

# The Overture Initiative Integrating Tools for VDM

[www.overturetool.org](http://www.overturetool.org)

Peter Gorm Larsen  
Aarhus School of Engineering, Denmark  
e-mail: [pgl@iha.dk](mailto:pgl@iha.dk)

Miguel Ferreira  
Software Improvement Group, Netherlands  
e-mail: [m.ferreira@sig.nl](mailto:m.ferreira@sig.nl)

Kenneth Lausdahl  
Aarhus School of Engineering, Denmark  
e-mail: [kel@iha.dk](mailto:kel@iha.dk)

Nick Battle  
Fujitsu, UK  
e-mail: [nick.battle@uk.fujitsu.com](mailto:nick.battle@uk.fujitsu.com)

John Fitzgerald  
Newcastle University, UK  
e-mail: [john.fitzgerald@ncl.ac.uk](mailto:john.fitzgerald@ncl.ac.uk)

Marcel Verhoef  
Chess, Netherlands  
e-mail: [Marcel.Verhoef@chess.nl](mailto:Marcel.Verhoef@chess.nl)

## Abstract

Overture is a community-based initiative that aims to develop a common open-source platform integrating a range of tools for constructing and analysing formal models of systems using VDM. The mission is to both provide an industrial-strength tool set for VDM and also to provide an environment that allows researchers and other stakeholders to experiment with modifications and extensions to the tools and language. This paper presents the current status and future vision of the Overture project.

## 1 Introduction

Formal methods are mathematical techniques for the modelling, analysis and development of software and systems [WLB<sup>+</sup>F09]. Their use is motivated by the expectation that, as in other engineering disciplines, performing an appropriate mathematical analysis can promote early discovery of defects and contribute directly to the increased reliability and robustness of a design.

The Vienna Development Method (VDM)<sup>1</sup> is one of the most mature formal methods [Jon99, FL09]. The method focuses on the development and analysis of a system model expressed in a formal language. The language's formality enables developers to use a wide range of analytic techniques, from testing to mathematical proof, to verify the consistency of a model and its correctness with respect to an existing statement of requirements. The VDM modelling language has been gradually extended over time. Its most basic form (VDM-SL), standardised by ISO [LH<sup>+</sup>96] supports the modelling of the functionality of sequential systems. Extensions support object-oriented modelling and concurrency [FLM<sup>+</sup>05], real-time computations [MBD<sup>+</sup>00] and distributed systems [VLH06]. There is consequently a need to provide a common basis for supporting the analytic tools covering all these extensions.

Currently, the most feature-rich tool available is VDM-Tools [ELL94, FLS08], a commercial product which includes syntax- and type-checking facilities, an interpreter to support testing and debugging of models, proof obligation generators that produce formal specifications of integrity checks that can not be performed statically, and code generators. From the perspective of modern Integrated Development Environments (IDEs), VDMTools has some weaknesses, including a relative lack of extensibility.

The Overture project aims to provide at least as much functionality as VDMTools, but built on an open and extensible platform based on the Eclipse framework. An alternative VDM tool, VDMJ [Bat09], provides many of the features of VDMTools. It is small, relatively fast, pure Java and open source but has only a command-line interface. These characteristics make it suitable for integration with the Eclipse IDE and allow it to be distributed freely with the Overture project.

This paper provides a report on the current state of Overture, and plans for its future development. It provides a short introduction to Eclipse (Section 3) and to the Overture plugin architecture (Section 4). Section 4 then provides a short introduction to the features that currently exist in Overture. Section 5 explains the development strategy for the Overture tool. Finally Section 6 describes future development plans.

## 2 VDM

A VDM model describes data and functionality. User-defined data types are built from base types and constructors defining compositions such as record and union types, and collections such as sets, sequences and mappings. Types may be constrained by invariants which are arbitrarily complex logical predicates that all elements of the defined type respect. Persistent data is modelled as state variables, again restricted by invariants.

Functionality in VDM is expressed as operations over state variables or as referentially transparent functions over the

<http://doi.acm.org/10.1145/1668862.1668864>

<sup>1</sup><http://www.vdmportal.org/>

available data types. Functionality can be defined explicitly by means of simple algorithms or implicitly by means of postconditions characterising the required relationship between the results and the inputs. In either style, the assumptions under which a function or operation may be called are recorded as logical preconditions. Since functionality can be defined implicitly, and given the expressiveness of the logic for defining functionality, VDM models are not necessarily executable. There is, nevertheless, a large executable subset of the modelling language, allowing validation of models by dynamic testing.

