneth Lausdahl, Joey W. Coleman, and Peter Gorm Larsen. *Semantics of the VDM Real-Time Dialect*. Technical Report ECE-TR-13, Aarhus University, April 2013.

[P84] Kenneth Lausdahl, Joey W. Coleman, and Peter Gorm Larsen. *Towards a Co-simulation Semantics of VDM-RT/Overture and 20-sim*. In Nico Plat, Claus Ballegaard Nielsen, and Steve Riddle, editors, Proceedings of the 10th Overture Workshop, number CS-TR-1345 in Technical Report Series, pages 30–37. Computing Science, Newcastle University, August 2012.

In addition to the above listed publications, a new book is being prepared as a follow-up to the DESTECS project. The author of this PhD thesis will also contribute as co-author of chapters based on tool usage and the semantics of the co-simulation that forms part of this PhD project. Finally, the author of this PhD thesis has been co-author of [P128, P155], but these publications are not related to the research conducted in this PhD project.

1.9 Outline and Reading Guide

This thesis is structured in two parts. Part I, gives an overview of the contributions based on a selection of the publications carried out as part of this PhD project. Contributions are numbered e.g. [C1], and framed. Part II, contains a selected subset of the actual publications that is the base of the contributions; all publications in this part start with a description about where the paper was published followed by the paper as published.

The publications introduced in Part I all fall within the topic of *"Enhancing Formal Modelling Tool Support with Increased Automation"*. The purpose of this part is to give an overview of the publications while introducing relevant background material and related work. Part I introduces a total of 15 publications. To make it possible to distinguish these publications from other references, they are prefixed with "P" e.g. [P89].

Part I is structured as follows; after the introduction in this chapter, Chapter 2 presents the publications: [P74, P75, P108, P89, P83, P87, P107, P79, P90, P125] all concentrating on *Tool Automation*. The chapter starts by giving some background about the context in which the work has been carried out, followed by a description of the work related to tool automation. The contributions described in this chapter primarily focuses on tool aspects of validation. This includes both interpretation of discrete-event systems and reactive control systems that require high-fidelity representation of the environment.

Chapter 3 introduces the *Semantics* work that the tool automation is based upon with focus on the new semantics for VDM-RT and a co-simulation framework. This chapter is based on the publications: [P91, P85, P86, P84, P18]. Finally, Chapter 4 concludes and discusses the contributions[12] from Chapter 2 and 3, and how they relate to each other and fulfil the objectives from Section 1.6.

Part II lists a selection of papers that is written by the author of this PhD thesis or in collaboration with others. Each chapter presents one or two publications and starts by listing the bibliography entry for the publication used throughout this thesis followed by the publication in its original published form.

[12] The electronic version of this thesis uses hyper-links to link each contribution reference to its definition.

2

Tool Automation

This chapter explains how the Overture tool has developed over time and how this PhD project has been involved in the development of many of the current features and how it influenced the development of the internal architecture. The current status of the Overture features is shown in Figure 2.1 which groups the features into four groups: (1) The *development* group includes basic features needed for a Integrated Development Environment (IDE), (2) The *validation* group includes different feature all related to interpretation, (3) The *translation* group includes different translators for other modelling or programming languages, (4) The *verification* group includes features for model checking or proving properties about VDM models. The work carried out during this PhD thesis is mainly focused on the three first groups where work has been done in the development group to create a framework to support the research carried out in the verification and translation groups.

2.1 Overture in a Historical Perspective

The Overture open source initiative was started in 2003 by the authors of [33]. From the beginning, the mission of the Overture project was defined as:

- To provide an industrial-strength tool that supports the use of precise abstract models in any VDM dialect for software development.
- To foster an environment that allows researchers and other interested parties to experiment with modifications and extensions to the tool and the different VDM dialects.

In the first years of the Overture project, work was solely performed by MSc students starting with [142, 143] with a focus on using XML as the internal data structure. This was followed by [109] where the Abstract Syntax Tree (AST) was manually implemented resulting in numerous bugs. As a reaction

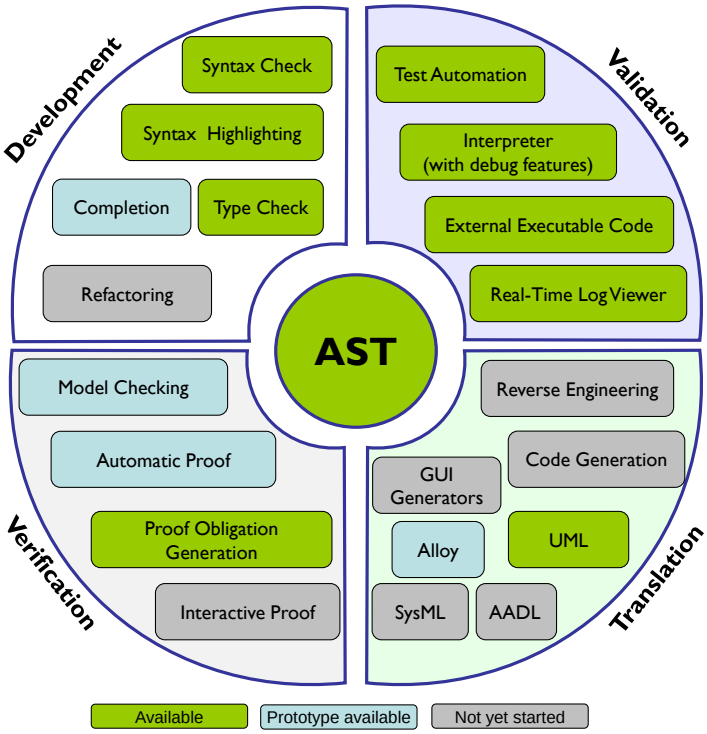


Figure 2.1: Overture Overview

to this Marcel Verhoef suggested a way to automate the classes for the underlying AST by producing a tool called `ASTGen`. A key feature of `ASTGen` was the ability to automatically generate both the java code implementing the different nodes in the AST. This is to be used inside the Eclipse integrated development environment for the implementation of the Overture tool and in addition a possibility to generate the corresponding AST at a VDM level.

The AST representation at the VDM level was desirable in order to enable development of core components using VDM itself in the same kind of bootstrapping fashion that was successfully applied in the development of VDM-Tools [73]. This approach has been used in the following MSc projects: static semantics [16], proof support using HOL [149], connection to JML [151], test automation [131] and coupling to UML2.0 [88].

In parallel with these efforts a stand-alone command-based tool called VDMJ was developed by Nick Battle [8]. An attempt to build a common

Overture front-end using Eclipse was then made in [104]. This naturally also meant that basic conversions between the AST generated using `ASTGen` and the AST inside VDMJ was needed.

The Eclipse front-end was then subsequently significantly improved and extended with full interpretation and debug capability using VDMJ. Furthermore, both the Eclipse front-end and VDMJ were updated to enable the DESTECS project to use Overture as its platform.

During 2012 the development of Overture struggled with an ever increasing effort required to maintain the multiple ASTs used by the various features. Therefore, a new attempt was made to create a new common AST that could meet the analysis requirements of all existing features of Overture but at the same time make it flexible and extensible to prepare for new features. It was decided to create a tool that could generate the AST and visitors supporting any analysis code needed.

The new tool `AST-Creator`, inspired by `ASTGen`, was developed enabling the automatic generation of a new common Overture AST based on the VDMJ internal structure. This new tool was developed to create a new AST for Overture with improved analysis features for e.g. translators. This included a new feature with ability to extend the Overture AST in a way that allowed reuse of all existing features. The Overture front-end and VDMJ were converted to use the new AST in the period between summer 2012 to spring 2013. As a part of this PhD project, a version of the coupling to UML [P89] and a new translator for VDM to Alloy [P83] were developed using the new analysis feature after the successful conversion of Overture.

Pre-dating this PhD thesis and the DESTECS project, we published a vision paper about the Overture tools for VDM [P74] describing the then current status and further plans for the Overture community. That then later was followed-up by a number of tool automation papers [P78, P75, P108] published during this PhD project to make others aware of the active research on tools for VDM.

2.2 Development

The Overture platform was chosen as the basic research platform for this PhD project. Before this PhD project, a number of features were developed independently of each other, and as a result, a number of different ASTs for the VDM language were used internally. During this PhD project, features were upgraded to make use of the same AST, and thus allowing a better integration; this resulted in a modern IDE that both includes editing capabilities as well as

different kinds of analysis for the VDM language. Most of the work carried out during this development did not lead to any direct publications; the most important contribution was the influence on the design of the internal AST for which a new language and generator was developed. This new generator was used to solve the issue with multiple ASTs [82], and at the same time make it easier to implement various kinds of analysis e.g. type check, translations for UML and Alloy. The outcome of this development effort have been a new IDE that is designed with extensibility in mind that was able to serve as a platform for the rest of the work described in this chapter.

2.3 Validation

Validation of specifications can be carried out in a number of different ways e.g. manual inspection, proof of properties of the system or execution. This section describes various contributions related to the execution of VDM specifications. It can be useful for the user to demonstrate the behaviour of a specification before implementation costs are accumulated [43, 72, 46]. This gives the user hands-on experience early in the specification process. In [48] they suggest that this leads to over-specification of systems, and that only system properties should be recorded so that one does not exclude any possible valid implementations. However, others [43, 72] argue that only minimal design decisions are needed to make specifications executable. This makes validation through execution a cost-effective supplement to e.g. proofs [46].

The VDM language is generally non-executable but it contains a subset that can be executed [77], earlier work has also been carried out to extend the executable subset with implicit function and operations [42]. In this section we present a way to interpret VDM specifications, improve test generation through automation and how to improve analysis and the representation of embedded systems using VDM-RT. We also show how a specification can be graphically visualized and integrated with existing code.

2.3.1 Interpretation

Simulation of a system is an easy way to gain knowledge about design options before any formal analysis is carried out. One way to efficiently find problems with a formal model is to evaluate expressions using the definitions of the model. In the event that such expressions do not yield the expected values, it is essential to be able to deterministically reproduce the problem, for example by debugging the model using a deterministic interpreter. The VDM language

contains looseness that allows the modeller to gain abstraction and to avoid implementation bias; to perform simulation, tool support must provide an interpreter and debugging environment that allows the designer to investigate and reproduce problems; as a result the interpreter must be able to deal with looseness deterministically both during interpretation and debugging, otherwise it may not be possible to reproduce problems.

In [P87] we presented a deterministic interpreter and debugger that supports all VDM dialects. The interpreter is based on the semantics given in [146, 52, 145] and is able to interpret the executable subset of the different VDM dialects including looseness [77] and non-determinism. A specification that contains looseness may have many semantic models but validation through interpretation only selects a single model that can be executed at the time. As a result, only a single semantic model is executed leaving the rest unexplored.

Validation of specifications is not a new approach but has been done by others like VDMTools [77] from which this work was mainly inspired. The ProB tool [95] for the B language focuses on execution of implicit definitions. The ProB tool has also been used for Z specifications [118]; none of the other interpreters/translators to programming languages of the Z notation [23, 141, 11, 132] include the notion of concurrency as presented in our work. The POOSL approach has significant similarities with VDM-RT and as opposed to the other tools for B and Z includes the ability to describe distributed systems [140, 38]; they resolved non-determinism by always selecting the first option when a finite collection of possibilities exist but they do not have the interactive debugging functionality as described in our work. Our work differs from previous work by mainly focusing on the deterministic execution/debugging of distributed and concurrent systems, and the support for run-time termination check of recursive functions.

This section is outlined with a description of the sequential interpretation of VDM in Section 2.3.1.1, followed by the concurrent interpretation in Section 2.3.1.2 and finished with Section 2.3.1.3 about combining the interpreter with external executable code.

2.3.1.1 Interpreting Sequential Models

The interpretation of sequential specifications starts out by converting the definitions from the model into their semantic equivalent; this is achieved by a tree traversal. The VDM language was not designed for interpretation and therefore the process of this transformation can be complex because of the potential dependencies between definitions. The interpreter operates with specific values and not symbolic values, and therefore the initialization

of a specification amounts to the evaluation of the state, either in VDM-SL state definitions or in VDM++ or VDM-RT static class definitions. The initialization is guided by the definition dependency ordering and deterministically initializes definitions that do not have an explicit ordering; as a result a re-initialization always produces the same initial state.

The execution of a function or operation in the interpreter is in the simplest case carried out by evaluation of the argument expressions followed by the body of the function or operation. However, the interpreter is also capable of checking pre- and post-conditions, type and state invariants, recursive measures and perform general runtime type checking during execution. The complete execution order is as follows:

1. Type check function arguments
2. Check function pre-condition
3. Check measure if defined
4. Evaluate function body
5. Type check function result
6. Check function post-condition

The new addition here is the measure check that checks for termination in case of a recursive function. The measure check uses a special measure function [124] that must be defined; this function takes the same arguments as the recursive function it is a measure of but must have a return type of $\mathbb{N}$. The measure must be defined in terms of its arguments so that it is strictly decreasing for every nested call in the recursion function[1].

Looseness in VDM can be illustrated by e.g. the let-be expression, the expression represents a choice as shown in **let** `a` **in set** {`1, 2`} **in** `a` where this expression denotes either 1 or 2. This is deliberately left as an implementation choice from a refinement perspective. To interpret such models the interpreter chooses one of the possible semantic models. In order for the interpreter to produce executions, it must choose one of the possible semantic models in order to produce a deterministic interpretation. The iterations over a set of elements poses a similar challenge where the order of the set must be the same for every execution. As a result, this interpreter can deterministically execute sequential models containing looseness and therefore it is easy to

[1] Note there are still cases where it is impossible to express a measure functions i.e. directly over tree structures

reproduce any problems discovered during execution for later inspection. The interpreter only selects a single semantic model to execute for performance reasons; it is theoretically possible to calculate all semantic models from an executable specification as described in [71].

Contribution 1. Run-time termination checks of recursive functions using measure functions.

2.3.1.2 Interpreting Concurrent Real-Time models

Interpretation of concurrent and concurrent real-time systems is in itself not new, and has been done in e.g. LOTOS [55] with CSP [51] and CSS [102] and TCOZ [97] for Timed CSP [123]. However, the VDM language differs from CSP and Timed CSP by using a state based approach alternative to the algebraic approach used in CSP. The approach taken in TCOZ is to combine Timed CSP with a state based approach (Object Z). This is to some extent similar to VDM-RT which also includes timing primitives and has support for modelling of multi-threaded concurrency.

The sequential interpretation presented until now only reflects execution of VDM-SL or non-threaded VDM++ specifications and thus results in a single threaded evaluation that always produces the same result because the interpreter treats all loose operations as under-determined rather than non-deterministic [77]. Under-determined denotes the set of all possible deterministic implementations whereas a non-deterministic approach denotes all possible implementations including non-deterministic implementations.

To simulate specifications of real-time systems that include concurrency and distribution the initialization process must be extended with deployment of objects to CPUs and configuration of the system topology that the interpreter must use during the evaluation. The VDM-RT dialect supports a more precise simulation of how a system behaves, not only with respect to concurrency but also how the interpreted execution time influences the model. The VDM-RT dialect is defined in terms of computational resources such as CPUs and busses which each can have maximum a single running thread at a time. The interpreter must use this information and that of the special **`system`** class to determine whether interprocess communication over a bus is required.

The VDM scheduling is responsible for scheduling resources and the control of the interpreted execution time, and is controlled by a *Resource-Scheduler* that is based on *Resources* as shown in Figure 2.2. A resource is

represented by a separate limited resource in the model, such as a CPU or a bus; A CPU is limited to only run one thread at a time while busses are limited to only communicate one message at a time, this creates a queue of activities per resource. The resource scheduler is responsible for scheduling the execution of resources.

