Enhancing System Realisation in Formal Model Development

PhD Dissertation

Peter W. V. Tran-Jørgensen

July, 2016

AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

ISBN: 978-87-7507-375-7
DOI: 10.7146/aul.152.125

Enhancing System Realisation in Formal Model Development

A Dissertation
Presented to the Faculty of Science and Technology
of Aarhus University
in Partial Fulfilment of the Requirements for the
PhD Degree

by
Peter W. V. Tran-Jørgensen
July 16th, 2016

Abstract

Software for mission-critical systems is sometimes analysed using formal specification to increase the chances of the system behaving as intended. When sufficient insights into the system have been obtained from the formal analysis, the formal specification is realised in the form of a software implementation. One way to realise the system's software is by automatically generating it from the formal specification – a technique referred to as code generation. However, in general it is difficult to make guarantees about the correctness of the generated code – especially while requiring automation of the steps involved in realising the formal specification. This PhD dissertation investigates ways to improve the automation of the steps involved in realising and validating a system based on a formal specification. The approach aims to develop properly designed software tools which support the integration of formal methods tools into the software development life cycle, and which leverage the formal specification in the subsequent validation of the system. The tools developed use a new code generation infrastructure that has been built as part of this PhD project and implemented in the Overture tool – a formal methods tool that supports the Vienna Development Method. The development of the code generation infrastructure has involved the re-design of the software architecture of Overture. The new architecture brings forth the reuse and extensibility features of Overture to take into account the needs and requirements of software extensions targeting Overture. The tools developed in this PhD project have successfully supported three case studies from externally funded projects. The feedback received from the case study work has further helped improve the code generation infrastructure and the tools built using it.

Resumé

Software til mission kritiske systemer analyseres nogle gange ved brug af formel specifikation for at øge chancerne for at systemet fungerer efter hensigten. Når den nødvendige indsigt i systemet er opnået via den formelle analyse, så realiseres den formelle specifikation i form af en software-implementering. En måde at realise systemets software på er ved automatisk at generere denne ud fra den formelle specifikation – en teknik som kaldes kodegenerering. Generelt set er det dog udfordrende at kunne stille garanti for korrektheden af den genererede kode – særligt hvis der også kræves automatisering af de trin der er involveret i at realisere den formelle specifikation. Denne ph.d.-afhandling undersøger måder hvorpå automatiseringen af de trin der er involveret i at realisere og validere et system, baseret på en formel specifikation, kan forbedres. Tilgangen har til formål at udvikle vel-designede software-værktøjer som: understøtter integrationen af værktøjer til formelle metoder ind i livscyklussen for softwareudvikling, samt gøre nytte af den formelle specifikation i den efterfølgende validering af systemet. De udviklede værktøjer anvender en ny kodegenererings-infrastruktur, der er blevet udviklet i dette ph.d.-projekt, og implementeret i Overture – et værktøj som understøtter den formelle udviklingsmetode, Vienna Development Method. Udviklingen af kodegenererings-infrastrukturen har involveret et redesign af Overtures softwarearkitektur. Denne nye arkitektur fremmer genanvendelsen og udvidbarheden af Overture for at imødekomme de behov og krav som stilles til software-udvidelser til dette værktøj. Værktøjerne, udviklet i dette ph.d.-projekt, er blevet anvendt i forbindelse med tre casestudies fra eksternt finansierede projekter. Feedbacken fra dette casestudy-arbejde har yderligere bidraget til forbedringen af kodegenererings-infrastrukturen og de værktøjer, der er udviklet ved brugen af denne.

Acknowledgements

There are several people I would like to thank for their collaborations and support over the course of my PhD project. Without them, the work reported in this dissertation would not have been possible.

First, I would like to express my sincere gratitude to my academic supervisor, Peter Gorm Larsen for his excellent advice and collaboration in the pursuit of my PhD degree. Peter has had a huge influence on my personal development and the way I have evolved as a researcher. For that, I am deeply grateful.

I would like to thank Gary Leavens for helping arrange my stay abroad at the University of Central Florida, for inviting me as a visiting scholar, and for his collaboration on parts of the work reported in this dissertation. I am also thankful to Foundation Idella for partly funding the trip to Florida.