The VDM modelling language has a formal semantics, allowing users to conduct more sophisticated static analyses than regular syntax and type checking. For example, it is possible to analyse models to detect violations caused by potential misuse of partial operators (e.g. indexing out of range) or by potential violation of invariants. The conditions for a consistent model are recorded as *proof obligations* which are logical predicates that can be proven to hold using the mathematical semantics of the language. Such proof obligations cannot be checked completely automatically in general because of the expressiveness of the modelling language, but they can be proved using modern proof technology or manual inspection.

The VDM extensions that support object-oriented structuring use concepts common to UML and so there is scope for integrating the tools that support formal analysis of VDM models with graphical views of the models via UML class diagrams, for example. Other VDM extensions to support concurrency, distribution and real-time systems open the possibility of providing tools that allow exploration of complex run-time behaviours, again via graphical representations.

### 3 Eclipse

Eclipse is intended to serve as a common platform that “blends separately developed tools into a well designed suite”<sup>2</sup>. The Eclipse platform user interface is based on *editors*, *views*, and *perspectives*. A perspective defines an appropriate collection of views, their layout, and applicable actions for a given user task. A view is a workbench part that can navigate a hierarchy of information or display properties for an object. Figure 2 shows a screenshot of the Overture IDE built on Eclipse.

## 4 Overture Architecture and Features

Overture uses a plug-in architecture consisting of components that supply core functionality and components that interact directly with the user through the Eclipse GUI. The dependencies between these components are quite complex but it is notable that most of the components depend on the abstract syntax trees (ASTs) generated from the parsing of

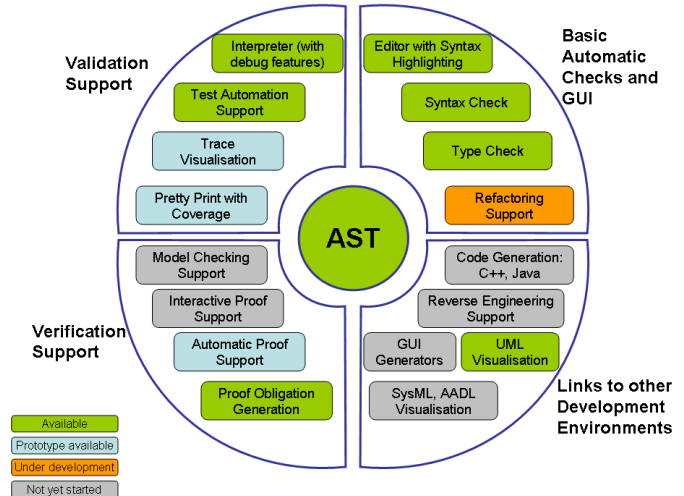


Figure 1: Overture Tool Components

a VDM model. The current and envisaged Overture components are shown in Figure 4. The remainder of this section describes each of the main components in more detail.

### 4.1 Overture Parsers and ASTs

VDM models are input in plain text using a standard concrete syntax. A parser is therefore required to transform the textual model into an AST that all plug-ins may use to access the model. The parser developed for Overture is generated directly from the VDM grammar using the JFLEX and BYACC/J tools. The VDM tools that Overture seeks to integrate also have their own legacy parsers. For example, as a matter of convenience, the VDMJ parser is used to mark the syntax problems highlighted in the Overture editor plug-in. This parser also permits VDM source to be embedded within a  $\text{\LaTeX}$  document.

### 4.2 Overture IDE

The basic Overture IDE [MT09] is illustrated in Figure 2. The *Explorer view* (on the left of the figure) allows the user to create, select, and delete Overture projects and navigate between the files in these projects. The panel to the right of the Explorer is the *editor area*. Since there are currently several dialects of VDM, the particular editor that opens in the panel depends on the dialect of VDM being used in the current project. The *Outline view*, to the right of the editor, summarises the content of the file currently selected in the editor, displaying declared classes, instance variables, values, types, functions, operations etc. The *Problems view*, shown here at the bottom of the window, displays information generated by Overture, such as warnings and errors relevant to the current project.