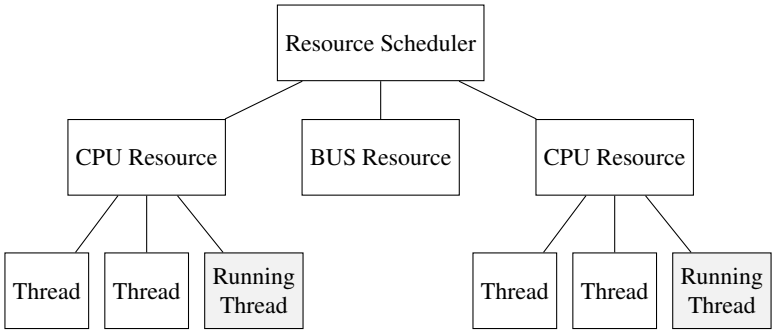


Figure 2.2: Overview of the VDM Resource Scheduler.

To use the same scheduler for simulation of VDM-SL and VDM++ specifications that exclude the notion of time, a special CPU resource (called a virtual CPU) is used and no bus resources are available. The virtual CPU has an infinitely fast simulation time and therefore does not put any timing constraints on the evaluation of the models. Every resource has a scheduling policy which is responsible for determining which thread is the best to run next, and for how long it should be allowed to run (its time-slices).

With VDM-RT, in the event that the active thread is trying to move interpreted execution time, the resource will identify this fact. The resource scheduler is responsible for waiting until *all* resources are in this state, and then finding the minimum time step that would satisfy at least one of the waiting threads. The interpreted execution time is moved forward at this point, and those threads that have their time step satisfied are permitted to continue, while those that need to wait longer remain suspended. This reflects the semantics of VDM-RT as defined in [145, 52] and Chapter 3.

A VDM extract of this behaviour is specified in Listing 2.1. The initial loop establishes whether any resources can progress. The `CanProgress` operation does a compute step for the resource, if possible. The `progressing` flag will be **`true`** if any resource was able to progress. The `getTimestep` operation either returns the timestep requested by the resource, or **`nil`**, indicating that it is not currently waiting for time to advance. If no resource

can progress and no resource is waiting for a timestep, the system is *deadlocked*. Otherwise, if no resource can progress and at least one is waiting for a timestep, then system time can advance by the *smallest* requested amount. In this event, every resource is adjusted by the minimum step, which will result in at least one resource being able to progress. This scheduling process continues until the original expression supplied by the user completes its evaluation. Note that the set `resources` is interpreted here according to the looseness defined in [77] posing a deterministic ordering on the resources.

```
progressing := false;
for all resource in set resources do
  -- record if at least one resource is able to progress
  progressing := CanProgress(resource) or progressing;
let timesteps =
  {resource.getTimestep() | resource in set resources}\{nil}
in
  -- nobody can progress and nobody is waiting for time
  if not progressing and timesteps = {}
  then error -- deadlock is detected
  -- nobody can progress and somebody is waiting for time
  elseif not progressing and timesteps <> {}
  then let mintime = Min(timesteps)
    in
      (SystemClock.advance(mintime);
      for all resource in set resources do
        AdvanceTime(resource,mintime))
  else -- continue scheduling
```

Listing 2.1: Overview of the internal scheduling sequence.

Debugging multi-threaded programs is another challenge not solved by simply using a deterministic interpreter. Non-deterministic behaviour can easily be introduced if a debugger suspends a single thread in a multiple threaded program. This causes non-deterministic scheduling since the suspended thread is excluded from scheduling, thus decreasing the number of threads in the scheduling algorithm. This introduces non-deterministic behaviour during debug which is difficult to repeat, however, this approach is common and both used by the Java debugger in Eclipse and the C# debugger in Visual Studio.

The solution chosen for the VDM interpreter suspends all threads when a breakpoint is reached; this does not in itself ensure deterministic debugging

but the resource scheduler described above has the overall control and thus controls that the overall scheduling order is preserved.

Contribution 2. Deterministic interpretation and debugging of distributed and concurrent systems using a deterministic scheduler.

2.3.1.3 Combining VDM with Executable Code

Formal methods are generally difficult to comprehend for people who are not already used to them and this poses a challenge when the formal specification must be compared to the users expectations. Furthermore, links between a specification and external subsystems that are not modelled can be unclear.

In [P107] we present work that allows one to build an interactive graphical representation of an executable specification, including the ability to integrate with external code. The work is inspired by previous work in VDMTools [35] that supported a CORBA link [138] and dynamic loading of DLLs [139]. However, both these approaches were challenging to use compared to our solution. The main difference is that we do not require any syntactic changes to the language. The interpretation of the specification is kept consistent by run-time checks of all calls to external code; including checks for pre/post-conditions, invariants and run-time type checking. The integration of external code still allows the user to debug the specification.

The approach for Coloured Petri Nets: Comms/CPN [44] is a standard ML library that augments e.g. DESIGN/CPN with the necessary infrastructure to establish communication between a CPN model and external processes. The Comms/CPN library enables two-way communication between the CPN model and the external process using TCP/IP, by defining generic send and receive-functions which accept a byte stream of data. Encoding/decoding functions have to be implemented to marshal data for transmission. Comms/CPN has the advantage of using TCP/IP which allows heterogeneous clients to interact with the simulator, while the send/receive approach has the weakness of potentially blocking the simulator while waiting for data transmission and it requires the external process to implement some conveying, and mapping of the received data into concrete functionality e.g. an update of a graphical animation. Comms/CPN differs from our solution since they need to define the connection points used for the TCP/IP connections.

The successor of DESIGN/CPN is Access/CPN [154] which provides a Java interface to the CPN simulator and thus provides a simpler higher level interface. Communication is still done through TCP/IP and this still requires

the definition of connection points. Another approach for Event-B, is B-Motion Studio [67] that makes it possible to create visualizations via a visual editor, and it establishes a link to the model using Event-B expressions as gluing code. The key feature is that it allows a for faster creation of graphical representations without the knowledge of graphical programming. Its focus on easy visualization comes at the cost of flexibility, as users who might want to do advanced features, in particular parts of the visualisation, will lack the versatility provided by access to lower-lever graphical programming. To our knowledge there is no possibility of interacting with the model or visualisation through external executable code. The easy visualization provided by B-Motion Studio would be beneficial for VDM; however, it cannot be directly mapped to VDM because of the dynamic construction of objects allowed in VDM. This dynamic construction of objects is handled in our solution, and a solution is provided that integrates with a graphical user interface generator [111, 110] that in-part resembles B-Motion Studio. Additionally, our solution also has an integration with external legacy code, which is not the focus of B-Motion Studio.

Contribution 3. A foundation for integration of graphical user interfaces with an interpreter for run-time manipulation of a specifications.

Contribution 4. Integration of external code with the interpretation of specifications without changes to the syntax of the formal language, while preserving the internal run-time checks.

2.3.2 Automatic Testing of VDM Specifications

Testing can be used to gain confidence in a specification, but to obtain sufficient coverage of even small specifications a significant effort is required by the user. A solution is to use automatic testing that can generate test cases which then can be checked against the specification. However, this limits testing to only consider run-time errors; a test oracle would also be required to test for defects not resulting in run-time errors.

In [P79] we introduce a combinatorial testing tool that is based on earlier work conducted with the TOBIAS tool by the research group led by Yves Ledru [113, 94, 92, 93], and related to recent work conducted on the

generation of test cases from model checkers [5, 112, 126, 39]. Combinatorial testing in VDM involves the automatic generation and execution of large collections of test cases derived from templates provided in the from of trace definitions added to the VDM language. This approach uses regular expressions to describe possible sequences of operation calls and utilises the run-time checks for VDM to determine if a single test sequence yields a valid result. The results are groups into three categories:

failed – if a run-time error occur during the execution of a test sequence, or

inconclusive – if the test contains a call to an operation which pre-condition does not hold, which indicates that the generated test was not a valid test for the specification at hand;

passed – this indicates that no run-time errors occurred during execution and that the test is correct in relation to the specification.

The number of tests generated though automatic test generation quickly reaches thousands or even millions. Therefore, it is favourable to reduce the test set while keeping them as representative as possible. Two reduction techniques can be applied: (1) A random selection technique, or (2) a shape reduction technique. The shape reduction technique is divided into three types of reductions where the simplest type defines a shape as a sequence of named operation calls, regardless of their argument values. By guaranteeing to retain at least one example of every test shape, the reduced set of tests can claim to be more representative than e.g. a random selection. The two other shape reduction techniques is based on the injection of explicit variable assignments into test sequences. This enables the shape reduction to take variables into consideration in the analysis, either simply by their name and position in the test sequence, or by both considering their name and assigned value. The effect of using the latter two techniques is finer grained shapes that therefore limits the degree of reduction. The following example from [P79] illustrates the syntax, and what effect the reduction techniques have on the number of tests generated:

```
T: let {x,y,-} in set {{1,2,3},{2,3,4}} in
    (op1(x, y) | op2(x + y)) | op1(1, 2)
```

This trace expands to 25 tests: each subset produces six pair-matches for `x` and `y`, giving 12 pairs; each pair produces a call to `op1` and `op2`, giving 24 tests; lastly, there is one call to `op1` on the end, giving a total of 25.

Random reduction: If we ask for a test reduction of 0.01 (i.e. 1% of the original 25), then using a random reduction technique, we would select one test at random – the reduction will never select less than one test.

Simple shape reduction: If we ask for a simple shaped reduction of 1%, we select two tests at random: one is a call to `op1`, and one is a call to `op2`. This is because these are the only two shapes in the set of 25, and the reduction guarantees to retain at least one test of each shape.

Shape reduction with variable names: If we ask for shaped reduction of 1% with variable *names*, we get three tests: two are as with simple shaped reduction, noting that `x` and `y` are set, and the third is the `op1(1, 2)` call which does not involve any variable settings and therefore is now regarded as a different shape.

Shape reduction with variable values: Lastly, if we ask for shaped reduction of 1% with variable *values*, we get 21 tests: the only tests missing from the original 25 are those which were duplicated because of the presence of `2` and `3` in *both* subsets – i.e. there are two ways for the variables to be set to `2` and `3`, and each of these produces two tests because of the `op1`/`op2` alternative, so four duplicates are missing from the total, giving 21 tests.

Note that as the specification of shapes becomes more detailed, it is not possible to achieve the requested 1% reduction in the number of tests. This is a natural consequence of the reduction process retaining at least one test of each shape.

It is worthwhile noting that we extended the combinatorial testing in [P79] to also incorporate the execution of operations that start concurrent threads in VDM++ and VDM-RT so that possible deadlock situations can be detected. However, to fully exploit this we need to extend the syntax to include the non-deterministic VDM statement, which in the interpreter simply executes the statements it contains in a non-deterministic order. In a combinatorial testing context, this construct would expand to all possible orderings of the operation calls included.

Many different approaches to test automation based on formal methods exist [24, 45, 114, 49, 144] and our work on combinatorial testing, is based on the work of Ledru [113, 94, 92]. The main difference in our work is the fact that we have extended the VDM dialects with syntax for trace definitions rather than using an external tool. In addition, we have devoted significant

effort to enable the user to work efficiently with combinatorial testing by enhancing the number of test cases that can be handled and by creating a user interface that enables efficient access to the defects discovered.

In earlier work, the performance of combinatorial testing techniques has been criticized [17]. However, in that work test sequencing was not addressed at all, so they could not bring the system under test into a particular state as we can do with the approach presented here. In that respect, from a test sequencing perspective, combinatorial testing is more closely related to model checking [5, 112, 126, 39]. However, model checkers are typically limited by the total number of classes in the system, whereas here we can directly select test scenarios of interest.

Contribution 5. Integration of combinatorial testing in the VDM language, including: a) a new test reduction techniques using variable bindings to define new shape reduction types b) a new construct to express parallelism.

2.3.3 Using The Real-Time Extensions

The real-time extension in VDM aims to give a user the ability to analyse how the timing behaviour of the specification changes under different deployment configurations. This information can then be used to assist a user in answering some of the common design questions relating to deployment configurations, which may include the following [145]:

(a) Does the proposed architecture meet the performance requirements of all applications?

(b) How robust is the chosen architecture with respect to changes in the application or architecture parameters?

(c) Is it possible to replace components by cheaper, less powerful equivalents to save cost while maintaining the required performance targets?

The interpreter described in Section 2.3.1.2 is able to simulate real-time systems that include distribution but lacks a way to check *time constraints*, and thus leaves the user with no more knowledge than for a regular non-distributed system. Therefore, the analysis of real-time systems must include

timing requirements; a real-time system can only be considered correct if the system reacts with the correct behaviour within a certain time frame.

In previous work [30], a number of *validation conjectures* were introduced to describe the temporal relationship between system-level events that can be observed in an execution trace from a simulation of a VDM-RT model using VDMTools. An example of a conjecture for a car navigation specification is shown in Listing 2.2; it requires that a volume change must be reflected in the display of the navigation system within 35 ms.

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

Listing 2.2: Time Constrain Example.

The approach taken in [30] using post-analysis of an execution trace limits the analysis to small examples with simple conjectures due to the amount of data required for their evaluation. Validation conjectures are specified using trigger events associated with operations and expressions over instance variables, and may require a *deadline* to be met, a *separation* or *required separation* between calls. A trigger event indicates that the conjecture must be checked when the event occurs. Events can be one of the following: *request*, *activate* or *finished*; which indicates different stages of the evaluation of operations. The consequence of being able to express conjectures based on instance variables is that it requires all values of instance variables to be stored every time an event included in a trigger occurs during interpretation of a specification.

In [P125] we show how conjectures effectively can be checked during simulation by extending the VDM interpreter. This solution only logs the information required to visualize the execution trace, annotated with markers showing where conjectures were violated as illustrated in Figure 2.3. As a result, our solution is more scalable, and it is a good starting point for exploration of alternative architectures. This solution can easily be extended to raise a run-time error when it is detected that the specification does not respect a conjecture and thus mark the system configuration as not acceptable.

The solution presented so far gives the user the ability to access certain aspects of the design questions presented above. However, the process of exploring alternative system architectures is a manual process, even-though the analysis of each architecture can be automatically evaluated using con-

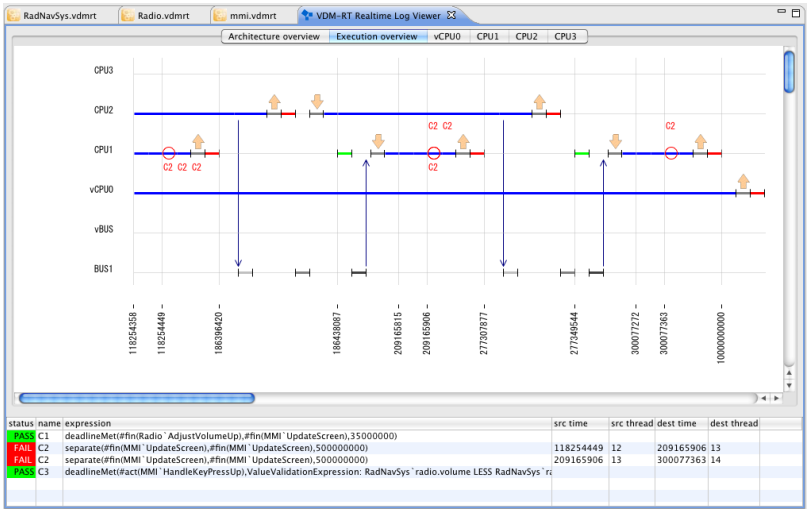


Figure 2.3: Visualization of an execution trace from a specification of a *in-car navigation and radio system* .

jectures. The main issue is the design of the VDM **`system`** class which combines the dependency between deployable instances with the architecture. As a result, the user must manually replace this part of the specification for each architecture that must be explored.

In [P90] we describe how automated exploration of alternative system architectures can be achieved by changing the way VDM defines deployment. The approach taken is to separate the deployable instances (artifacts) from the actual deployment and in the same manner separate the constraints, (the speed of CPUs and capacity of busses), of an architecture from its topology. This then allows automatic generation of hardware topologies and deployments of artifacts onto the architecture taking into account the dependencies between the artifacts. This enables a user to quickly generate different architectures and automatically use the conjectures to judge if an architecture is a valid candidate[2]. Figure 2.4 illustrates how the two extensions presented here extend the functionality of the interpreter.