I am grateful to Nick Battle with whom I have enjoyed working on developing the Overture tool. I deeply appreciate all the time Nick has spent helping me, and all the efforts he has put into reviewing my work, including this dissertation, and providing me with excellent feedback. I would also like to thank Marcel Verhoef for reviewing this dissertation and providing me with valuable feedback on my work.

Thanks to the current and former members of the Software Engineering Group at Aarhus University with whom I had many great times and fruitful collaborations, in particular José Antonio Esparza Isasa, Luís Diogo Couto, Kenneth Lausdahl, Sune Wolff, Morten Larsen, Stefan Hallerstede, Martin Peter Christiansen, Joey Coleman, Victor Bandur, Claus Ballegaard Nielsen, Giorgos Kanakis, Miran Hasanagić, Casper Thule Hansen, and Rene Nilsson. I owe a special thanks to my great friend and excellent colleague José who is always helpful, positive and great to be around. It has been a pleasure working together with José throughout my PhD project, and having him as a colleague. I also owe a special thanks to my former office mate Luís for our great collaborations, which have resulted in a large part of the work presented in this dissertation. I would also like to thank Luís for his valuable feedback on earlier versions of some of the chapters of this dissertation. I am also particularly grateful to Kenneth for all the time and effort he has spent giving me excellent advice on technical matters and for all the things he has taught me about Overture.

Finally, I would like to express my deepest love and gratitude to my beautiful wife, Cecilia Tran-Jørgensen, and our daughter Evelyn Nhi Tran-Jørgensen who are the best things that have ever happened to me. I owe my most sincere gratitude to my wife for all the support she has given me throughout my PhD studies, and for the excellent and hard work she is doing as a mother to raise our daughter. I love my wife and daughter more than anything.

Contents

Part I

Summary

1

Introduction

"Program testing can be used to show the presence of bugs, but never to show their absence! "

–Edsger W. Dijkstra, Notes On Structured Programming, April, 1970

1.1 Context

According to Dijkstra's statement, traditional approaches to software development alone are not sufficient to guarantee that a software system will behave as intended. Although Dijkstra's statement dates back to 1970, it still remains true as of today. In software development it is common to anticipate the presence of "bugs" when the software is released. For systems where problems with the software do not impose a significant risk to the user, it might be acceptable to simply correct a problem once it has been identified, and release a new version of the software. For some systems this might be the most viable approach to developing software.

For another type of system called *mission-critical systems*, system failure may have fatal consequences, and correction of problems, once they have been identified, might not be an option. More specifically, a mission-critical system is a type of system whose failure may lead to loss of human lives or huge financial costs. Examples of such systems range from everyday objects such as smartphones and modern televisions to safety critical systems such as cars and aeroplanes. Common to many of these systems is that they rely on software to make central control decisions. Naturally, this makes the success of these systems dependent on the correct functioning of the software that controls them.

Mission-critical systems must be carefully analysed and developed in order to obtain a high degree of confidence in the correct functioning of these systems. One way to develop mission-critical systems is by use of formal methods – a set of mathematically-based techniques that can be employed to analyse the system under development. This approach to system development is motivated by the expectation that mathematical analysis supports the development of robust systems that function correctly. Several examples exist that document the successful application of formal methods in industrial projects [86].

Eventually the system is realised using concrete implementation technologies. This process involves translation of formal specification into hardware and software that, when put together, form the system realisation or some part of it. This PhD project is mostly concerned with formal specifications that are used to model behaviour of software systems, and the dissertation therefore refrains from describing formal methods used to analyse hardware systems.

This dissertation provides insight into the research conducted during a PhD project on system realisation in the context of formal model development. Input for the research is based on literature survey and by working on externally funded projects. The PhD project takes a starting point in the challenges of transitioning from system analysis to the implementation phase, where the formal specification is realised. Software development practices that are gaining ground are studied in order to reflect on ways to improve traditional formal model development. The research conducted during the PhD project is focused on better leverage of insights obtained during system analysis in the subsequent phases of system development.

Following this section, the central subject *development of software systems using formal modelling* is described in section 1.2. Next, the motivation underlying the PhD project is described in section 1.3. This is followed by a description of the research method employed in the project in section 1.4. Subsequently, the research objectives are described in section 1.5. Next, the criteria used to evaluate the results produced during the project are presented in section 1.6. An overview of the publications produced during the PhD is given in section 1.7, and finally a reading guide for this dissertation completes this chapter in section 1.8.