Most of the other features of the workbench, such as the menu or the toolbar are similar to those of other familiar applications. There is also a special menu with Overture-specific functionality. One convenient feature is a toolbar of

<http://doi.acm.org/10.1145/1668862.1668864>

<sup>2</sup><http://help.eclipse.org/>

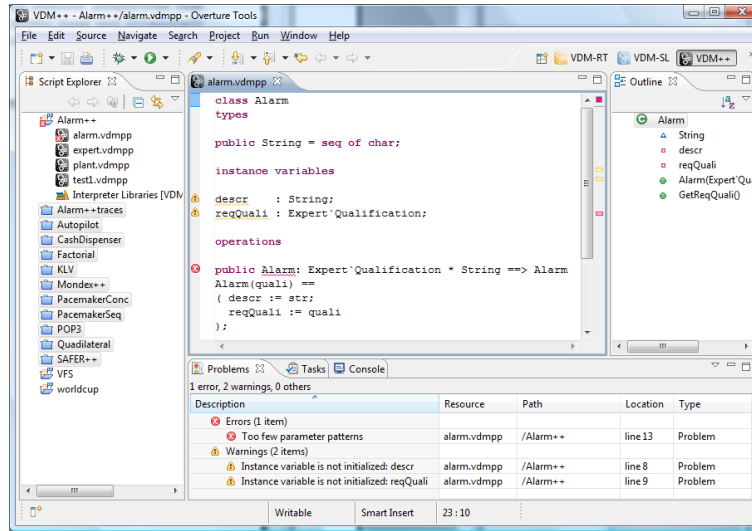


Figure 2: Overture IDE

shortcuts to switch between different perspectives that appears on the right side of the screen; these vary dynamically according to context and history.

### 4.3 Type Checking

Static type checking is performed automatically as a model is entered via the Overture editor. The simplest errors are often typographical in origin and are readily detected, for example entering the type of a parameter incorrectly, so that the type name used cannot be found. More subtle errors can indicate semantic problems with the model, such as when trying to compose mappings where the domain and range of the operands do not match. Warnings are also generated for unused variables, obviously unreachable code, and so on.

Overture supports type checking with “possible” semantics. An expression is considered type correct if there is an assignment of values that yields a result of the correct type, regardless of whether there are other assignments that could be type incorrect. For example, consider a function with a return type composed of natural numbers constrained by an invariant to be less than 10. If the function has a body that returns a real number, this is considered correct by the checker because a real number is possibly a natural number less than 10. There is a risk that, at run time, the real value is actually not a whole number, or is less than zero or greater than 10. The VDM run-time system performs *dynamic type checking* which catches such errors. By contrast “definite” semantics for type checking raises errors in all cases where there might be a run-time error, regardless of whether this is guarded against.

### 4.4 Proof Obligation Generation

As indicated above, static type correctness does not guarantee freedom from run-time errors. The formality of VDM’s DOI: 10.1145/1668862.1668864

semantics makes it possible to generate, for a model, a set of logical assertions that must be true if that model is to be dynamically type correct and semantically consistent. We call these assertions *proof obligations*. Both VDMTools and VDMJ provide tools that automatically generate proof obligations for a given model [Ber97].

Proof obligations are generated for a range of properties relating to the internal consistency of a model. These include obligations checking for potential run-time errors caused by misapplication of partial operators, consistency of results with postconditions and termination of recursion [Rib08]. In our type checking example from Section 4.3, proof obligations would be generated that state that, whenever a real number is returned as a result of the function, the real value satisfies the invariant of the return type. The obligations will take account of the context in the model, so for example a result returned from within nested *if* or *else* clauses would yield a proof obligation that was qualified with the same tests as the *if-expressions* which lead to the returned result.

Proof obligations can be verified to different levels of confidence using a range of techniques including manual inspection and formal mathematical proof.

### 4.5 Proof Support

The highest level of confidence in the validity of proof obligations can be gained by constructing machine-verifiable formal proofs. The production of such proofs manually is a complex and error-prone process, but it is susceptible to automated support. There is currently no VDM-specific theorem prover, but it is possible to exploit off-the-shelf proof technology. A model can be translated into a theory (in the prover’s language) that captures its semantics, and the proof obligations can be set as proof commands. Along with the translation, the connection to the theorem prover is complemented with

<http://doi.acm.org/10.1145/1668862.1668864>

a set of VDM-specific theorems and proof tactics.

The viability of this approach to automated verification in VDM was established some time ago [AF97]. More recently, an automated proof support system was developed in the Overture context [Ver07]. A VDM prototype model of a translator tool was developed, converting VDM to HOL. The interaction between the proof obligation generators, the translator tool (automatically generated from the VDM++ prototype) and HOL was later implemented in a Java proof component that integrates all this functionality. Integrating the proof component in Eclipse is now a question of improving interaction and usability of a tool which design is stable. This is a part of the future work in Overture.

#### 4.6 Interpreter/debugger

Automatic verification of proof obligations is an extremely powerful analytic technique, but it is also time consuming, even with tool support. However, a great deal of value can still be gained from less formal analysis of a model. For example, proof obligations may be discharged by informal inspection. The simplest level of validation for a VDM model is to run a simulation using an interpreter, and in the case of problems, to step through the evaluation of expressions using a debugger.