[2] Note that this does not rank the architectures nor does it estimate cost.

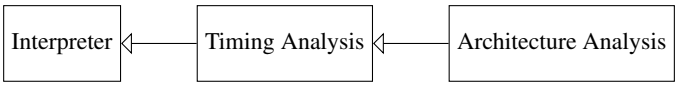


Figure 2.4: Illustration of how the two VDM analysis features conceptionally extend the interpreter.

Contribution 6. Run-time check of VDM-RT timing constraints, expressed as validation conjectures during interpretation.

Contribution 7. Automatic exploration of VDM-RT architectures.

2.3.4 Co-Simulation Extension

Modelling and analysis of real-time systems using simulation is supported by the VDM-RT dialect. This makes it possible to model embedded controllers which operates as reactive systems. However, if the reactive system needs to control movement of e.g. mechanical devices, controlled by physical laws then only a rough approximation can be described in VDM-RT because of its mathematical basis. While this is sufficient for a high level specification it is insufficient for the fine tuning of control loops etc. A solution is to provide a higher-fidelity model of the environment that can be used during the simulation. Conceptionally this can be done in two ways: either by creating a new language that has the ability to express both discrete-event controllers and continuous-time physics, or by using multiple languages joined in a co-simulation.

There are a number of projects that focus on simulation and co-simulation. Ptolemy II [22] offers a heterogeneous simulation framework, where each hierarchical diagram has its own model of computation. The project provides a single tool where multiple domains can be combined and is implemented by using a hierarchical model structure in which a different model of computation can be used (indicated by a so-called director) on each level of the hierarchy. Such an integrated solution is useful, but currently it uses graphical modelling symbols which deviate too much from commonly used symbols. Furthermore, Ptolemy II, being a generic tool, does not always meet the necessary domain-specific requirements. Modelica [40, 15] is an object-oriented, equation-based multi-domain language for simulating controlled

physical systems; it provides a number of libraries of physical components. A number of tools are available that can translate the Modelica language into a format that can be simulated; one of which is jModelica[3] it provides a compiler from Modelica to C which then is compiled into DLLs. These DLLs integrate with Python that enables user interaction. Since the Modelica specification is compiled into executable C code debugging becomes difficult compared to our solution which interprets the specification. Cosimate[4] is a backplane co-simulation tool offering interfaces to tools like Simulink, Modelsim, Modelica etc. The tool only support time synchronization with exchange of data between simulators every time step. Furthermore, Cosimate has been tried out on the control of a mechatronic test setup [47]. The connection between the two models involved was rather cumbersome.

The solution used in the DESTECS project[5] is to use co-simulation instead of creating a new language that combines languages of multiple domains. One advantage of this approach is that the individual members of a development team only needs to master a single discipline, and not e.g. both physics/control engineering and software engineering; however, the team still requires a combined knowledge of both disciplines. Furthermore, special modelling tools for each domain would no longer be possible to use if a new combined language was created.

The co-simulation for VDM enabled the use of a higher-fidelity environment model by allowing a VDM-RT specification of a controller to be co-simulated with a continuous-time model of a physical system. Each model can be written in its own notation using standard tools already available. To achieve this a semantics for a generic co-simulation framework has been developed, including an extension to the VDM-RT semantics; this allows a VDM specification to be used with the co-simulation framework presented in Chapter 3. The interpreter has likewise been extended to meet the requirements stated by the co-simulation framework which primarily consists of a protocol layer.

Figure 2.5, illustrates how the co-simulation extension extends the VDM interpreter and how the internal control of the co-simulation framework has been implemented into the simulation *engine* which is the simulation driver.

The *VDM Co-Sim* simulator implements a new $ResourceScheduler$ that is capable of respecting the time steps requested by the *master* simulation

[3] See more about jModelica at `http://www.jmodelica.org/`

[4] `http://www.chiastek.com/products/cosimate.html`

[5] See `www.destecs.org` for more information about the DESTECS project.

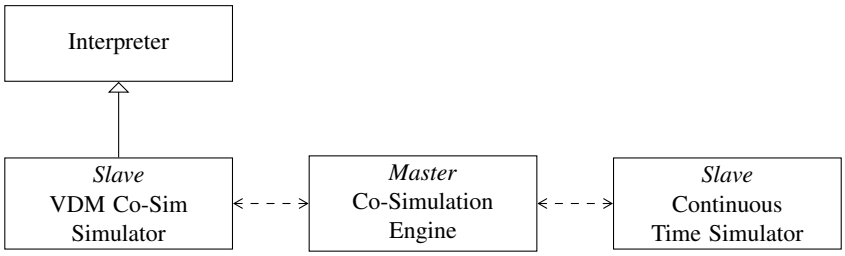


Figure 2.5: Illustration of the co-simulation extension of the VDM interpreter.

engine, and has the ability to commit internal state until a particular time bound when requested upon. It also implements the communication protocol, the functionality to update the internal state of the interpreter from a shared co-simulation state as well as updating the co-simulation state from the internal state at the completion of a simulation step. The simulator must also be able to handle events created during simulation. This is handled by the creation of new asynchronous threads during a step, just before the resource scheduler continues. This allows the newly created threads to be scheduled under the same conditions as the existing threads within the resources of the system. Furthermore, the simulator has been optimized, reducing the number of synchronizations with the *master*; this was achieved by skipping communication to the *master* in the cases where no shared state was either read or changed during a simulation step[6]. In addition to the extension of the interpreter the co-simulation implementation included various other utility features (e.g. functionality to statically check if both the VDM and continuous-time model meet the requirements stated by the co-simulation framework). The tools have been integrated into the Eclipse framework and used in the DESTECS project by both the researchers and the industrial partners that each used the tool to model a specific challenge.

Contribution 8. A co-simulation extension to the VDM interpreter enabling it to integrate into a co-simulation framework.

Contribution 9. A generic tool for co-simulation of DE and CT specifications i.e. the DESTECS tool.

[6] See about the optimization in Section 3.3.3.

2.4 Translation

The tool support for a single modelling language is typically limited to a particular type of analysis, since a language often is constructed to allow easy analysis of certain properties. A translation is an easy and efficient way to extend the types of analysis available for a language; if such a translation is simpler than the implementation of the new types of analysis themselves. In this section two automated translations are presented. Section 2.4.1 presents a translation of VDM to and from UML, and Section 2.4.2 presents a translation from VDM to Alloy.

2.4.1 A UML Translation for VDM

A translation between formal languages and Unified Modelling Language (UML), can be seen as an example, that has been attempted several times in the past (e.g. [65, 133, 21, 68, 29, 137]), and the concept of combining formal and informal languages to exploit the best of both worlds has been investigated by others before [65, 133, 21, 68, 29, 137, 54]. Common to these is the mapping between a formal method (Alloy, B, Z, Z++ and VDM++) and the UML class diagrams which provides a static view. Most of these focus on a one-way translation from UML class diagrams to a formal notation or vice versa. However, in [P89], we present a bi-directional translation that enables a user to describe the structure of a specification in UML, and then translate this to VDM and back. Since UML is widely used in industry, a translation to this modelling language extends the group of researchers and software developers that can participate in the development of formal specifications without extensive knowledge thereof. The work in [P89] describes how a VDM++ specification can be translated to and from UML class diagrams using various new language constructs compared to earlier work developed as part of VDMTools [28, 35]; that included a connection between VDM++ and UML class diagrams. Our work also includes a translation between the combinatorial testing feature [131, P79] of VDM that uses traces resembling regular expressions and UML sequence diagrams.

This UML translation feature [P89] focuses on the static structure of VDM specifications using UML class diagrams. The translation was inspired by the VDMTools that incorporated functionality to translate VDM++ models into UML 1.4 class diagrams using the Rational Rose API. Our work upgrades the translation to use the new UML 2.1 standard and integrates

with the commercial tool Enterprise Architect (EA)[7]. This translation uses a number of new mapping rules and thereby utilizes UML constructs to express VDM types, instead of the text strings used in the Rational Rose mapping. A small train example, visualized by EA, is shown in Figure 2.6; this translation uses both a constrained association to express a VDM union type and a N-ary association to express a VDM product type. The translation differs from the VDMTools translator by using UML constructs to model VDM types instead of the use of text strings.

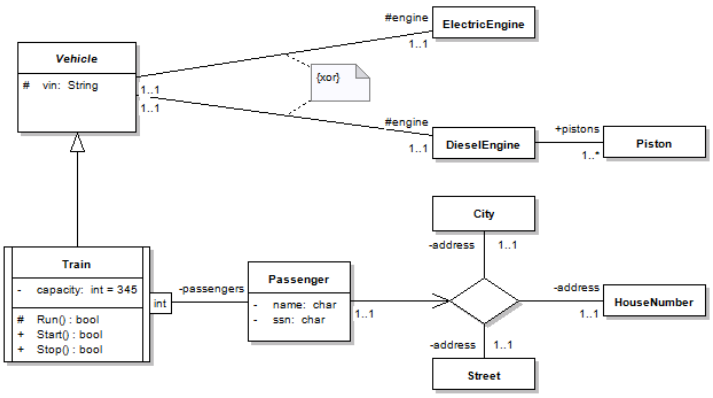


Figure 2.6: Example from the UML paper [P89].

The translation between *trace definitions*, used for combinatorial testing in VDM, and UML sequence diagrams is a new feature that enables a trace definition to be graphically illustrated as a UML sequence diagram. This has the advantage that a user can define trace definitions with only little knowledge about VDM. The translation uses the lifelines, call-events and combined-fragments of UML to express the variables, operation calls, and repeat patterns or choice operators that are used inside a trace definition.

The translation described above has been implemented in Overture and used in a number of teaching classes as well as in the DESTECS research project for about three years; during this period it became clear that the translation needed to be extended to support all language constructs within the VDM++ and VDM-RT dialects.

One of the unsupported constructs both in VDMTools and in our work were union types when used as the return type of functions or operations. The

[7] See `http://www.sparxsystems.com/` to get more information about Enterprise Architect.

translation of types was also limited to associations and not as parameters of functions or operations. Therefore, the translator was upgraded to use a new concept that allowed all VDM constructs to be translated to and from UML while still allowing the mappings defined in [P89].

As a consequence of the deficiencies found with the published UML translator [P89] a new solution was implemented using the Eclipse UML2 framework and thereby making any tools available that support UML2 which includes many free open-source tools, instead of the commercial tool EA. The translation uses the same mapping rules as defined in [P89] but defaults to a number of generic classes for `Set`, `Seq`, `Map`, `Union` and `Product` types where the previous translation was unable to translate the construct.

Experience also indicated that the users were unfamiliar with constrained and N-ary associations that therefore these were excluded from the new translator. However, this is a design choice not due to technical restriction, and could be added for instance variables and values. Figure 2.7 shows an extract of the previous translation from Figure 2.6; it is visualized by the free UML modelling tool *Modelio*[8]; note the use of Set, Union and Product in the `Vehicle` and `Passenger` classes. This translation supports all VDM types and thus enables a complete round-trip for VDM specifications consisting of classes, types, functions and operations. Furthermore, the new translation adds a translation of the VDM **`system`** class into UML deployment diagrams.

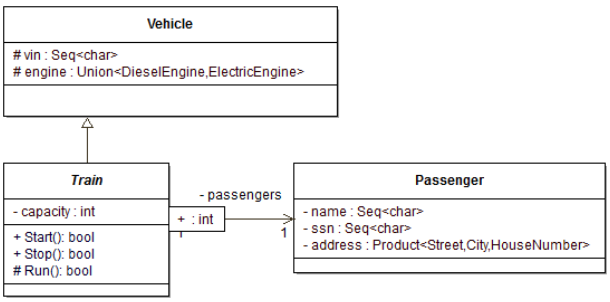


Figure 2.7: Example from the new UML implementation.

[8] See `http://modelio.org` for more information about Modelio.

Contribution 10. A translation between VDM and UML 2.1 including class diagrams that use the internal UML structure to represent VDM constructs, and a translation of UML sequence diagrams to combinatorial testing, test sequences in VDM.

2.4.2 A Translation from VDM to Alloy

In a VDM setting initial software design is often best described using implicit-style specifications in the VDM-SL dialect. However, limited tool support exists to help with the difficult task of validating that such VDM-SL specifications capture their intended meaning. The validation technique presented in Section 2.3, cannot directly be used since implicit specifications generally are non-executable. However, work has been done for a subset of VDM that enables implicit specifications [41, 42] to be interpreted by e.g. rewrite of post-conditions into an explicit form that can be executed using the standard VDM interpreter. A different approach is to use a model-finding technique to find instances of the specification and to check the specification for contradictions; it also enables user specified assertions to be checked against the specification.

The Alloy Analyzer is an automated model-finder for checking and visualizing Alloy specifications that has proved to be useful for validation of specifications [57]. However, to take advantage of the automated analysis of Alloy, a model-oriented VDM-SL specification must be translated into a constraint-based Alloy specification.

Various previous works have used the Alloy Analyzer to visualize or check properties of specifications expressed in different languages. However, to the author's knowledge no such attempt has been made for VDM. The translations from both B and Event-B to Alloy [101, 99] both combined theorem proving with model-checking and thus used Alloy to check properties. It is noted that Alloy does not have standard operations for manipulating ordinary sets that result in unnecessarily long specifications: however, the current edition of Alloy (version 4.2)[9] includes a number of utility modules providing such features which are utilized in this work. The translation of UML with OCL to Alloy [6] identifies differences related to e.g. inheritance, collections and namespace. The latter was also encountered in our work which requires

[9] See `http://alloy.mit.edu/` for more information about Alloy and the Alloy Analyzer

all argument identifiers for functions and predicates to be disjoint from any field names used in signatures. The translation of Z to Alloy [98] defines a semantics preserving translation for a subset of the Z notation that enables automatic syntactical translation to Alloy because of the language similarities. However, this is not possible to the same extent for VDM due to the higher-order constructs but we are also able to give a semantics preserving translation of a subset of the VDM language that can be translated automatically.

In [P83], we describe how to translate a subset of the VDM-SL into Alloy, and how the Alloy Analyzer can be used to find errors in specifications. It is worth noting that the translation supports sequences but that these require a careful selection of the scope used for integers when analysis is performed, a similar problem occurs when using VDM values of complex types. Due to the limitation of the Alloy Analyser, recursion and statements used in operation bodies are not included. However, these are expressible in Alloy.

The translation of VDM to Alloy translates VDM types into Alloy signatures that are similar to VDM types but limited to only expressing relations between atoms, and thus not able to directly express e.g. sets of sets often used in VDM. However, by introducing extra atoms and relations most VDM types can be translated including constraints defined by invariants. In Figure 2.8, an extract of a telephone specification is shown in the VDM notation; the extract shows a state definition $Exchange$ with two fields $status$ and $calls$, declared as a map and an injective map. The state has an invariant associated stating a relation between the two fields. The specification also contains an implicit operation with a read/write frame condition on the $status$ field of the state and a pre- and post-condition. An automatic translation of this extract is shown in Figure 2.9, where the state from the VDM specification is expressed as a signature with two fields resembling the VDM state fields. However, since Alloy uses relations rather than maps and injective maps additional constraints are needed, as shown in the first part of the constraint of the signature. The second part of this constraint represents the state invariant from $Exchange$. The VDM operation is translated into an Alloy predicate which takes two additional arguments representing the state. The predicate conjoins the restrictions on the state derived from the frame-condition with the pre- and post-condition. By default it is required that the two state arguments, representing the pre- and post-state are equal; however, if a write frame condition for a state field exists then the equality constraint for that field is dropped.

$$\begin{array}{l}\textbf{state } Exchange :: status : Subscriber \xrightarrow{m} Status \\ \qquad\qquad\qquad\quad calls : Subscriber \xleftrightarrow{m} Subscriber\end{array}$$

inv

$$(mk\text{-}Exchange(s,c)) \triangleq \forall i \in \textbf{dom}\, c \cdot (s(i) = \text{WI} \wedge s(c(i)) = \text{WR})$$

$$Lift\ (s{:}\ Subscriber)$$

ext wr $status$

pre $s \in \textbf{dom}\,(status \rhd \{\text{FR}\})$

post $status = \overleftarrow{status} \dagger \{s \to \text{AI}\}$

Figure 2.8: VDM extract of a telephone specification.

```
sig Exchange{ status: Subscriber →lone Status,
              calls: Subscriber lone →lone Subscriber
}{ functional[status,Exchange] and
   injective[calls,Exchange] and functional[calls,Exchange] and
   all i : dom[calls] | ( status[i] = WI and status[calls[i]]= WR) }

pred Lift(e : Exchange, e' : Exchange, s: Subscriber)
{ e'·calls = e·calls
  s in dom[e·status :>FR]
  e'·status = e·status ++ s →AI}
```

Figure 2.9: Alloy extract of a telephone specification.