1.2 Development of Software Systems using Formal Modelling

Formal methods serve different purposes and assume various levels of rigour. Heavy-weight formal methods are concerned with formal verification of system properties, for example, by formal proof. Light-weight formal methods, on the other hand, are often used for informal validation, for example, by animation of the system specification. Although light-weight analysis such as animation does not serve as a proof of correctness, it might still help the developer to obtain the necessary insights into the system under development.

A formal specification can be realised through refinement [85], which is a step-wise process of translating a formal specification into code using semantics-preserving rules. Alternatively, a formal specification may be realised using code generation. The idea is for the generated code to be a single-step refinement of the formal specification through code generation translation rules. Code generation is used to (1) reduce the effort needed to realise the system through tool automation and (2) avoid introducing problems in the implementation due to manual translation of formal specification into code. Despite the advantages achieved via tool automation it is generally difficult to make guarantees about the correctness of the generated code. This is especially a problem in mission-critical system development, where system failure may have severe consequences.

The remainder of this section gives examples of formal methods that take different approaches to system realisation in formal model development. The Vienna Development Method (VDM) [10, 53, 54, 32] and the Java Modeling Language (JML) [13, 67] constitute examples of formal methods, which have been used in this PhD project to study the development of mission-critical systems. VDM has been used to analyse systems during the early phases of development and study challenges inherent to realising a formal specification. Additionally, JML has been used to bridge the gap between an abstract system specification and the implementation of it. To consider system realisation in formal modelling in breadth the B-Method [1] – a refinement-based formal method – is presented in section 1.2.1. VDM and JML are described in section 1.2.2 and section 1.2.3.

1.2.1 The B-Method

The B-Method is based on B – a formal method that uses abstract machine notation to describe software systems. One of the goals of B is to support the correct implementation of a formal specification through proof-based refinement. In consequence, B operates at a lower level of abstraction compared to (say) VDM. The final refinement of a B specification, expressed using a subset of B called B0, can be automatically translated into code [12]. B is tool supported by the Atelier B tool [14].

Event-B [2] is a formal method derived from B, which uses a simpler notation. Whereas B is concerned with the correct construction of software, the purpose of Event-B is to model the entire system (the software, the hardware and the environment) [49]. Event-B is tool supported by the Rodin tool [3].

1.2.2 VDM

VDM is one of the most mature formal methods with a long history of developing computer-based systems. The development of VDM was originally carried out at IBM in Vienna in the 1970s to support the development of a compiler for PL/1 [45]. Specifications written in VDM can be analysed by formal proof [9] or informal animation [66]. VDM is a Design-by-Contract (DbC) language that uses contract-based concepts such as invariants and pre- and post conditions to specify system behaviour [72]. A VDM specification can be validated against these "contracts" by animation of the VDM specification. VDM is tool supported by the Overture tool [62], VDMTools [35], VDMJ [7] and ViennaTalk [76]. A VDM specification can be realised through refinement or by direct translation into code using the code generation features of VDMTools, Overture or ViennaTalk. Related work on code generation for VDM is described in section 3.6.

1.2.2.1 The VDM Dialects

VDM has evolved into the three dialects VDM-SL [5], VDM++ [34], and VDM-RT [66]. VDM-SL is an ISO standardised sequential specification language that supports description of data and functionality. As an informative annex to the standard, modules have been added to the language to support organisation of data and functionality that can be imported or exported between modules. Every type in VDM-SL is passed by value, i.e. as a *deep copy*, when the value is passed as an argument, returned as a result or appear on the right-hand-side of an assignment. In consequence, *aliasing* can never occur in a VDM-SL specification.

VDM++ adds object-orientation and concurrency to VDM. Object values have reference semantics, i.e. they are passed as *shallow copies*, and therefore aliasing can occur. Access to shared resources is specified using permission predicates and mutex constraints. A permission predicate specifies a guard condition that must hold for an operation to be executed, while a false guard condition blocks the calling thread. Permission predicates may refer to instance variables or special self-contained variables called history counters, which record the number of times each operation has been requested, activated or completed.