Overture provides an interpreter for all of the VDM language dialects, allowing the evaluation to be stepped through using the normal Eclipse debugging views, i.e. setting breakpoints, inspecting local and state variables, call stack etc. Typically, a model is provided with abstract data to represent the world being modelled, and functions or operations are evaluated over that data to produce abstract results representing the outcome of the action performed by the model. If the interpreter results are as expected, confidence in the model is strengthened. If the results are not as expected, the debugger can be used to find out why.

The interpreters in VDMTools and VDMJ both keep track of how much of the model is covered during an evaluation. This allows a report to be generated highlighting the parts of the model that have not been exercised by a set of tests. The intention is to integrate this with the Overture editor for immediate feedback in the IDE.

#### 4.7 Combinatorial Testing

The combinatorial testing feature supports the automatic execution of a large number of test cases generated from templates provided in the form of trace definitions added to a VDM++ model. Trace definitions are defined as regular expressions describing possible sequences of operation calls, and are conceptually similar to the input provided to model checkers. A plug-in enabling trace expansion, evaluation and inspection is developed in two steps: first as a VDM model prototype and then as an optimised version with direct integration into VDMJ to speed up the evaluation process of large test cases [San08, LLB09].

DOI: 10.1145/1668862.1668864

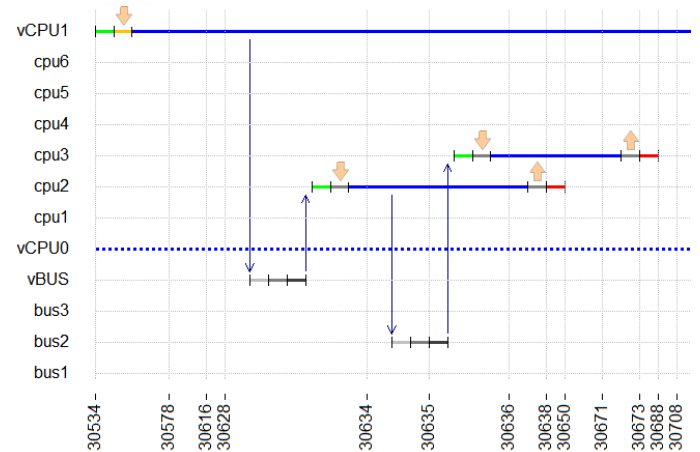


Figure 3: Overture Realtime Log Viewer

#### 4.8 Bi-directional UML mapping

The UML bi-directional mapping [LLL09] feature supports an automatic connection between the object-oriented dialect of VDM (VDM++) and UML 2.1, allowing models developed in both notations to be kept consistent as they evolve. The feature supports two forms of link: between UML class diagrams and VDM models, and between UML sequence diagrams and VDM trace statements.

The link between class diagrams and VDM models relates the static structure of the two models. The link is bi-directional so that modifications to the class structure in one models are reflected automatically in the other.

The link between UML sequence diagrams and VDM trace statements enables a test scenario to be specified in a UML sequence diagram and then transformed into a VDM trace statement which enables the Combinatorial Testing feature to evaluate the test scenario specified in the UML sequence diagram.

#### 4.9 Realtime Log Viewer

The VICE dialect of VDM allows the description of real-time concurrent processes which can be distributed over a virtual architecture of CPUs and buses. The interpreter for this dialect produces a time-stamped log file showing every internal event (e.g. operation request, activation and completion), every message handling event on the buses and thread events such as creating, swapping in and out and termination. The *Realtime Log Viewer* plug-in for Overture can read the log file and produce a visualisation of the execution in terms of the CPUs and BUSES as shown in Figure 3. It is possible to see both how busy the components are, and the details of each thread swap in the given scenario. It is also possible to visualize violations of formally specified timing requirements directly [FLTV07].

<http://doi.acm.org/10.1145/1668862.1668864>

## 5 Overture Development

The Overture project's main development technologies are Java and VDM itself. The development code, maintained under a Subversion repository at SourceForge<sup>3</sup>, is divided into:

**core** The core components such as VDMJ, proof obligation translation and Realtime log viewer

**ide** The Eclipse plug-ins which make up the Overture Editor.

**tools** Additional build tools e.g. Maven Eclipse build plug-ins and Maven VDMTools integration plug-in. Tools such as ASTGen are also located under tools.

The core components are either written directly in Java or in VDM++ and then code generated to Java. The Apache Maven<sup>4</sup> tool is used for project management and build automation. The Maven build tool has been extended with plug-ins to enable building Eclipse plug-ins, enabling an easy integration of VDMTools code generation as a part of the Maven build life cycle.