Contribution 11. A translation of the higher-order model-oriented language VDM to Alloy, that includes higher-order relations.

Contribution 12. A new approach to checking and evaluating implicit VDM specifications based on Alloy.

2.5 Formal Verification

Different approaches may be used to prove that certain properties hold for a specification using formal methods and thereby show that the system is correct in relation to its specification. Two common approaches are: (1) Proofs

that can be used to show that a system is correct in relation to a number of properties one wishes should hold for the system. These properties are described as *validation conjectures* in [32] and are denoted as logical expressions which can be used in proofs. (2) Model Checking is a technique for automated verifying the correctness of a finite state system according to a number of properties. This is done by a systematic exhaustive exploration of the state space of the model, checking that the model meets the specification that consists of properties one wishes to hold for the system.

2.5.1 Using Proofs

Prior to this PhD thesis a translation of a subset of the VDM-SL language to HOL [149, 150] was defined; it uses HOL to automatically discharge the proof obligations. Therefore, during this PhD, a proof obligation generator [4, 124] has been created and integrated into the Overture platform enabling generation and viewing of proof obligations. The intention was to create a framework that could serve as the base for integration with interactive theorem provers. However, no work has been published by the author of this PhD in this area.

2.5.2 Model Checking

The term model checking refers to a collection of techniques for automatic analysis of reactive systems; these techniques often finds subtle errors missed by conventional simulation and testing techniques. The input to a model checker is a description of the system to be analysed, and a number of properties, often expressed in temporal logic, that are expected to hold. The model checker then either confirms that the properties hold, or that they are violated. In the latter case a counterexample is provided.

The translation presented in Section 2.4.2 makes it possible to use the Alloy Analyzer to check properties for a VDM specification. However, due to the lack of a formal way to capture system properties in VDM such properties, (traditionally expressed as assertions), must be manually written in Alloy; this reduces the automated analysis but still allows one to find valid instances of the system and present them graphically (essentially using symbolic evaluation).

The translation not only allows a specification to be checked for contradictions in pre- and post-conditions but it turns out that a number of published VDM-SL specifications contained contradictions that all were detected

using the Alloy Analyzer. The $Telephone$ example published in [37, 60], and the $Hotel$ included in the *Software Abstractions* book about Alloy [57] are both examples of specifications that were published while containing a contradiction in a post-condition.

In [P83] we also present a new way to use the VDM language to express assertions like that of Alloy. The idea is that VDM expressions could be used to express assertions providing a better alternative to the use of an informal description or logical expression from any language, thereby these expressions could be translated into an assertion in Alloy.

A simple textual description of an assertion for the $Telephone$ example may be described by:

$$Lift \rightarrow Connect \rightarrow ClearWait$$

where $Lift$, $Connect$ and $ClearWait$ are functions or operations in the $Telephone$ example. The description is interpreted to mean that if one calls $Lift$ followed by $Connect$ and $ClearWait$ then the post-condition of each operation should at least ensure that the pre-condition of the following operation holds, finally that the state before the call sequence should be equal to the state after the last call. A manual conservative translation of this into Alloy can be expressed as follows:

```
1  assert liftConnectClearWait{
2    all e,e',e'',e''' : Exchange, s1,s2 : Subscriber |
3      (free[e,s1] and Lift[e,e',s1] and
4      Connect[e',e'',s1,s2] and
5      ClearWait[e'',e''',s1]) implies eq[e,e''']}
```

where *free* and eq are utility functions that respectively set up the initial state and compare the final and first state. We propose to expresses this in VDM using operation quotation [61]. For each function or operation in VDM, two generated boolean functions $pre\text{-}Lift$ and $post\text{-}Lift$ representing the pre- and post-condition, exist. By utilizing this, the above assertion can be written in the VDM notation as follows:

$$\begin{aligned}
&\forall e, e', e'', e''' : Exchange, s1, s2 : Subscriber \cdot \\
&\quad (free(e, s1) \land \\
&\quad pre\text{-}Lift(s1, e) \land post\text{-}Lift(s1, e, e') \land \\
&\quad pre\text{-}Connect(s1, s2, e') \land post\text{-}Connect(s1, s2, e'') \land \\
&\quad pre\text{-}ClearWait(s1, e'') \land post\text{-}ClearWait(s1, e'', e''')) \\
&\Rightarrow \ eq(e, e''')
\end{aligned}$$

the $free$ and eq functions are equal to the one used in Alloy and just represented as simple boolean functions in VDM. The pre- functions take the same arguments as the function they are guarding with the addition of the pre-state. The same applies for the $post$-functions with the addition of the pre- and post-state. The assertion is conservative in the sense that it only has to hold in the case, where all functions denoted by pre- and post- are true, which does not allow cases where a post-condition is false but still implies a valid pre-condition.

Contribution 13. A new way to express system assertions in VDM using the existing expressions.

3

Semantics

This chapter describes the semantic evolution of the VDM language; Section 3.1 starts with a brief reference to the semantics of the VDM-SL standard, followed by references to the extensions for concurrency in VDM++, and time and distribution aspects in VDM-RT which is described in more detail. Finally, Section 3.2 describes the contributions made to the VDM-RT language where a new Structured Operational Semantics (SOS) semantics has been created, and how this integrates with the new co-simulation framework.

3.1 Existing VDM Semantics Efforts

The VDM notation has constantly evolved since the 70's resulting in an ISO standard being defined in the mid 90's. This then served as a basis for new dialects of the language which have been given semantics in, primarily, research projects.

3.1.1 The VDM-SL Semantics

The formal semantics of VDM-SL is included in the VDM-SL ISO standard [70]. It is written in a denotational style based on basic set theory with least fixed point semantics for recursive definitions [81]. The domain universe for VDM-SL has been inspired by [135]; it provides denotations for all values expressible in VDM-SL. The meta-notation used for expressing the formal methods has itself been precisely described but here we refer the reader to [81] for an explanation of these.

3.1.2 The VDM++ Semantics

The formal semantics of VDM++ was first created in the European ESPRIT-III project Afrodite (Project No. 6500) [25, 26, 27]. The intent was that the semantics of constructs from VDM-SL should be unchanged. An ax-

iomatic semantics was provided in [64]. The concurrency aspects of VDM++ was given semantics in [69] using Real-Time Logic (RTL) [103]. However, VDM++ has changed substantially since that time, and as a result the only semantic descriptions that exist are limited to the executable subset of VDM++ that is part of the development of the VDMTools interpreter [35]; unfortunately this document is not publically available.

3.1.3 The VDM-RT Semantics

The formal semantics of VDM-RT is based on the existing formal semantics of VDM++ and has been developed in two stages; first the notion of time was added and secondly distribution was introduced. The notion of time was introduced in the European project VICE (project no. 27618) [105] where the basic idea was to simulate a timing behaviour of the target processor i.e. simulate the clock of the target processes [105]. A dynamic semantics was constructed [20] that extended the VDM++ semantics with the notion of time. The time is modelled in the semantics using durations; a duration is an estimate of how much time a particular portion of a VDM model will take to execute, in the implementation on the target processor. The semantics requires that all expressions and statements of the language have a default duration specified. However, the semantics also includes a new duration statement that is used to override the default durations ignoring any timing information contained within the body of it. Furthermore a constant thread switching overhead is also taken into account.

The VICE approach worked quite well when applied to systems with only a single CPU, but it failed entirely for distributed embedded systems both with respect to the modelling and analysis capabilities of VICE. Therefore a new notion of distribution was introduced by Marcel Verhoef as part of his PhD thesis [145]. A core of the language is given an operational semantics in [145, 52]; using a functional style with natural English as the meta semantics. The semantics includes the definitions from VICE but adds a few new concepts to model distribution as well as the ability to configure the capacity of the distributed nodes (`CPU/BUS`):

1. The ability to construct an explicit system architecture on which functionality can be deployed. For this purpose, the **`system`** construct is introduced in the syntax of VDM-RT and `CPU` and `BUS` classes are provided as first-class language citizens. The capacity, scheduling policy

and task switch (or protocol) overhead of both architectural elements can be specified.

2. A new **`cycles`** statement that adds a time penalty to a statement. The penalty is calculated based on the capacity of the CPU the statement is deployed onto.

The operational semantics was defined in terms of two main transitions: *Execute Instruction* and *Time Step*. An execute instruction step will execute the head of all thread bodies until all threads have a duration at the head of their body. In this case a time step will be performed, incrementing the simulated time by the time of the shortest duration (t_{step}); followed by a decrement of all durations by t_{step}, and removal of all duration zeros. Then execution will continue by again performing a execute instruction step, repeating the cycle.

Any state changes that are a result of computation are not made visible to other threads or resources until the time required for the state change has passed. Then the state change is committed and becomes visible to other threads, as the internal record of time is updated and time-related bookkeeping is dealt with.

The VDM-RT semantic model described in [145, 52] is lacking precision with respect to some of the standard VDM++ constructs. This is a result of only providing a core semantics for the full VDM-RT notation without taking all supported constructs into account. There are cases where the standard behaviour of VDM++ is not appropriate for VDM-RT, and this is described in [P91]. These cases include the following: (1) static access to variables in a distributed setting, (2) static operation calls in a distributed setting, (3) and reading of distributed variables without a bus. The core issue is that the distributed nature of variables and calls would be ignored if the VDM++ semantics were directly adopted. Unfortunately, this is the case in the current implementation of the VDM-RT interpreter [P87].

3.2 The VDM Real-Time Semantics Developed in this PhD Project

The semantics of VDM-RT has been revised in [P85] where we define a new semantics based on the work in [145, 52]. The new semantics uses the SOS format [120, 121] to present the semantic definitions as oppose to the ambiguous natural language used in the earlier version. The logical notation

used in the semantics is the basic VDM-SL type system and expressions. This notation is used to define the static structure of the VDM-RT language.

In a SOS definition, the entire system is modelled as a *configuration* containing all of the information needed to capture the state of a system at any given point. A configuration is typically given as a tuple, in this case of the listed components.

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

The transition relations are defined through the use of inference rule schemata where each rule's conclusion defines a subset of the entire transition relation. The least relation that satisfies all of the inference rules is taken to be the relation defined.

The entities used to describe the VDM-RT semantics form a hierarchy starting with the $VDMRT$ structure (shown below). At the top level, the $VDMRT$ structure records the CPUs in the system ($cpus$), the busses connecting the CPUs ($busses$), the current time that the model has reached ($time$), and the defined classes in the model ($classes$).

$$\begin{aligned} VDMRT :: \quad & cpus : CPUs \\ & busses : Busses \\ & time : Time \\ & classes : Classes \end{aligned}$$

The behaviour is then described using these entities starting with the top-level rule Big Step in Figure 3.1, that gives the whole semantics of a running VDM-RT model. There are six hypotheses of this rule, and each represents a phase in an execution step.

$$\text{Big Step}\ \frac{\begin{array}{ll} vdmrt_1 = commitPendingValuesAndUpdateTime(vdmrt, \tau) & (1) \\ vdmrt_1 \xrightarrow{busses} vdmrt_2 & (2) \\ vdmrt_3 = createPeriodicThreads(vdmrt_2) & (3) \\ vdmrt_4 = doContextSwitches(vdmrt_3) & (4) \\ vdmrt_4 \xrightarrow{exec} (vdmrt_5, \tau_b) & (5) \\ \tau_b' = min(\tau_b, minPendingCommitTime(vdmrt_5)) & (6) \end{array}}{(vdmrt, \tau) \xrightarrow{vdmrt} (vdmrt_5, \tau_b')}$$

Figure 3.1: Definition of the Big Step rule.

1. The first hypothesis is the internal update of the model state. It updates the model's present time, and then commits all of the pending values held in threads up to that point.

2. The second hypothesis is in the form of a transition relation as the action of the busses is inherently non-deterministic. This phase actually delivers messages on the busses to their target CPUs if sufficient time has passed. Where the message is an operation call, a new thread will be created on the target CPU to run the operation.

3. The third hypothesis potentially creates more new threads based on the timing of periodic threads.

4. The fourth hypothesis performs any potential context switches, allowing a CPU to change from one running thread to another. Note that this phase happens after the creation of new threads so that those new threads have the potential to start execution within this step of the execution.

5. The fifth hypothesis is also a transition relation, as the execution of VDM-RT statements may be non-deterministic. This transition attempts to execute the first duration of the body of every active, running thread in the model. The number of threads attempted will be no greater than the number of CPUs in the system, as each CPU may only execute in one thread per step. If it is not possible to fully execute the duration at the head of a thread's body, then a $PartialDuration$ will replace that duration on the head of the body. The partial duration will have the remainder of the original duration's body that remains to be executed, and execution will continue during the next step that the thread is active. This hypothesis also exposes the minimum time until the next commit of pending values.

6. The sixth hypothesis calculates the time at which the next action in the interpreter must happen. This may be due to things such as threads with pending variables, the creation of a new periodic thread, and so forth. This results in the minimum amount of time until the next action that the interpreter can handle, which then define τ for the next application of the Big Step rule.

During an execution step, threads may change the values of instance variables in objects. However, such changes are not committed to the object state immediately: instead they are stored in the $pending$ field of the $Thread$

constructs, hiding the values from other threads until time has progressed sufficiently to cover the time specified by the active $PartialDuration$ of the $Thread$.

The resolution of pending values and durations are handled in the Big Step rule by the first hypothesis, which considers the prior state of the executing model and the time that the model will be updated to. The $vdmrt_1$ object represents the state of the interpreter after the time is set to τ and all pending values are committed for threads currently ready to commit, i.e. those with PENDING status and with remaining elapsed time equal to the time delta between the old and updated times.

Furthermore, all pending threads that have their values committed are returned to RUNNABLE status if they have remaining work to do, and the remaining pending threads have the $elapsed$ time, from the $PartialDuration$ at the head of their $body$, decremented by the time delta between the old and updated times. All threads with empty bodies are checked to ensure they have COMPLETED status and altered if necessary.

The behaviour of value commit and time update is contained in the semantic function shown in Figure 3.2. Note that the post-condition of the function depends on the $mergePending$ function, which is a helper function that merges the pending values of a thread into the object states those values are associated with.

The new semantics presented in [P85] solves the weaknesses identified in [P87] by disallowing e.g. static access to variables, and therefore forcing cross-CPU reads of variables to use bus communication. This does not restrict the expressive power of the language but makes the distribution and share of data or functionality explicit to the user.

Contribution 14. A new SOS semantics for the full VDM-RT notation without the weaknesses identified in [P87].