VDM-RT further extends VDM++ with support for modelling of real-time and distributed hardware architectures. System architecture is modelled in the **`system`** class using special class constructs for Central Processing Units (CPUs) and buses. An object deployed on one CPU can invoke a function or operation on an object deployed on a different CPU, which causes data to be transmitted across the connecting bus. Deployable objects are stored using **`public static`** instance variables in the **`system`** class. In addition to the user-defined CPUs there exists a special virtual CPU, which by default runs infinitely fast without affecting system time. Objects that are not deployed to one of the user-defined CPUs get deployed on the virtual CPU. The virtual CPU is typically used for objects that represent elements that are external to the system such as the environment. Every CPU is connected to the virtual CPU via the virtual bus. The VDM-RT interpreter maintains a global notion of time that can be referred to using the **`time`** keyword. System time increases by a default number of nanoseconds when VDM constructs are executed. This default increase in time can be overruled using the **`duration`** and **`cycles`** statements, which allow time delays to be specified as an absolute time measure or relative to the CPU.

1.2.2.2 Analysis Techniques for VDM

Various techniques exist to support the analysis of VDM specifications. VDMUnit is a unit testing library that supports regression testing of VDM++ and VDM-RT specifications in a way similar to that of JUnit [58]. The VDMUnit library provides different classes to support writing of test suites, which can be executed using the VDM interpreter.

Proof obligation analysis complements static analysis for situations where it cannot be statically determined whether the VDM specification suffers from inconsistencies that may lead to runtime errors [35]. The potential presence of such inconsistencies entails the generation of proof obligations – predicates that must be proven to hold in order to ensure that the specification is internally consistent.

In VDM test automation can be achieved using combinatorial testing [63, 64]. This technique is used to identify problems related to the contracts of a VDM specification such as a missing pre condition. Combinatorial testing uses a trace definition, which is a kind of pattern, to describe test collections. The combinatorial test generator expands the trace definition into tests that are executed independently of one another against the VDM specification.

VDM offers several constructs to define traces. The **`let`** binding introduces trace variables but does not contribute additional tests per se. The bar operator ("`|`") expands to all alternatives for the tests of its child nodes. The **`let be st`** binding produces a test for each binding introduced by this construct. The non-deterministic choice operator ("`||`") expands to all possible orderings of the tests of its child nodes. The repetition operator (e.g. `op(x){1,2}`) is used to repeat tests a specified number of times. Terms separated by "`;`" expand to the sequencing of the terms.

At the end of the expansion every test is a list of variable assignments and operation or function calls. A test that does not violate any constraints is considered a `PASS`. A test that violates a constraint directly when an operation is called from the test (i.e. from the outermost level) is considered `INCONCLUSIVE`, since the test generation may be at fault. For other situations the violation of a constraint is considered a `FAIL`.

Combined execution of VDM and Java is enabled via Overture's Java bridge [73]. This feature allows functions and operations, defined using the **`is not yet specified`** construct, to be executed in an external environment. Whenever the Overture interpreter encounters this construct it tries to find and execute the corresponding Java method, which must be named according to a certain convention. When a VDM function or operation is invoked via the Java bridge, the arguments are passed as `Value` instances to the corresponding method on the Java side. The `Value` classes form part of a runtime library that provides Java-based implementations of VDM values. Similarly, a value returned to the interpreter must also be represented as a `Value` instance. The Java bridge can, for example, be used to build a User Interface (UI) on top of a VDM specification in order to visualise the specification to a stakeholder who does not understand VDM.

1.2.3 JML

JML is a DbC language that is used for detailed specification of Java classes and interfaces. In JML DbC elements are written as specialised source code comments that are added to the Java program. Therefore, JML has the advantage that the Java program can be executed without the overhead of checking the JML contracts, if desired. JML provides several constructs to support contract-based specification. For example, in JML pre- and post conditions are specified using the **`requires`** and **`ensures`** keywords, respectively. Furthermore, a method that is marked as **`pure`** is not allowed to have write effects. Such methods can be used in JML specifications.