## 6 Future Plans

The Overture community plans to further develop the platform by extending existing functionality, integration with other formal methods tools and by integration of more Eclipse technology to improve ease of use. Regarding other tools, work on the link to JML [LC05] and Alloy [Jac02] is in progress. Work to build a VDM-specific theorem prover is also planned. Regarding better Eclipse integration, style consistency for VDM (with CheckStyle) and task management (with Mylyn) is being considered.

A priority for the VDM community in recent years has been successful industrial deployment [FL07] and this influenced the priorities for the development of modelling facilities including support for object-orientation, real-time and concurrency as well as the integration of formal modelling into industrial development processes. The European Framework 7 project DESTecs (Design Support and Tooling for Embedded Control Software, [www.destecs.org](http://www.destecs.org)) starting in early 2010, will develop methods and tools that combine continuous time models of systems (in tools such as 20-Sim) with discrete event controller models in VDM through co-simulation [VVHB07, Ver08]. The approach is intended to encourage collaborative multidisciplinary modelling, including modelling of faults and fault tolerance mechanisms.

### Acknowledgments

We are grateful to Thomas Christensen, Jens Kielsgaard Hansen, Hans Kristian Lintrup, Hugo Macedo, David Møller, Paul Mukherjee, Jacob Porsborg Nielsen, Nico Plat, Augusto

Ribeiro, Shin Sahara, Adriana Sucena Santos, Pieter van der Spek, Christian Thillermann, Sander Vermolen, Carlos Vilhena and all the other contributors who have helped to create the Overture platform and tools.

## References

- [AF97] Sten Agerholm and Jacob Frost. Towards an integrated case and theorem proving tool for vdm-sl. In John 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 278–297. Springer-Verlag, September 1997. ISBN 3-540-63533-5.
- [Bat09] Nick Battle. VDMJ User Guide. Technical report, Fujitsu Services Ltd., UK, 2009.
- [Ber97] 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.
- [ELL94] 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.
- [FL07] J. S. Fitzgerald and P. G. Larsen. Triumphs and Challenges for the Industrial Application of Model-Oriented Formal Methods. In T. Margaria, A. Philippou, and B. Steffen, editors, *Proc. 2nd Intl. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2007)*, 2007. Also Technical Report CS-TR-999, School of Computing Science, Newcastle University.
- [FL09] 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.
- [FLM+05] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object-oriented Systems*. Springer, New York, 2005.

<sup>3</sup><http://sf.net/projects/overture>

<sup>4</sup><http://maven.apache.org/>

- [FLS08] John Fitzgerald, Peter Gorm Larsen, and Shin Sahara. VDMTools: Advances in Support for Formal Modeling in VDM. *Sigplan Notices*, 43(2):3–11, February 2008.
- [FLTV07] 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.
- [Jac02] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.
- [Jon99] 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.
- [LC05] Gary T. Leavens and Yoonsik Cheon. Design by Contract with JML. Draft, available from [jmlspecs.org](http://jmlspecs.org), 2005.
- [LH<sup>+</sup>96] P. G. Larsen, B. S. Hansen, et al. Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language, December 1996. International Standard ISO/IEC 13817-1.
- [LLB09] Peter Gorm Larsen, Kenneth Lausdahl, and Nick Battle. Combinatorial Testing for VDM++. In *Submitted for publication*, July 2009.
- [LLL09] Kenneth Lausdahl, Hans Kristian Agerlund Lintrup, and Peter Gorm Larsen. Connecting UML and VDM++ with Open Tool Support. In *Formal Methods 09*, November 2009.
- [MBD<sup>+</sup>00] 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](http://www.vdmportal.org).
- [MT09] 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.
- [Rib08] Augusto Ribeiro. An Extended Proof Obligation Generator for VDM++/OML. Master’s thesis, Minho University with exchange to Engineering College of Aarhus, July 2008.
- [San08] Adriana Sucena Santos. VDM++ Test Automation Support. Master’s thesis, Minho University with exchange to Engineering College of Aarhus, July 2008.
- [Ver07] Sander Vermolen. Automatically Discharging VDM Proof Obligations using HOL. Master’s thesis, Radboud University Nijmegen, Computer Science Department, August 2007.
- [Ver08] Marcel Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. PhD thesis, Radboud University Nijmegen, 2008. ISBN 978-90-9023705-3.
- [VLH06] 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*, pages 147–162. Lecture Notes in Computer Science 4085, 2006.
- [VVHB07] 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.
- [WLBF09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal Methods: Practice and Experience. *ACM Computing Surveys*, 41(4):1–36, 2009.