3.3 Co-Simulation Semantics

Co-simulation is one approach that supports simulation of a system where different notations are used from different domains. In this section we will consider semantic work done to allow a co-simulation with the VDM-RT language being used as a DE simulator. Co-simulation has been attempted by others (e.g. Ptolemy II [22], Modelica [40, 15], Cosimate) as described in Section 2.3.4.

$$
\begin{array}{l}
commitPendingValuesAndUpdateTime\colon VDMRT \times Time \to VDMRT \\
commitPendingValuesAndUpdateTime(vdm, \tau)vdm' == \\
\textbf{pre } \tau \geq vdm.\tau \\
\textbf{post } vdm'.\tau = \tau \ \wedge \\
\quad \forall id_c \in \textbf{dom } vdm.cpus \cdot \\
\quad \forall id_t \in \textbf{dom } vdm.cpus(id_c).threads \cdot \\
\qquad \textbf{let } thr = vdm.cpus(id_c).threads(id_t), \\
\qquad\quad thr' = vdm'.cpus(id_c).threads(id_t)) \ \textbf{in} \\
\qquad (thr.body = [\,] \ \Rightarrow \ thr'.status = \textsc{Completed}) \wedge \\
\qquad (thr.status = \textsc{Pending} \ \Rightarrow \\
\qquad\quad \textbf{let } step_t = \tau - vdm.\tau \ \textbf{in} \\
\qquad\quad \textbf{let } (d_t, d_t') = ((\textbf{hd } thr.body).elapsed, (\textbf{hd } thr'.body).elapsed) \ \textbf{in} \\
\qquad\qquad [d_t = step_t \ \Rightarrow \\
\qquad\qquad\quad (thr'.pending = \{\,\} \wedge \\
\qquad\qquad\quad thr'.body = \textbf{tl } thr.body \wedge \\
\qquad\qquad\quad (thr'.body \neq [\,] \ \Rightarrow \ thr'.status = \textsc{Runnable}) \wedge \\
\qquad\qquad\quad \textbf{let } objects = vdm.cpus(id_c).objects \\
\qquad\qquad\qquad pending = thr.pending \ \textbf{in} \\
\qquad\qquad\qquad\quad vdm'.cpus(id_c).objects = mergePending(objects, pending))] \wedge \\
\qquad\qquad [d_t \neq step_t \ \Rightarrow \ d_t' = d_t - step_t])
\end{array}
$$

Figure 3.2: Definition of $commitPendingValuesAndUpdateTime$

3.3.1 A Co-Simulation Semantics for VDM

To perform any simulation between a DE and CT simulator that is based on time synchronization, it is necessary to fix the notion of time, used by the simulators, to standard time units, as mentioned in [P87]. In the case of VDM this also requires the corresponding units used to define the capacity of CPUs and busses to be defined.

An initial description of a VDM-RT semantics with an embedded co-simulation engine has been described by Verhoef et al. in [53, 147, 145]. This presentation mixes the behaviour of the co-simulation with with the DE simulator i.e. VDM-RT, and presents a framework where it may be possible to use any CT simulator but where it is difficult to use other DE simulators.

This semantics inspired the creation of a new SOS semantics [P84] that defined a new semantics model that extracted the co-simulation out of the VDM-RT semantics and presented it as a standalone semantics. This has two primary advantages: the first is that this provides a clear, modular architecture; and the second is that it allowed us to consider requirements for each of the semantics taking part in a co-simulation.

3.3.2 Semantics of a Generic Co-Simulation Framework

The work in [P84] inspired the creation of a generic co-simulation framework that could be applied to any notation that respected a number of constraints. In [P18] we present a semantics for generic co-simulation of heterogenous models. The framework defines two types of semantics that can be used with the framework:

- *Single-state Simulation Semantics (SSS)* and
- *Transactional Simulation Semantics (TSS)*,

It also lists a number of common and SSS/TSS specific constraints. The common constraints generally require the semantics to be able to "step" based on time, and be able to take shared state into account. The main difference between SSS and TSS is that the complete state of SSS always is observable while TSS may hide transactional updates until a certain time has passed.

The static state for a co-simulation with a SSS-type and TSS-type simulator is given by the configuration ($Config$):

$$Config = TSS \times SSS \times \Sigma \times Time \times Time \times Event\text{-}\mathbf{set} \times Tag$$

where TSS and SSS represents the TSS-type and SSS-type simulators. Σ represents the shared state between the simulators. The first $Time$ represents the current time, and the second $Time$ represents a time bound that must be respected by single-state simulator semantics; $Event$-**set** is the set of $Events$ generated by SSS for the TSS models to react to. The Tag is a token to record which of the two semantics models took the last step.

The behaviour of the co-simulation is described by the transition relation, $\xrightarrow{cs}$ which is a relation over configurations. The two inference rules, in Figure 3.3, define the high-level behaviour of the co-simulation semantics. The first rule, TSS Step, takes a configuration, previously stepped by $\langle \mathrm{SSS} \rangle$ and uses the transition rule $\xrightarrow{tss}$ to step the $\langle \mathrm{TSS} \rangle$ simulator. The second rule, SSS Step, takes the output of the TSS Step rule as input and uses the transition rule $\xrightarrow{sss}$ to step the $\langle \mathrm{SSS} \rangle$ simulator. Each of the rules merges back in the state changes performed by the individual simulators according to the function shown in Figure 3.4.

The two simulators $\langle \mathrm{TSS} \rangle$ and $\langle \mathrm{SSS} \rangle$ described above can be replaced by e.g. a customized VDM interpreter for $\langle \mathrm{TSS} \rangle$, and a differential equation simulator, like *20-sim*, for the $\langle \mathrm{SSS} \rangle$ simulator; each of which has been shown to be useful in performing a co-simulation in the DESTECS project. Further

$$\text{TSS Step}\ \frac{(tss, \sigma, \tau, events) \xrightarrow{tss} (tss', \sigma', \tau_b') \quad \sigma'' = mergeStates(\sigma, \sigma', \langle \text{TSS} \rangle)}{(tss, sss, \sigma, \tau, \tau_b, events, \langle \text{SSS} \rangle) \xrightarrow{cs} (tss', sss, \sigma'', \tau, \tau_b', events, \langle \text{TSS} \rangle)}$$

$$\text{SSS Step}\ \frac{(sss, \sigma, \tau_b) \xrightarrow{sss} (sss', \sigma', \tau', events') \quad \sigma'' = mergeStates(\sigma, \sigma', \langle \text{SSS} \rangle)}{(tss, sss, \sigma, \tau, \tau_b, events, \langle \text{TSS} \rangle) \xrightarrow{cs} (tss, sss', \sigma'', \tau', \tau_b, events', \langle \text{SSS} \rangle)}$$

Figure 3.3: Inference rules for the behaviour of the co-simulation engine.

$$mergeStates: \Sigma \times \Sigma \times Tag \to \Sigma$$
$$mergeStates(\sigma_o, \sigma_t, tag) == \sigma_o \dagger \{id \mapsto \sigma_t(id) \mid id \in \mathbf{dom}\, \sigma_t \wedge tag = \sigma_o(id).\#2\}$$

Figure 3.4: Function to merge two shared states, taking only the values paired with a specific tag value.

details about the constraints of the simulators and usage in the DESTECS project can be found in [P18].

Contribution 15. A new generic co-simulation framework to co-simulate two different types of semantic models.

3.3.3 The Semantics of VDM in a Co-Simulation Setting

The VDM-RT semantics needed to be extended to integrate into the co-simulation framework from [P18]. The semantics of VDM-RT uses transactions, and therefore it can only be integrated into the co-simulation framework as a TSS simulator. The extensions required to achieve this is mainly focused on the external communication where state is shared with other simulators. In [P86] we describe how the $VDMRT$ record is extended with two new fields $sharemap$ and $eventmap$ representing the shared state and the handlers for events that may be generated by others (SSS simulators):

$$
\begin{array}{rrcl}
DE & :: & cpus & : & CPUs \\
 & & busses & : & Busses \\
 & & classes & : & Classes \\
 & & time & : & Time \\
 & & sharemap & : & SharedMap \\
 & & eventmap & : & EventMap
\end{array}
$$

The extension to the semantics is confined to a single rule, Big Step, in the VDM-RT semantics. The extensions shown in Figure 3.5 extends the original rule DE Big Step in Figure 3.5. The rule is extended in three places: line 1 and 9 handles the synchronization between the internal VDM state and that of the co-simulation based on the $sharemap$, line 5 handles execution of event handlers based on the $eventmap$.

$$
\frac{\begin{array}{ll}
de^1 = updateDEFromShared(de, \sigma_o) & \textbf{(1)} \\
de^2 = commitPendingValuesAndUpdateTime(de^1, \tau) & (2) \\
de^2 \xrightarrow{busses} de^3 & (3) \\
de^4 = createPeriodicThreads(de^3) & (4) \\
de^5 = createEventThreads(de^4, events) & \textbf{(5)} \\
de^6 = doContextSwitches(de^5) & (6) \\
de^6 \xrightarrow{exec} (de^7, \tau_b) & (7) \\
\tau_b' = min(\tau_b, minPendingCommitTime(de^7)) & (8) \\
\sigma_s = extractValuesFromDE(de^7) & \textbf{(9)}
\end{array}}{(de, \sigma_o, \tau, events) \xrightarrow{vdmrt} (de^7, \sigma_s, \tau_b')} \text{DE Big Step}
$$

Figure 3.5: Definition of the DE Big Step rule.

These small semantic changes illustrate that the semantic definition of VDM-RT easily integrates with the co-simulation without requiring substantial modifications to the language. The modifications only extract information, updates shared state, or creates new threads as a result of an external event. The changes does not influence the semantics of the language but rather control the semantics steps and updates shared state in VDM between the steps.

An possible implementation[1] of the DE Big Step might be able to optimize the execution by skipping the new additions in Figure 3.5 line 1, 5 and 9 in the cases where the $\xrightarrow{exec}$ neither reads or writes shared state. The result is that the

[1] See Section 2.3.4 for more detail about an implementation of the VDM-RT interpreter in a co-simulation setting.

standard Big Step transition rule from the VDM-RT semantics can be used for these cases and thus speeding up any implementation since communication with the co-simulation framework is not needed in these cases. The argument for allowing such an optimization is that the update/extract of shared state in the first and last hypothesis of the DE Big Step rule is an identity function in these cases where the model neither reads or writes shared state. The same applies to the fifth hypothesis that creates new event threads, where the set of events always is empty when the shared state is unchanged. For practical usage such semantics optimisations are essential for the user.

Contribution 16. An extension to the VDM-RT semantics integrating it with the co-simulation framework [P18] from Contribution [C15].

4

Conclusion

This chapter summarizes and concludes the results achieved in this thesis. The objectives of the thesis defined in Chapter 1 are related to the tool automation and semantics contributions in Chapters 2 and 3. The various kinds of automated analysis, and the underlying semantics definitions comprise the results of this thesis.

4.1 Introduction

This thesis provides various automated kinds of analysis for software systems that have been formally specified; it also includes translations between specification languages. The primary focus is validation of specifications but the area of verification is also introduced. The ability to model embedded reactive control systems that rely on higher-fidelity environment representations is made possible by the definition of a co-simulation framework and supported by a well-defined semantics.

The purpose of this chapter is to evaluate the outcome of the thesis, and assess to what extent the objectives have been met. Section 4.2, summarises the research contributions made. Section 4.3, evaluates to what extent the research contributions meet the objectives of the thesis as defined in Chapter 1. Finally, future work is described and presented in Section 4.4.

4.2 Research Contributions

The research contributions are grouped into two main categories: *tool automation* and *semantics*. The contributions are shown in Figure 4.1 which illustrates how some contributions form a basis for others. The contributions are grouped in the categories described in Chapter 2 with the addition of the *Semantics* category that shows how the semantic contributions provides a foundation that other contributions rely upon.

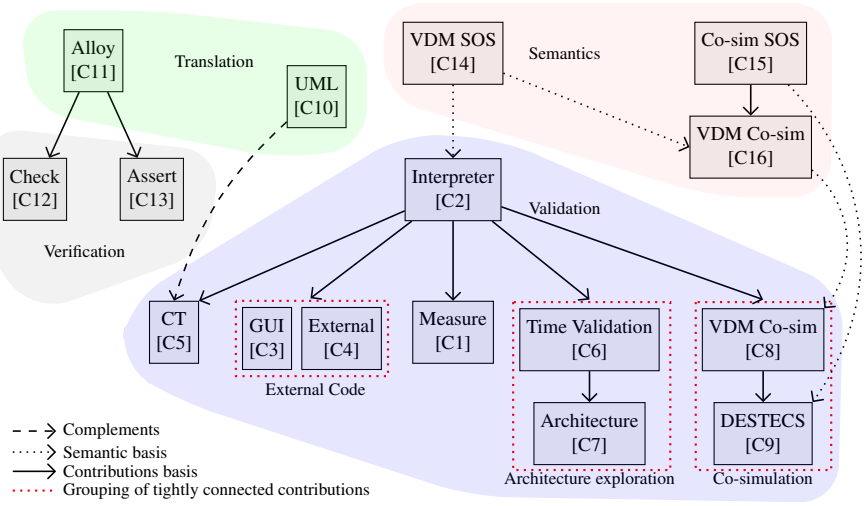


Figure 4.1: Contribution overview.

4.2.1 Tool Automation

The contributions of this PhD thesis that relate to tool automation, as described in Chapter 2, enable various kinds of automated analysis and transformation with the primary focus on validation. Section 2.3.1, describes how run-time termination checks of recursive functions can be enabled during interpretation by the use of measure functions [C1], and how an interpreter can be made deterministic during interpretation but also during debugging [C2]. The contributions [C3] and [C4] describe how a foundation is defined which allows the integration of a graphical interface with an interpreter for run-time manipulation of a specification, and how external code can be used with a specification during run-time without changing the formal language.

Section 2.3.2, describes how the testing process of VDM specifications can be automated using combinatorial testing, and how the generated tests can be reduced using shape reduction; a new construct to express parallelism is also described [C5].

Section 2.3.3, describes two contributions that enable analysis of time constraints [C6] at run-time and the automatic generation of multiple architectures for a specification of a distributed system; based on the number of CPUs and busses, and deployment of artifacts [C7]. Section 2.3.4, describes how the interpreter from [C2] is extended to enable co-simulation of specifi-

cations which represents systems that require a high-fidelity representation of the environment and thus consist of both a discrete-event (DE) controller and a continuous-time (CT) environment [C8]. This has been further extended and comprises the DESTECS tool [C9] used throughout the DESTECS project for a number of industrial case studies.

Section 2.4, describes two types of translations. The first translation is between VDM and UML and enables VDM specifications to be represented as UML class diagrams; it also enables a translation between UML sequence diagrams into the representation of combinatorial testing feature of VDM [C10]. The second translation is a translation from VDM to Alloy with focuses on implicit specifications [C11]. This translation then enables a new approach to check and evaluate implicit specifications through the static analysis provided by the Alloy analyser [C12]. Section 2.5, describe how assertions can be expressed using existing VDM expressions that then can be used in combination with the static analysis provided by the Alloy analyser [C13].

4.2.2 Semantics

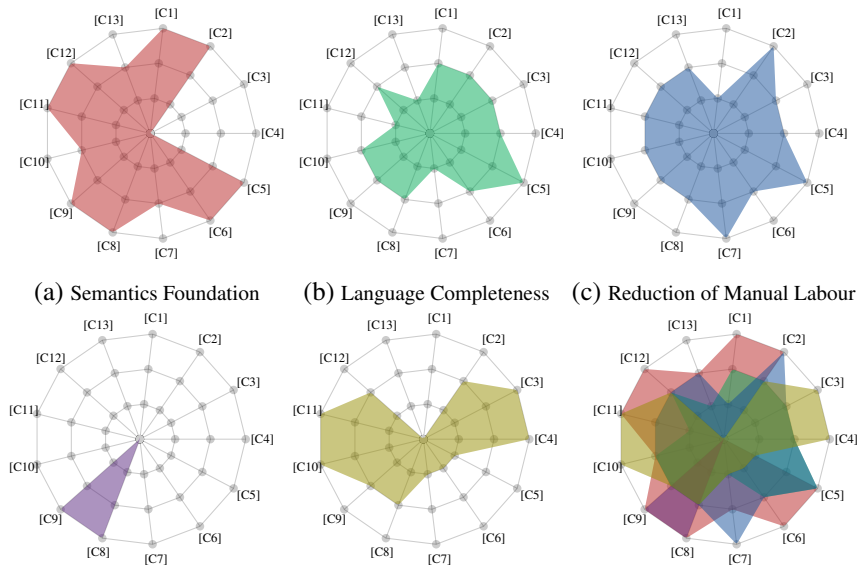
The contributions described in Chapter 3 define the semantic foundation required for contribution [C2] and [C8] from Chapter 2. Section 3.2, identifies where the semantics, existing prior to this work, is lacking precision. This is then addressed in a new semantics given to the real-time extension of the VDM language using an SOS style that forms the basis of the VDM interpreter [C14]. Section 3.3.2, addresses the semantic challenges of co-simulation and defines a new semantics framework allowing two types of semantics models to be joined in a co-simulation [C15]. Finally, Section 3.3.3, describes how the semantics for VDM-RT from [C14] is extended to enable VDM to integrate with the co-simulation framework as one of the two semantics models [C16].