VDM and JML are both DbC languages with the difference that they operate at different levels of abstraction: VDM is used for abstract system specification, whereas JML is used for specification at the implementation level. Therefore, these technologies can be regarded as complementary. JML annotated Java programs can be analysed statically or dynamically using tools such as OpenJML [15]. In this PhD project the OpenJML runtime assertion checker has been used to check the specification of Java programs by execution of JML annotations. An overview of different JML tools is provided in [13].

1.3 Motivation

The importance of robust tool support and automation in formal model development is recognised by practitioners and researchers of the community. The survey in [86, section 5.2] conducted by Woodcock et al. draws the conclusion that *"it appears almost inconceivable that an industrial application would now proceed without tools"*. The survey further states that existing tool support is generally not considered *"to be rugged enough for wide-scale application"*.

In [60, section 2] Knight et al. emphasise the need for better integration of formal methods tools into the software development life cycle. The authors state that this is necessary in order for formal methods to contribute to cost-effective development of high-quality software. The need for better tool integration is further supported by Wassyng et al. in [83, section 6.1]. The authors formulate general high-level requirements to support development of safety-critical systems. In particular the authors vision that tools should form a comprehensive and integrated tool suite that *"shall provide automated support for all phases of the software development lifecycle"*.

Improved integration of formal methods into the software development life cycle can be achieved using techniques that support transfer of knowledge and insights between the different phases of software development. Code generation is an example of one such technique, which has the potential to improve development productivity in formal model development [86, section 3.2]. Improvements in productivity due to use of code generation is further demonstrated by Banci et al. in [6, section 6]. The authors report experiences from the automotive industry on the combined use of Model-Based Development (MBD) and code generation. This approach is regarded by the authors to be effective due to the short time it takes to produce a working prototype.

The main motivation of this PhD project is the desire to enhance techniques and tool support used to realise formal specifications. Achieving this requires a deep understanding of the challenges that need to be overcome in order to ensure better transfer of knowledge and insights between the different phases of the software development life cycle.

1.4 Research Method

The research conducted in this PhD project is driven by challenges related to realisation of formal specifications. Identification of these challenges was done through literature survey and by analysing case studies that use formal specification.

When a challenge of interest was identified existing literature on the topic was studied to look for similar challenges and solutions. The existing literature served as input to develop new solutions tailored to conform to the concrete challenge at hand.

Based on the literature study, a solution was designed and developed. All the contributions resulting from the PhD project involve the development of tool support that either re-develops a part of the Overture tool (see chapter 2) or extends it in some way. The author of this PhD dissertation is a member of the Software Engineering Group at Aarhus University. This research group has a tradition of using formal methods and developing formal methods tools, and the research method reflects this approach. Once a solution was developed, it was applied to case studies to demonstrate that the solution contributes to addressing the challenge of interest.

Application of the proposed solutions resulted in feedback, which gave rise to additional changes and improvements to the proposed solutions. Once a solution was considered a complete piece of work, the findings and results were reported in the form of publications and submitted to either workshops, conferences or journals. The process of disseminating the findings and results helped reflect on the work, make the contributions available for others to use, and identify new areas of research.

1.5 Research Objectives

The main objective of this PhD project *is to advance the state of the art of system realisation in formal model development*. This objective is driven by the challenges inherent to realising formal specifications.

Although code generation has the potential to reduce the efforts needed to realise a formal specification, this approach also comes with certain limitations: Several target languages, implementation technologies and development environments exist, which are resource-demanding to provide code generation support for. This PhD project investigates how code generators can be designed to better meet the different needs and requirements of the steps (analysis, design, implementation, validation etc.) involved in realising a formal specification. Furthermore, the code generated version of the system cannot always be guaranteed to be a correct implementation of the formal specification. Naturally, this reduces the value of code generation as a way to realise a formal specification. To address this, this PhD project explores ways to increase the confidence in the correctness of the code generated version of the system.

The hypothesis of this PhD is that:

The automation of the steps involved in realising and validating a system based on a formal specification can be improved through use of properly designed tool support that seeks to (1) improve the integration of formal methods tools into the software development life cycle and (2) leverage the system properties described by the formal specification.