4.3 Evaluation of Contributions

In this section, the contributions described in Chapters 2 and 3 are evaluated with respect to the evaluation criteria listed in Section 1.7. Evaluation of the industrial impact of the work in this thesis is outside the scope of this thesis; however, the tools developed during this PhD project that implements all contributions, except [C7, C11, C12, C13], have been used by the industrial partners during the DESTECS project as well as commercially outside the DESTECS project.

A relation between the contributions, and the evaluation criteria as listed in Section 1.7, is presented in Figure 4.2. The figure illustrates an informal ranking of the contributions that provides an overview of how the individual contributions fulfil the general criteria. The scale used in the figures indicates to what extent the contributions fulfil the criteria measured in the figures, the closer to the edge of the circle the greater the contribution.

An evaluation of all contributions is described below, with respect to the evaluation criteria. The evaluation relates to each of the individual figures from Figure 4.2, except for Figure 4.2f which illustrates to what extent all the contributions add to the overall fulfilment of the evaluation criteria. Note that the semantics contributions are excluded from the figure since they serve as the underlying foundation behind the contributions in Figure 4.2a. Section 4.3.6, prioritises the contributions according to their academic contribution.



(a) Semantics Foundation (b) Language Completeness (c) Reduction of Manual Labour

(d) Fidelity of Embedded Control Systems (e) Integration with Other Tools (f) Combined Overview

Figure 4.2: Relation between contributions and evaluation criteria.

4.3.1 Semantics Foundation

The various kinds of analysis developed in Chapter 2 are all centred around the VDM formal modelling language, and the co-simulation framework. The VDM-SL dialect is already standardised and semantics have also been given to VDM++ prior to this work. The VDM-RT dialect did not have a sufficient strong semantic foundation, and thus a new semantics has been provided. A co-simulation semantics has also been given for the co-simulation that has been implemented, and used throughout the DESTECS project; including an extension semantics to the VDM-RT semantics that clearly describes how it integrates into the co-simulation framework. The only contributions that do not depend on the semantics of a language are [C3, C4]; these contributions extend the run-time interaction with the VDM interpreter and have intentionally been designed to avoid language changes. A graphical representation of this is illustrated in Figure 4.2a.

4.3.2 Language Completeness

The analysis of a specification can only be complete if the chosen analysis covers the complete modelling language used throughout the specification; otherwise the analysis must be comprised of one or more types of analysis that in combination covers the complete modelling language.

The contributions defined in Section 2.3 are complete with regard to the executable subset of VDM. However, this excludes non-executable specifications that make use of e.g. implicitly defined functions or operations. The contributions in Section 2.5 describes how it is possible to complement the validation analysis with formal verification enabling implicit specifications to be checked; this analysis is limited to a subset of VDM-SL excluding e.g. recursive functions. The completeness is illustrated in Figure 4.2b where it can be seen that validation does not alone provide a complete analysis of all possible models expressed in VDM.

4.3.3 Reduction of Manual Labour

In general, all contributions in Chapter 2 reduce manual labour. However, the contribution [C1] requires the user to manually extend the specifications with measure functions, and in return the run-time termination checks of recursive functions will be enabled. The three contributions that in particular reduce manual labour as illustrated in Figure 4.2c are: the interpreter [C2] itself, the combinatorial testing [C5] that in a compressed way can generate large test

suites, and at the same time use new reduction techniques to improve the test representation. Lastly, there is the automatic generation of alternative architectures and deployments for VDM-RT models [C7] that in combination with the run-time check of time constraints checking [C6] automatically checks various architectures and deployments of a system without user interaction.

4.3.4 Fidelity of Embedded Control Systems

The discrete-event (DE) domain of VDM is not sufficient to represent the physical environment required for an embedded reactive control system that relies heavily on a high-fidelity environment representation, and thus a co-simulation solution has been developed. This enables two languages to be combined in a simulation using individual simulators and notations for each language. This solution has been used with success in the DESTECS project on a number of industrial case studies [148]. The semantics work behind this solution is described in Chapter 3 as three contributions [C14, C15, C16], which have been implemented into a simulation tool [C8, C9] described in Section 2.3.4. Figure 4.2d, illustrates that these alone contribute to the fidelity of a embedded reactive control system but in full comply with the required criteria.

4.3.5 Integration with Other Tools

An integration with other tools has been found desirable and in Figure 4.2e it is illustrated how the individual contributions relate to the criteria. An integration is in particular desirable for e.g. visualization both in the form of a translation between VDM and UML [C10] as described in Section 2.4.1 but also by enabling visualization as part of a simulation. In Section 2.3.1.3 it is described how the interpreter is extended to allow models to be graphically presented during interpretation while accepting user input [C3] but also how the specification may access information from other sources or in other ways manipulate with other tools or systems [C4]. Finally, in Section 2.4.2 a translation between VDM and Alloy is presented enabling VDM specifications to be analysed by the Alloy Analyzer [C11, C12].

4.3.6 Academic Prioritisation of Contributions

This section prioritises the contributions based on the academic value and generality. The prioritisation is done in three groups and is illustrated in Figure 4.3. The group of most significant contribution is [C15, C9]; they define a

generic co-simulation framework which is generally applicable for other DE and CT languages. This means that it can be applied to other formal languages without requiring any change to the semantics.

The groups of second most significant contributions are [C1, C3, C4, C5, C6, C7] these are new but not directly applicable for other formal languages due to the tight coupling with the VDM language and tooling. However, they can still serve as a starting point for others but require customisation.

Last, is the group of contributions [C2, C8, C10, C11, C12, C13, C14, C16] that are new to VDM, but are inspired by existing theory that has already been applied to tools for other formal languages. These contributions contribute little new theory, but they have played a central role in the fulfilment of the objectives of this PhD project; thereby contribution to the industrial adoption of formal methods.

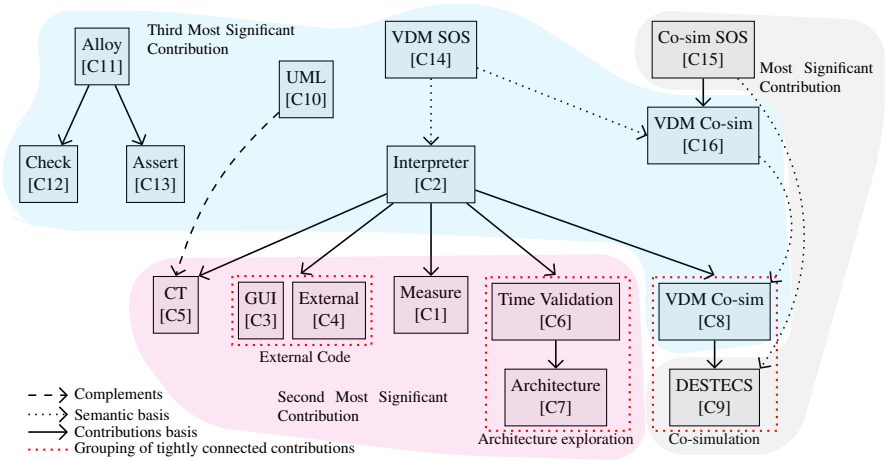


Figure 4.3: Contribution prioritisation overview.

4.4 Future Work

In this section, some directions for future work are given. These are grouped into three of the categories from Figure 2.1 in Chapter 2: Validation, Translation and Formal Verification.

4.4.1 Validation

Interpreter

The VDM interpreter presented in [P87] does not fully conform to the semantics. It interprets logical expressions using standard left to right evaluation which semantically is equivalent to McCarthy logic [100] instead of standard Logic of Partial Functions (LPF) [7, 63, 9] used in the standard of VDM-SL. Therefore, it is worth investigating to what extent a change from McCarthy logic to LPF adds complexity or if a better solution exists than the one proposed. Likewise, discrepancies with the new VDM-RT semantics [C14] could be addressed.

Measure Functions

The measure functions [C1] used for run-time termination check of recursive functions cannot be expressed over tree structures. However, from a theoretical point of view this should be possible. Therefore, it could be investigated if this claim is true and what, if any, changes or additions are needed in VDM to support these structures.

Code Integration

It would be desirable to extend the ability to use graphical user interfaces during interpretation [C3] with capabilities like the ones from B-Motion Studio. The main challenge is the handling of dynamic object creation that needs to be solved before a tool like B-Motion Studio can be integrated. However, if possible it will significantly lighten the user task of creating graphical interfaces. Likewise, the integration of external code [C4] could be extended with the capabilities to seamlessly integrate with other languages than Java.

Combinatorial Testing

The combinatorial testing feature [C5] may, in the future, be improved by using other techniques for test reduction and thus removing test sequences which are over represented. Alternatively, other techniques can be used where static analysis is used to generate tests so that all parts of a specification are covered [24].

The combinatorial test generation could also be improved by generating test sequences directly to a permanent storage instead of memory, and thus removing the memory restriction on the number of tests that can be generated. Finally, it could be extended to use multiple coordinated interpreters to better utilize many core systems.

Real-Time Extensions

The timing constraints for VDM-RT specifications are currently expressed in a custom syntax [C6]. However, it might be beneficial to use full temporal logic and the associated notation to express such timing constraints. It would also be worth completing the implementation for the automatic generation of architectures [C7] and evaluate to what extent it is automatically possible to rank architectures when used in combination with the timing constraints.

Co-simulation

The co-simulation presented in this thesis could be compared to a solution using a single language to describe elements of both the DE and CT domain. It could then be evaluate to what extent development tools could be reused.

It will also be advantageous to investigate how the simulation framework [C15] can be extended to support a N-ary simulation between multiple $\langle \text{TSS} \rangle$ and $\langle \text{SSS} \rangle$. It is believed that a simulation including multiple $\langle \text{TSS} \rangle$ simulators is possible with only minor adjustments. However, combining multiple $\langle \text{SSS} \rangle$ simulators may be significantly more challenging if they are represented by CT simulators that share elements that interact; if no interaction exists between the CT specifications then it is believed to be possible to integrate them within the framework.

4.4.2 Translation

UML Translation

The translation between VDM and UML [C10] mainly focuses on the static structure in VDM with the exception of the limited work done with sequence diagrams and combinatorial testing. It is believed that further work can be done to extend the translation to include sequence diagrams and state diagrams. A translation between OCL and the pre-, post-conditions and invariants of VDM can also be developed to enrich the UML representation of a VDM specification or to allow UML models annotated with OCL to be translated to VDM for further analysis and refinement. Finally, it would also be beneficial to investigate to what extent the translation can cope with large industrial specifications that may consist of thousands of classes.

Alloy Translation

The translation between VDM and Alloy is rather limited [C11], and to become truly useful it must be extended, and a fully automatic translation must be implemented.

4.4.3 Formal Verification

Formal verification has only briefly been touched in this thesis with the generation of proof obligations and model finding by the use of the Alloy analyzer. However, it will be a significant contribution to be able to translate VDM specifications and its proof obligations into an intermediate language shared among automated theorem provers, and thereby enable support for theorem provers. The existing work only considers a small subset of VDM-SL for a translation to HOL. However, a connection to the Isabelle theorem prover using a HOL logic is planned for the European research project COMPASS [36].

The support for static analysis is almost non-existent, for the VDM language but would be of great value; existing theories can be used to perform analysis of VDM specifications e.g. deadlocks. The outcome of the analysis performed by translating a number of VDM specifications to Alloy [C11] has indicated that the existing tool support in the area is insufficient. The proposed use of VDM expressions [C13] may also be integrated into the VDM language to allow system level properties to be specified within VDM and then later used in proofs and static checking.

Part II

Publications

5

The Overture Initiative – Integrating Tools for VDM

The paper presented in this chapter has been accepted as a Software Engineering Note publication.

[P74] Peter Gorm Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl, and Marcel Verhoef. *The Overture Initiative – Integrating Tools for VDM*. SIGSOFT Softw. Eng. Notes, 35(1):1–6, January 2010.

6

Connecting UML and VDM++ with Open Tool Support

The paper presented in this chapter has been accepted as a peer-reviewed conference paper.

[P89] Kenneth Lausdahl, Hans Kristian Agerlund Lintrup, and Peter Gorm Larsen. *Connecting UML and VDM++ with Open Tool Support*. In Ana Cavalcanti and Dennis R. Dams, editors, Proceedings of the 2nd World Congress on Formal Methods, volume 5850 of Lecture Notes in Computer Science, pages 563–578, November 2009. Springer-Verlag. ISBN 978-3-642-05088-6.

7

Translating VDM to Alloy

The paper presented in this chapter has been accepted as a peer-reviewed conference paper.

[P83] Kenneth Lausdahl. *Translating VDM to Alloy*. In Einar Broch Johnsen and Luigia Petre, editors, Integrated Formal Methods, volume 7940 of *Lecture Notes in Computer Science*, pages 46–60. Springer Berlin Heidelberg, 2013. 10th International Conference, IFM 2013.

8

A Deterministic Interpreter Simulating a Distributed Real Time System using VDM

The paper presented in this chapter has been accepted as a peer-reviewed conference paper.

[P87] Kenneth Lausdahl, Peter Gorm Larsen, and Nick Battle. *A Deterministic Interpreter Simulating a Distributed Real Time System using VDM*. In Shengchao Qin and Zongyan Qiu, editors, Formal Methods and Software Engineering, volume 6991 of Lecture Notes in Computer Science, pages 179–194, 2011. Springer-Verlag. ISBN 978-3-642-24558-9.

9

Combinatorial Testing for VDM

The paper presented in this chapter has been accepted as a peer-reviewed conference paper.

[P79] Peter Gorm Larsen, Kenneth Lausdahl, and Nick Battle. *Combinatorial Testing for VDM*. In Proceedings of the 2010 8th IEEE International Conference on Software Engineering and Formal Methods, SEFM'10, pages 278–285, Washington, DC, USA, September 2010. IEEE Computer Society. ISBN 978-0-7695-4153-2.

10

Combining VDM with Executable Code

The paper presented in this chapter has been accepted as a peer-reviewed conference paper.

[P107] Claus Ballegaard Nielsen, Kenneth Lausdahl, and Peter Gorm Larsen. *Combining VDM with Executable Code*. In John Derrick, John Fitzgerald, Stefania Gnesi, Sarfraz Khurshid, Michael Leuschel, Steve Reeves, and Elvinia Riccobene, editors, Abstract State Machines, Alloy, B, VDM, and Z, volume 7316 of Lecture Notes in Computer Science, pages 266– 279, 2012. Springer-Verlag. ISBN 978-3-642- 30884-0.

11

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

The paper presented in this chapter has been accepted as a peer-reviewed workshop paper.

[P125] Augusto Ribeiro, Kenneth Lausdahl, and Peter Gorm Larsen. *Run-Time Validation of Timing Constraints for VDM-RT Models*. In Sune Wolff and John Fitzgerald, editors, Proceedings of the 9th Overture Workshop, number ECE-TT-2 in Technical Report Series, pages 4–16, June 2011.

12

Semantics Focused Papers

The papers presented in this chapter has been submitted as journal papers.

[P18] Joey W. Coleman, Kenneth Lausdahl, and Peter Gorm Larsen. *Semantics for Generic Co-simulation of Heterogenous Models*. Submitted for publication to the Formal Aspects of Computing journal, April 2013.

[P86] Kenneth Lausdahl, Joey W. Coleman, and Peter Gorm Larsen. *The Execution Semantics of VDM Real-Time in a Co-Simulation Environment*. Submitted for publication to the International Journal on Software Tools for Technology Transfer, June 2013.

Bibliography

[1] J.-R. Abrial. *The B Book – Assigning Programs to Meanings*. Cambridge University Press, August 1996.

[2] Jean-Raymond Abrial. Data semantics. In *IFIP Working Conference Data Base Management*, pages 1–60, 1974.

[3] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.

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

[5] Paul Ammann, Paul E. Black, and Wei Ding. Model Checkers in Software Testing. Technical report, NIST-IR 6777, National Institute of Standards and Technology, 2002.

[6] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and et al. On challenges of Model Transformation from UML to Alloy. In *Software & Systems Modeling*, volume 9 of *1*, pages 69–86. Springer, 2010.

[7] H. Barringer, J.H. Cheng, and C.B. Jones. A Logic Covering Undefinedness in Program Proofs. *Acta Informatica*, 21:251–269, 1984.

[8] Nick Battle. VDMJ User Guide. Technical report, Fujitsu Services Ltd., UK, 2009.

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

[10] D. Bjørner and C.B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[11] Peter Breuer and Jonathan Bowen. Towards Correct Executable Semantics for Z. In J.P. Bowen and J.A. Hall, editors, *Z User Workshop*, pages 185–209. Springer-Verlag, 1994. Cambridge.

[12] J. F. Broenink, P. G. Larsen, M. Verhoef, C. Kleijn, D. Jovanovic, K. Pierce, and Wouters F. Design Support and Tooling for Dependable Embedded Control Software. In *Proceedings of Serene 2010 International Workshop on Software Engineering for Resilient Systems*, pages 77–82. ACM, April 2010.

[13] Jan Broenink. D1.4 — Project final report. Technical report, The DESTECS Project (INFSO-ICT-248134), February 2013.

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

[15] J.F. Broenink. Object-oriented modeling with bond graphs and modelica. In *Proceedings of the International Conference on Bond Graph Modeling and Simulation*, pages 163–168. Western Multi Conference, Simulation Series Vol. 31, No.1, Proceedings 1999 International Conference on Bond Graph Modeling and, 1999. 009_R99.htm.

[16] Thomas John Hørlyck Christensen. Extending the VDM++ Formal Specification Language with Type Inference and Generic Classes. Master's thesis, Aarhus University, Computer Science Department, April 2007.

[17] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The AETG System: An Approach to Testing Based on Combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, July 1997.

[P18] Joey W. Coleman, Kenneth Lausdahl, and Peter Gorm Larsen. Semantics for Generic Co-simulation of Heterogenous Models. Submitted for publication to the Formal Aspects of Computing journal, April 2013.

[19] Dan Craigen, Susan Gerhart, and Ted Ralston. *An International Survey of Industrial Applications of Formal Methods*, volume Volume 1 Purpose, Approach, Analysis and Conclusions. U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD 20899, USA, March 1993.

[20] CSK. The Dynamic Semantics of CSK VDM++ VICE. Technical report, CSK Corporation, Japan, 2005. Company Confidential.

[21] Sergiu Dascalu and Peter Hitchcock. An Approach to Integrating Semi-formal and Formal Notations in Software Specification. In *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*, pages 1014–1020, New York, NY, USA, 2002. ACM.

[22] J. Davis, R. Galicia, M. Goel, C. Hylands, E.A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, and Y. Xiong. Ptolemy-II: Heterogeneous concurrent modeling and design in Java. Technical Memorandum UCB/ERL No. M99/40, University of California at Berkeley, July 1999.

[23] A.J.J. Dick, P.J. Krause, and J. Cozens. Computer aided transformation of Z into Prolog. In J.E. Nicholls, editor, *Z User Workshop, Oxford 1989*, Workshops in Computing, pages 71–85. Springer-Verlag, 1990.

[24] Jeremy Dick and Alain Faivre. Automating the Generation and Sequencing of Test Cases from Model-Based Specifications. In J.C.P. Woodcock and P.G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, pages 268–284. Formal Methods Europe, Springer-Verlag, April 1993. Lecture Notes in Computer Science 670.

[25] E. Dürr and J.v. Katwijk. VDM++, A Formal Specification Language for Object Oriented Designs. In *COMP EURO 92*, pages 214–219. IEEE, May 1992.

[26] E.H. Dürr and N. Plat (editor). VDM++ Language Reference Manual. Afrodite (esprit-iii project number 6500) document, Cap Volmac, August 1995.

[27] Eugène Dürr, Stephen Goldsack, and Nico Plat. Rigorous Development of Concurrent and Real-Time Object-oriented Systems, March 1994. Tutorial presented at TOOLS Europe '94, Versailles, France.

[28] René Elmstrøm, Peter Gorm Larsen, and Poul Bøgh Lassen. The IFAD VDM-SL Toolbox: A Practical Approach to Formal Specifications. *ACM Sigplan Notices*, 29(9):77–80, September 1994.

[29] Houda Fekih, Leila Jemni Ben Ayed, and Stephan Merz. Transformation of B Specifications into UML Class Diagrams and State Machines. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1840 – 1844. ACM, 2006. ISBN:1-59593-108-2.

[30] J. S. Fitzgerald, P. G. Larsen, S. Tjell, and M. Verhoef. Validation Support for Real-Time Embedded Systems in VDM++. In Bojan Cukic and Jing Dong, editors, *Proc. HASE 2007: 10th IEEE High Assurance Systems Engineering Symposium*, pages 331–340. IEEE, November 2007.

[31] J. S. Fitzgerald, P. G. Larsen, and M. Verhoef. Vienna Development Method. *Wiley Encyclopedia of Computer Science and Engineering*, 2008. edited by Benjamin Wah, John Wiley & Sons, Inc.

[32] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, Second edition, 2009. ISBN 0-521-62348-0.

[33] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object–oriented Systems*. Springer, New York, 2005.

[34] John Fitzgerald, Peter Gorm Larsen, Ken Pierce, and Marcel Verhoef. A Formal Approach to Collaborative Modelling and Co-simulation for Embedded Systems. *To appear in Mathematical Structures in Computer Science*, August 2013.

[35] John Fitzgerald, Peter Gorm Larsen, and Shin Sahara. VDMTools: Advances in Support for Formal Modeling in VDM. *ACM Sigplan Notices*, 43(2):3–11, February 2008.

[36] John Fitzgerald, Peter Gorm Larsen, and Jim Woodcock. Modelling and Analysis Technology for Systems of Systems Engineering: Research Challenges. In *INCOSE*, Rome, Italy, July 2012.

[37] J.S. Fitzgerald and C.B. Jones. Proof in the Validation of a Formal Model of a Tracking System for a Nuclear Plant. In J.C. Bicarregui, editor, *Proof in VDM: Case Studies*, FACIT Series. Springer-Verlag, 1998.

[38] Oana Florescu, Jeroen Voeten, Marcel Verhoef, and Henk Corporaal. Reusing Real-Time Systems Design Experience Through Modelling Patterns. In *Forum on specification and Description Languages (FDL)*. ECSI, 2006. Received the best paper award at FDL 2006. This paper is available on-line at `http://www.es.ele.tue.nl/premadona/publications/FVVC06.pdf`.

[39] Gordon Fraser and Franz Wotawa. Improving Model-Checkers for Software Testing. In *Seventh International Conference on Quality Software (QSIC 2007)*. IEEE, 2007.

[40] Peter Fritzson and Vadim Engelson. Modelica - A Unified Object-Oriented Language for System Modelling and Simulation. In *ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 67–90. Springer-Verlag, 1998.

[41] Brigitte Fröhlich. Program Generation based on Postconditions. In M.H. Hmaza, editor, *Software Enginerring (SE'97)*. IASTED, ACTA Press, November 1997.

[42] Brigitte Fröhlich. *Towards Executability of Implicit Definitions*. PhD thesis, TU Graz, Institute of Software Technology, September 1998.

[43] Norbert E. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, pages 323–334, September 1992.

[44] Guy Gallasch and Lars M. Kristensen. Comms/CPN: A communication infrastructure for external communication with design/CPN. In *3rd Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN'01)*, pages 75–90. DAIMI PB-554, Aarhus University, aug 2001.

[45] Marie-Claude Gaudel. Testing can be Formal, too. In Peter Mosses and Michael Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development*, pages 82–96. CAAP/FASE, Springer, 1995.

[46] Andy Gravell, Peter Nederson, and So Bj. Executing formal specifications need not be harmful. *Software Engineering Journal*, 11:104–110, 1996.

[47] M.A. Groothuis, A.S. Damstra, and J.F. Broenink. Virtual prototyping through co-simulation of a cartesian plotter. In *Emerging Technologies and Factory Automation, 2008. ETFA 2008. IEEE International Conference on*, pages 697–700. IEEE Industrial Electronics Society, September 2008.

[48] I.J. Hayes and C.B. Jones. Specifications are not (Necessarily) Executable. *Software Engineering Journal*, pages 330–338, November 1989.

[49] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy Vilkomir, Martin R. Woodward, and Hussein Zedan. Using Formal Specifications to Support Testing. *ACM Comput. Surv.*, 41(2):1–76, 2009.

[50] Michael G. Hinchey and Jonathan P. Bowen. To formalize or not to formalize? *IEEE Computer*, 29(4):18–19, April 1996.

[51] C.A.R Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8), August 1978.

[52] J. Hooman and M. Verhoef. Formal semantics of a VDM extension for distributed embedded systems. In D. Dams, U. Hannemann, and M. Steffen, editors, *Concurrency, Compositionality, and Correctness, Essays in Honor of Willem-Paul de Roever*, volume 5930 of *Lecture Notes in Computer Science*, pages 142–161. Springer-Verlag, 2010.

[53] Jozef Hooman and Marcel Verhoef. Formal Semantics of a VDM Extension for Distributed Embedded Systems. In *Correctness, Concurrency and Compositionality*, LNCS Festscrift Series, 2008. Festscrift to honour professor Willem-Paul de Roever, Springer.

[54] Akram Idani and Yves Ledru. Object oriented concepts identification from formal B specifications. *Formal Methods System Design*, 30(3):217–232, 2007.

[55] ISO. Information Processing Systems — Open Systems Interconnection — LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. Technical Report ISO8807, International Standards Organization, 1989.

[56] ISO. Information technology – Z formal specification notation – Syntax, type system and semantics. Technical Report ISO/IEC 13568, International Organization for Standardization, 2002.

[57] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, Heyward Street, Cambridge, MA02142, USA, revised edition, February 2012. ISBN-10: 0262017156.

[58] Jonathan Jacky. *The Way of Z: Practical Programming with Formal Methods*. Cambridge University Press, November 1996.

[59] Jim Johnson. *My Life Is Failure: 100 Things You Should Know to Be a Better Project Leader*. Standish Group International, 2004.

[60] C.B. Jones and R.C.F. Shaw. *Case Studies in Systematic Software Development*. Prentice Hall International, 1990.

[61] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, Englewood Cliffs, New Jersey, second edition, 1990. ISBN 0-13-880733-7.

[62] Cliff B. Jones. Scientific Decisions which Characterize VDM. In J.M. Wing, J.C.P. Woodcock, and J. Davies, editors, *FM'99 - Formal Methods*, pages 28–47. Springer-Verlag, 1999. Lecture Notes in Computer Science 1708.

[63] Cliff B. Jones and Kees Middelburg. A typed logic of partial functions reconstructed classically. Technical Report 89, Department of Philosophy, Utrecht University, April 1993.

[64] Stuart Kent and Richard Moore. An Axiomatic Semantics for VDM++: OO Aspects, 1993.

[65] Soon-Kyeong Kim, Damian Burger, and David Carrington. An MDA Approach Towards Integrating Formal and Informal Modelling Languages. In Ian Hayes John Fitzgerald and Andrzej Tarlecki, editors, *FM'2005: Formal Methods*, pages 448–464, Berlin Heidelberg, July 2005. FME, Springer.

[66] J.C. Knight, K.S. Hanks, and S.R. Travis. Tool support for production use of formal techniques. In *Software Reliability Engineering, 2001. ISSRE 2001. Proceedings. 12th International Symposium on*, pages 242–251, 2001.

[67] Lukas Ladenberger, Jens Bendisposto, and Michael Leuschel. Visualising Event-B Models with B-Motion Studio. In *Proceedings of the 14th International Workshop on Formal Methods for Industrial Critical Systems*, pages 202–204. Springer-Verlag, November 2009.

[68] Regine Laleau. On the Interest of Combining UML with the B Formal Method for the Specification of Database Applications. In *ICEIS*, pages 56–63, 2000.

[69] K. Lano. Expressing the Semantics of VDM++ in RTL, 1994.

[70] P. G. Larsen, B. S. Hansen, H. Brunn, N. Plat, H. Toetenel, D. J. Andrews, J. Dawes, G. Parkin, et al. Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language, December 1996.

[71] Peter Gorm Larsen. Evaluation of Underdetermined Explicit Definitions. In M. Bertran M. Naftalin, T. Denvir, editor, *FME'94: Industrial Benefit of Formal Methods*, pages 233–250. Springer-Verlag, October 1994.

[72] Peter Gorm Larsen. Response to "the formal specification of safety requirements for storing explosives". *Formal Aspects of Computing*, 6(5):565–568, 1994.

[73] Peter Gorm Larsen. Ten Years of Historical Development: "Bootstrapping" VDM-Tools. *Journal of Universal Computer Science*, 7(8):692–709, 2001.

[P74] Peter Gorm Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl, and Marcel Verhoef. The Overture Initiative – Integrating Tools for VDM. *SIGSOFT Softw. Eng. Notes*, 35(1):1–6, January 2010.

[P75] Peter Gorm Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl, and Marcel Verhoef. The Overture Initiative – Integrating Tools for VDM. In Min Zhang and Volker Stolz, editors, *Harnessing Theories for Tool Support in Software*, pages 9–19, November 2010.

[76] Peter Gorm Larsen and John Fitzgerald. Recent Industrial Applications of VDM in Japan. In Jonathan Bowen Paul Boca and Peter Gorm Larsen, editors, *FACS 2007 Christmas Workshop: Formal Methods in Industry*, Electronic Workshops in Computing. British Computer Society, December 2007.

[77] Peter Gorm Larsen and Poul Bøgh Lassen. An Executable Subset of Meta-IV with Loose Specification. In *VDM '91: Formal Software Development Methods*. VDM Europe, Springer-Verlag, March 1991.

[P78] Peter Gorm Larsen and Kenneth Lausdahl. Overture/VDM Tools Status. Handout at SEFM20 Tool Workshop, September 2010. Second edition.

[P79] Peter Gorm Larsen, Kenneth Lausdahl, and Nick Battle. Combinatorial Testing for VDM. In *Proceedings of the 2010 8th IEEE International Conference on Software Engineering and Formal Methods*, SEFM '10, pages 278–285, Washington, DC, USA, September 2010. IEEE Computer Society. ISBN 978-0-7695-4153-2.

[80] Peter Gorm Larsen, Kenneth Lausdahl, and Nick Battle. The VDM-10 Language Manual. Technical Report TR-2010-06, The Overture Open Source Initiative, April 2010.

[81] Peter Gorm Larsen and Wiesław Pawłowski. The Formal Semantics of ISO VDM-SL. *Computer Standards and Interfaces*, 17(5–6):585–602, September 1995.

[82] Kenneth Lausdahl. Enhancing Formal Modelling Tool Support with Increased Automation. Technical Report ECE-TR-4, Aarhus University, October 2011.

[P83] Kenneth Lausdahl. Translating VDM to Alloy. In Einar Broch Johnsen and Luigia Petre, editors, *Integrated Formal Methods*, volume 7940 of *Lecture Notes in Computer Science*, pages 46–60. Springer Berlin Heidelberg, 2013. 10th International Conference, IFM 2013.

[P84] Kenneth Lausdahl, Joey W. Coleman, and Peter Gorm Larsen. Towards a Co-simulation Semantics of VDM-RT/Overture and 20-sim. In Nico Plat, Claus Ballegaard Nielsen, and Steve Riddle, editors, *Proceedings of the 10th Overture Workshop*, number CS-TR-1345 in Technical Report Series, pages 30–37. Computing Science, Newcastle University, August 2012.