1.6 Evaluation Criteria

The criteria listed below will be used to evaluate the research contributions of this PhD project. The first three criteria, *Tool automation*, *System validation* and *Tool integration* reflect the need of the formal methods community as described in section 1.3. In particular, the *System validation* criterion is concerned with the leverage of the formal specification in the validation of the system realisation. The last criterion, *Extensibility*, is important to support developers of formal methods tools and languages in contributing new functionality.

Tool automation: The possibilities of automating the development efforts involved in realising a formal specification shall be studied, and tools must be developed to support this.

System validation: Techniques for validating the system realisation shall be investigated to help ensure that the final version of the system meets the desired system properties, described by the formal specification.

Tool integration: The possibilities of integrating different tools shall be examined. The purpose of this is to take advantage of the functionality that the different tools have to offer and support their integration into the software development life cycle.

Extensibility: The extensibility and reuse of formal methods tools shall be analysed in order to facilitate the development of new tools that support system realisation in formal model development.

1.7 Published Work

This section lists the publications produced during the PhD project. Section 1.7.1 lists the publications that form the foundation of the PhD and which are included in the dissertation. Section 1.7.2 lists the publications that have not been included in the dissertation due to space limitations.

1.7.1 Publications Selected for Inclusion

The publications listed below are available in part II.

[P18] Luís Diogo Couto, Peter Gorm Larsen, Miran Hasanagić, Georgios Kanakis, Kenneth Lausdahl and Peter W. V. Tran-Jørgensen. *Towards Enabling Overture as a Platform for Formal Notation IDEs*. 2nd Workshop on Formal-IDE (F-IDE), June, 2015.

[P22] Luís Diogo Couto, Peter W. V. Tran-Jørgensen, Joey W. Coleman and Kenneth Lausdahl. *Migrating to an Extensible Architecture for Abstract Syntax Trees*. 12th Working IEEE / IFIP Conference on Software Architecture (WICSA 2015), May, 2015.

[P24] Luís Diogo Couto, Peter W. V. Tran-Jørgensen and Kenneth Lausdahl. *Principles for Reuse in Formal Language Tools*. 31st Annual ACM Symposium on Applied Computing (SAC), April, 2016.

[P55] Peter W. V. Jørgensen, Luís Diogo Couto and Morten Larsen. *A Code Generation Platform for VDM*. 12th Overture Workshop, June, 2014.

[P21] Luís Diogo Couto and Peter W. V. Tran-Jørgensen. *Integrating Real System Components in Model-Based Development*. Draft paper planned to be submitted for publication (venue to be found).

[P20] Luís Diogo Couto and Peter W. V. Tran-Jørgensen. *Extending the Overture code generator towards Isabelle syntax*. 13th Overture Workshop, June, 2015.

[P81] Peter W. V. Tran-Jørgensen, Peter Gorm Larsen and Gary T. Leavens. *Automated translation of VDM to JML annotated Java*. Submitted to the International Journal on Software Tools for Technology Transfer (STTT), January, 2016, and invited for a second round of review.

[P80] Peter W. V. Tran-Jørgensen, Peter Gorm Larsen and Nick Battle. *Using JML-based Code Generation to Enhance Test Automation for VDM Models*. Submitted to the 21st International Symposium on Formal Methods (FM), November, 2016.

1.7.2 Other Publications Related to the PhD Project

The publications listed below are available from the respective publishers.

[57] Peter W. V. Jørgensen, Kenneth Lausdahl and Peter Gorm Larsen. *An Architectural Evolution of the Overture Tool*. 11th Overture Workshop, August, 2013.

[56] Peter W. V. Jørgensen and Peter Gorm Larsen. *Towards an Overture Code Generator*. 11th Overture Workshop, August, 2013.

[8] Nick Battle, Anne Haxthausen, Sako Hiroshi, Peter W. V. Jørgensen, Nico Plat, Shin Sahara and Marcel Verhoef. *The Overture Approach to VDM Language Evolution*. 11th Overture Workshop, August, 2013.

[47] José Antonio Esparza Isasa, Peter W. V. Jørgensen and Peter Gorm Larsen. *Hardware In the Loop for VDM-Real Time Modelling of Embedded Systems*. 2nd International Conference on Model-Driven Engineering and Software Development (Modelsward), January, 2014.

[46] José Antonio Esparza Isasa, Peter W. V. Jørgensen and Claus Ballegaard. *Modelling Energy Consumption in Embedded Systems with VDM-RT*. 4th International ABZ conference, June, 2014.

[41] Miran Hasanagić, Peter Gorm Larsen and Peter W. V. Tran-Jørgensen. *Generating Java RMI code for the distributed aspects of VDM-RT models*. 13th Overture Workshop, June, 2015.

[61] Morten Larsen, Peter W. V. Tran-Jørgensen and Peter Gorm Larsen. *Improving Time Estimates in VDM-RT Models*. 13th Overture Workshop, June, 2015.

[59] Georgios Kanakis, Peter Gorm Larsen and Peter W. V. Tran-Jørgensen. *Code Generation of VDM++ Concurrency*. 13th Overture Workshop, June, 2015.

[23] Luís Diogo Couto, Peter W. V. Tran-Jørgensen and Gareth T. C. Edwards. *Combining Harvesting Operation Optimisations using Strategy-based Simulation*. Accepted to appear at the 6th International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH), July, 2016.

[42] Miran Hasanagić, Peter W. V. Tran-Jørgensen, Kenneth Lausdahl and Peter Gorm Larsen. *Formalising and Validating the Interface Description in the FMI standard*. Submitted to the 21st International Symposium on Formal Methods (FM), November, 2016.

1.8 Outline and Reading Guide

This dissertation is divided into two parts: Part I presents the PhD project and provides a summary of the conducted research and resulting contributions. Part II contains the publications that the contributions in part I are based on.

Throughout part I the contributions are presented in a frame to stand out in the text and subsequently these contributions are referred to by their ID, e.g. [C2]. Publications that are selected for inclusion are prefixed with “P” for easy identification, e.g. [P81].

Part I is structured as follows: This introductory chapter is followed by chapter 2, which presents Overture and describes the architectural enhancements contributed to this tool. This chapter is based on [P18, P22, P24]. Chapter 3 describes several code generation features developed using the architectural enhancements made to Overture. Additionally, this chapter presents projects that use the code generation infrastructure of Overture to contribute new code generation features. This chapter is based on [P55, P21, P20]. Chapter 4 explains how the properties of a formal specification support validation of the system realisation. This chapter is based on [P81, P80]. In particular, chapter 4 presents rules and tool support that enable fully automated translation of VDM constraints to JML. This contribution is considered the most significant scientific result of this PhD. Chapter 5 describes how the tool features presented in chapter 3 and chapter 4 have been employed in externally funded projects that involve realisation of formal specifications. Finally, Chapter 6 concludes part I, evaluates the contributions and discusses future work.

Part II contains the papers selected for inclusion, which have been written by the author of this dissertation in collaboration with others. Each chapter contains a single publication, which is first presented using its bibliographic entry and subsequently the publication is presented in its submitted or published form.

2

The Overture Platform

Cost-effective use of formal methods in software development is dependent on robust tools that support all phases of the software development life cycle. The efforts needed to develop these tools can be reduced by creating an extensible platform that facilitates tool development. In this chapter the architecture of formal methods tools is studied with a particular focus on the Overture platform and the architectural changes made to it. Chapter 3 describes how these architectural changes have enabled the development of a new code generation infrastructure for Overture.

2.1 Introduction

Tool support for formal methods is often developed on top of a platform to take advantage of the functionality that the platform has to offer [P18]. Throughout this dissertation a platform is regarded as a framework that facilitates the development of Integrated Development Environments (IDEs). An IDE is therefore regarded as an instantiation of the platform that it is based on. The Overture tool is an example of an IDE that builds on top of the Overture platform. This platform has supported the contributions developed over the course of this PhD.

This chapter considers the Overture platform from an extensibility perspective and presents different contributions that support reuse in formal methods tools. These contributions cover a number of architectural changes made to the Overture platform that affect the internal representation of the VDM specification – referred to as the Abstract Syntax Tree (AST). Although the contributions presented in this chapter have been implemented in the Overture platform, the results are general enough to be applicable to other platforms or tools as well.

This section is followed by a description of the architecture of the Overture platform in section 2.2. Next, the architecture of the Overture AST is studied from an extensibility perspective in section 2.3. Finally, the experiences gained from using the Overture AST have been distilled into a set of principles for reuse in formal methods tools, which are described in section 2.4.