[P85] Kenneth Lausdahl, Joey W. Coleman, and Peter Gorm Larsen. Semantics of the VDM Real-Time Dialect. Technical Report ECE-TR-13, Aarhus University, April 2013.

[P86] Kenneth Lausdahl, Joey W. Coleman, and Peter Gorm Larsen. The Execution Semantics of VDM Real-Time in a Co-Simulation Environment. Submitted for publication to the International Journal on Software Tools for Technology Transfer, June 2013.

[P87] Kenneth Lausdahl, Peter Gorm Larsen, and Nick Battle. A Deterministic Interpreter Simulating A Distributed real time system using VDM. In Shengchao Qin and Zongyan Qiu, editors, *Formal Methods and Software Engineering*, volume 6991 of *Lecture Notes in Computer Science*, pages 179–194, Berlin, Heidelberg, October 2011. Springer-Verlag. ISBN 978-3-642-24558-9.

[88] Kenneth Lausdahl and Hans Kristian Lintrup. Coupling Overture to MDA and UML. Master's thesis, Aarhus University/Engineering College of Aarhus, December 2008.

[P89] Kenneth Lausdahl, Hans Kristian Agerlund Lintrup, and Peter Gorm Larsen. Connecting UML and VDM++ with Open Tool Support. In Ana Cavalcanti and Dennis R. Dams, editors, *Proceedings of the 2nd World Congress on Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 563–578, Berlin, Heidelberg, November 2009. Springer-Verlag. ISBN 978-3-642-05088-6.

[P90] Kenneth Lausdahl and Augusto Ribeiro. Automated Exploration of Alternative System Architectures with VDM-RT. In Sune Wolff and John Fitzgerald, editors, *Proceedings of the 9th Overture Workshop*, number ECE-TT-2 in Technical Report Series, pages 17–31, June 2011.

[P91] Kenneth Lausdahl, Marcel Verhoef, Peter Gorm Larsen, and Sune Wolff. Overview of VDM-RT Constructs and Semantic Issues. In Ken Pierce, Nico Plat, and Sune Wolf, editors, *Proceedings of the 8th Overture Workshop*, number CS-TR-1224 in Technical Report Series, pages 57–67, September 2010.

[92] Y. Ledru and L. du Bousquet. An Executable Formal Specification of a Test Generator. In *Automated Software Engineering 06*. IEEE, 2006.

[93] Yves Ledru, Frédéric Dadeau, Lydie du Bousquet, Sébastien Ville, and Elodie Rose. Mastering Combinatorial Explosion with the Tobias-2 Test Generator. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 535–536, New York, NY, USA, 2007. ACM.

[94] Yves Ledru, Lydie du Bousquet, Olivier Maury, and Pierre Bontron. Filtering TOBIAS Combinatorial Test Suites. In M. Wermelinger and T. Margaria-Steffen, editors, *FASE 2004*, pages 281–294, Springer-Verlag Berlin Heidelberg, 2004. LNCS 2984.

[95] M. Leuschel and M. J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.

[96] Michael Leuschel and Michael Butler. Prob: A model checker for b. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 855–874. Springer Berlin Heidelberg, 2003.

[97] B. Mahony and Jin Song Dong. Blending Object-Z and Timed CSP: an introduction to TCOZ. In *Software Engineering, 1998. Proceedings of the 1998 International Conference on*, pages 95–104, 1998.

[98] Petra Malik, Lindsay Groves, and Clare Lenihan. Translating Z to Alloy. In *Abstract State Machines, Alloy, B and Z*, volume 5977/2010 of *Lecture Notes in Computer Science*, pages 377–390. Springer, 2010.

[99] Paulo J. Matos and Jo ao Marques-Silva. Model Checking Event-B by Encoding into Alloy. In *Abstract State Machines, B and Z*, volume 5238/2008 of *Lecture Notes in Computer Science*, page 346. Springer, 2008.

[100] J. McCarthy. A Basis for a Mathematical Theory of Computation. In *Western Joint Computer Conference*, 1961. Then published in: Computer Programming and Formal Systems (P.Braffort, D.Hirstberg eds.) North Holland 1967, 33–70.

[101] Leonid Mikhailov, Michael Butler, Leonid Mikhailov, and Michael Butler. An approach to combining b and alloy. In *In Proc. of ZB 2002, volume 2272 of LNCS*, pages 140–161. Springer-Verlag, 2002.

[102] R. Milner. A calculus of communicating systems. *LNCS 92*, 1980.

[103] A.K. Mok. Towards mechanization of real-time system design. In A.M. van Tilborg and G.M. Koob, editors, *Foundations of Real-Time Computing. Formal Specifications and Methods*. Kluwer Academic Publishers, 1991.

[104] David Holst Møller and Christian Rane Paysen Thillermann. Using Eclipse for Exploring an Integration Architecture for VDM. Master's thesis, Aarhus University/Engineering College of Aarhus, June 2009.

[105] Paul Mukherjee, Fabien Bousquet, Jérôme Delabre, Stephen Paynter, and Peter Gorm Larsen. Exploring Timing Properties Using VDM++ on an Industrial Application. In J.C. Bicarregui and J.S. Fitzgerald, editors, *Proceedings of the Second VDM Workshop*, September 2000. Available at www.vdmportal.org.

[106] P. Naur and (Ed.) B. Randell. Software Engineering: Report on a Conference sponsored by the NATO Science Committee. Garmisch, Germany, 7th to 11th October 1968, Brussels, Scientific Affairs Division, NATO, January 1969.

[P107] Claus Ballegaard Nielsen, Kenneth Lausdahl, and Peter Gorm Larsen. Combining VDM with Executable Code. In John Derrick, John Fitzgerald, Stefania Gnesi, Sarfraz Khurshid, Michael Leuschel, Steve Reeves, and Elvinia Riccobene, editors, *Abstract State Machines, Alloy, B, VDM, and Z*, volume 7316 of *Lecture Notes in Computer Science*, pages 266–279, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-30884-0.

[P108] Claus Ballegaard Nielsen, Kenneth Lausdahl, and Peter Gorm Larsen. Using the Overture Tool as a More General Platform. In Franco Mazzanti, editor, *iFM 2012 & ABZ 2012 - Proceedings of the Posters & Tool demos Session*, pages 1–34. CNR-ISTI, June 2012.

[109] Jacob Porsborg Nielsen and Jens Kielsgaard Hansen. Development of an Overture/VDM++ Tool Set for Eclipse. Master's thesis, Technical University of Denmark, Informatics and Mathematical Modelling, August 2005. IMM-THESIS-2005-58.

[110] Carlos Nunes and Ana Paiva. Automatic Generation of Graphical User Interfaces From VDM++ Specifications. In *ICSEA 2011, The Sixth International Conference on Software Engineering Advances*, pages 399–404, 2011.

[111] Carlos Alberto Loureiro Nunes. Automatic Generation of Graphical User Interfaces From VDM++ Specifications. Master's thesis, Faculty of Engineering of the University of Porto, July 2011.

[112] Vadim Okun and Paul E. Black. Issues in Software Testing with Model Checkers. In Edmund Clarke, Masahiro Fujita, and David Gluch, editors, *Proc. 2003 International Conference on Dependable Systems and Networks (DSN-2003)*, San Francisco, California, June 2003. IEEE Computer Society.

[113] Pierre Bontron Oliver Maury, Yves Ledru and Lydia du Bousquet. Using TOBIAS for the automatic generation of VDM test cases. In J. Fitzgerald J. Bicarregui and P.G. Larsen, editors, *VDM Workshop 3*, Copenhagen, Denmark, July 2002. Part of the FME 2002 conference.

[114] Jan Peleska. Formal Methods for Test Automation – Hard Real-Time Testing of Controllers for the Airbus Aircraft Family. In *Integrating Design and Process Technology, IDPT-2002*. Society for Design and Process Science, 2002.

[115] Ken Piece, John Fitzgerald, Carl Gamble, Yunyun Ni, and Jan F. Broenink. Collaborative Modelling and Simulation — Guidelines for Engineering Using the DESTECS Tools and Methods. Technical report, The DESTECS Project (INFSO-ICT-248134), September 2012.

[116] K. G. Pierce, C. J. Gamble, Y. Ni, and J. F. Broenink. Collaborative modelling and co-simulation with destecs: A pilot study. In *3rd IEEE track on Collaborative Modelling and Simulation, in WETICE 2012*. IEEE-CS, June 2012.

[117] Ken Pierce, John Fitzgerald, and Carl Gamble. Modelling faults and fault tolerance mechanisms in a paper pinch co- model. In *Proceedings of the ERCIM/EWICS/Cyber-*

physical Systems Workshop at SafeComp 2011, Naples, Italy (to appear). ERCIM, September 2011.

[118] Daniel Plagge and Michael Leuschel. Validating Z Specifications using the ProB Animator and Model Checker. In *Integrated Formal Methods (IFM 2007), LNCS 4591*, pages 480–500. Springer-Verlag, 2007.

[119] Nico Plat and Peter Gorm Larsen. An Overview of the ISO/VDM-SL Standard. *Sigplan Notices*, 27(8):76–82, August 1992.

[120] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[121] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, July–December 2004.

[122] Ben Potter, David Till, and Jane Sinclair. *An Introduction to Formal Specification and Z.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 1996.

[123] G. M. Reed and A. W. Roscoe L. A Timed Model for Communicating Sequential Processes. In *Theoretical Computer Science*, pages 314–323, 1988.

[124] Augusto Ribeiro and Peter Gorm Larsen. Proof Obligation Generation and Discharging for Recursive Definitions in VDM. In Jin Song and Huibiao, editors, *The 12th International Conference on Formal Engineering Methods (ICFEM 2010)*. Springer-Verlag, November 2010.

[P125] Augusto Ribeiro, Kenneth Lausdahl, and Peter Gorm Larsen. Run-Time Validation of Timing Constraints for VDM-RT Models. In Sune Wolff and John Fitzgerald, editors, *Proceedings of the 9th Overture Workshop*, number ECE-TT-2 in Technical Report Series, pages 4–16, June 2011.

[126] D. Richard Kuhn and Vadim Okum. Pseudo-Exhaustive Testing for Software. In *SEW '06: Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop*, pages 153–158, Washington, DC, USA, 2006. IEEE Computer Society.

[127] Peter Froome Robin Bloomfield and Brian Monahan. SpecBox: A toolkit for BSI-VDM. *SafetyNet*, Software Engineering for Real Time Systems(5):4–7, 1989.

[P128] John Rohde, Sune Wolff, Thomas Skjødeberg Toftegaardand Peter Gorm Larsen, Kenneth Lausdahl, Augusto Ribeiro, and Poul Ejnar Rovsing. *Towards Green ICT*, chapter 13: Optimizing Energy Usage in Private Households, pages 185–209. River Publishers, 2010.

[129] A. Romanovsky and M. Thomas (Eds.). *Industrial deployment of system engineering methods providing high dependability and productivity*, volume ISBN 978-3-642-33169-5 of *Lecture Notes in Computer Science*. Springer-Verlag, 2012.

[130] Alexander Romanovsky. DEPLOY: Industrial Deployment of Advanced System Engineering Methods for High Productivity and Dependability. *ERCIM News*, 74:54–55, July 2008.

[131] Adriana Sucena Santos. VDM++ Test Automation Support. Master's thesis, Minho University with exchange to Engineering College of Arhus, July 2008.

[132] Linda B. Sherrell and Doris L. Carver. Experiences in Translating Z Designs to Haskell Implementations. *Software Practice and Experience*, 24(12):1159–1178, December 1994.

[133] Colin Snook and Michael Butler. UML-B: Formal modeling and design aided by UML. *ACM Trans. Softw. Eng. Methodol.*, 15(1):92–122, 2006.

[134] J. M. Spivey. *The Z notation: a reference manual*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1992.

[135] Andrzej Tarlecki and Morten Wieth. A Naive Domain Universe for VDM. In Dines Bjørner, C.A.R. Hoare, and Hans Langmaack, editors, *VDM '90 VDM and Z – Formal Methods in Software Development*, pages 552–579. VDM Europe, Springer-Verlag, April 1990.

[136] The-Standish-Group. The Chaos Report. http://www.projectsmart.co.uk/docs/chaos-report.pdf, 1995.

[137] The-VDM-Tool-Group. The Rose-VDM++ Link. Technical report, CSK Systems, January 2008.

[138] The VDM Tool Group. VDM Toolbox API. Technical report, CSK Systems, January 2008.

[139] The VDM Tool Group. The Dynamic Link Facility for VDM++. Technical report, SCSK Corporation, January 2013.

[140] B.D. Theelen, O. Florescu, M.C.W. Geilen, J. Huang, P.H.A. van der Putten, and J.P.M. Voeten. Software/hardware engineering with the parallel object-oriented specification language. In *Proceedings of the ACM-IEEE International Conference on Formal Methods and Models for Codeesign (MEMOCODE)*, pages 139–148, Los Alamitos (USA), 2007. IEEE Computer Society.

[141] S.H. Valentine. Z^{-}, an executable subset of Z. In J.E. Nicholls, editor, *Z User Workshop, York 1991*, Workshops in Computing, pages 157–187. Springer-Verlag, 1992.

[142] P. van der Spek. The overture project: Designing an open source tool set. Master's thesis, Delf University of Technology, August 2004.

[143] P. van der Spek, N. Plat, and C. Pronk. Syntax error repair for a java-based parser generator. *SIGPLAN Not.*, 40(4):47–50, 2005.

[144] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer. In *Formal Methods and Testing*, pages 39–76. Springer-Verlag, 2008. vol. 4949.

[145] Marcel Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. PhD thesis, Radboud University Nijmegen, 2009.

[146] Marcel Verhoef, Peter Gorm Larsen, and Jozef Hooman. Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods*, Lecture Notes in Computer Science 4085, pages 147–162. Springer-Verlag, 2006.

[147] Marcel Verhoef, Peter Visser, Jozef Hooman, and Jan Broenink. Co-simulation of Real-time Embedded Control Systems. In Jim Davies and Jeremy Gibbons, editors, *Integrated Formal Methods: Proc. 6th. Intl. Conference*, Lecture Notes in Computer Science 4591, pages 639–658. Springer-Verlag, July 2007.

[148] M.H.G. Verhoef. Co-simulation enhances the dialogue between design disciplines. In N. Roos, editor, *Bits & Chips*, pages 54 – 55, October 2012. in Dutch.

[149] Sander Vermolen. Automatically Discharging VDM Proof Obligations using HOL. Master's thesis, Radboud University Nijmegen, Computer Science Department, August 2007.

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

[151] Carlos Vilhena. Connecting between VDM++ and JML. Master's thesis, Minho University with exchange to Engineering College of Arhus, July 2008.

[152] Alan Wassyng and Mark Lawford. Lessons learned from a successful implementation of formal methods in an industrial project. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 133–153. Springer Berlin Heidelberg, 2003.

[153] Alan Wassyng and Mark Lawford. Software tools for safety-critical software development. *Int. J. Softw. Tools Technol. Transf.*, 8(4):337–354, August 2006.

[154] Michael Westergaard and Lars Kristensen. The access/cpn framework: A tool for interacting with the cpn tools simulator. In Giuliana Franceschinis and Karsten Wolf, editors, *Proceedings of the 30th International Conference on Applications and Theory of Petri Nets*, pages 313–322. Springer Berlin / Heidelberg, 2009.

[P155] Sune Wolff, Peter Gorm Larsen, Kenneth Lausdahl, Augusto Ribeiro, and Thomas Skjødeberg Toftegaard. Facilitating Home Automation Through Wireless Protocol Interoperability. In *WPMC'09: The 12th International Symposium on Wireless Personal Multimedia Communications*, September 2009.

[156] Jim Woodcock and Jim Davies. *Using Z – Specification, Refinement, and Proof*. Prentice Hall International Series in Computer Science, 1996.

[157] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal Methods: Practice and Experience. *ACM Computing Surveys*, 41(4):1–36, October 2009.

[158] John Wordsworth. *Software development with Z - a practical approach to formal methods in software engineering*. International computer science series. Addison-Wesley, 1992.