2.2 The Architecture of the Overture Platform

The Overture platform was originally developed to support the Overture tool, but has evolved to become a generic platform for construction of formal methods IDEs [P18]. The architecture of the Overture platform is visualised using a Unified Modeling Language (UML) package diagram in fig. 3.4. As shown in this figure, the Overture platform consists of two main parts – the *Overture language core* and the *Overture Eclipse extensions*.

The Overture tool consists of a set of plugins that are based on Eclipse [82]. These plugins analyse a VDM specification using elements of the language core and provide user interaction via the elements of the Eclipse extensions.

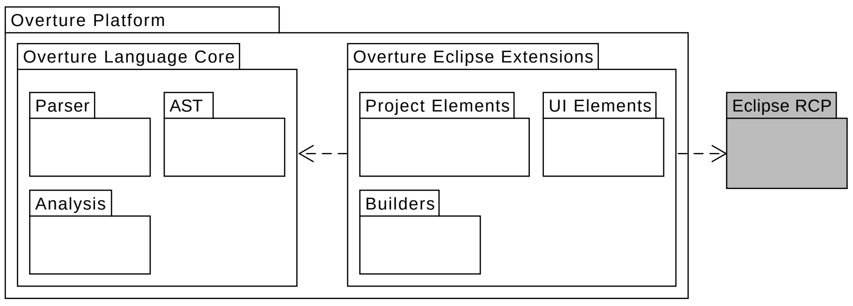


Figure 2.1: The architecture of the Overture Platform.

2.2.1 The Overture Language Core

The Overture language core encapsulates all the language related components to decouple language processing from the UI. The AST is the central component of the language core – every tool feature interacts with it in some way. The parser is responsible for instantiating the AST from model sources, containing concrete syntax. Afterwards, the AST is subjected to different analyses such as type checking and evaluation.

The AST, including tree-walkers, are generated from an AST specification, using a tool called *AstCreator*. The design of the AST produced by AstCreator, including the principles for reuse that this tool supports, are described in section 2.3.2 and section 2.4, respectively. The tree-walkers generated by AstCreator are based on the visitor pattern [39] and can be sub-classed to implement specific kinds of analyses. To name a few examples, the VDM type checker and interpreter are implemented in this way. One of the key features of the language core is that it enables language extensions via the extensible design of the AST, as described in section 2.3.

2.2.2 The Overture Eclipse Extensions

The Overture Eclipse extensions provide a set of elements for building UIs using the Eclipse Rich Client Platform (RCP) – a generic framework for building rich client applications based on a dynamic plugin model called OSGi [4]. The Eclipse extensions simplify access to the Eclipse RCP to facilitate the development of formal methods tools. The Eclipse extensions consist of UI elements such as editors and launch configurations; project elements used to manage and represent the model sources and; builders that construct and maintain the AST from the model sources.

2.2.3 Instantiating the Overture Platform

In addition to the Overture tool, other tools have been built using the Overture platform. These tools demonstrate different examples of how the extensibility features of the language core and the Eclipse extensions can be used.

The Crescendo tool [36], developed as part of the EU FP7 DESTECS project, supports collaborative modelling and simulation of Cyber-Physical Systems (CPSs). In this tool, the computer-based part of the system (the discrete-event model) is described using VDM-RT and the physical dynamics (the continuous-time model) are expressed using differential equations. The co-simulation engine coordinates the execution of the discrete-event and the continuous-time models, which are simulated using their respective simulators. The discrete-event simulator extends the VDM interpreter [66] from the language core to support (1) simulation of the VDM model until a given time-bound is reached and (2) sharing of variables between the discrete-event and continuous-time models. The Crescendo tool mostly uses ordinary Eclipse extension points (builders, debug UI etc.), but it also uses the Overture Eclipse extensions to implement e.g. editors and debugging.

The Symphony tool [16], developed as part of the EU FP7 COMPASS project, supports the COMPASS Modelling Language (CML) [84] which combines VDM with Communicating Sequential Processes (CSP) [43]. This tool makes heavy use of the extensibility features of the language core. The concrete syntax of CML differs from that of VDM due to the CSP constructs. This necessitated the development of a CML parser from scratch, which constructs an AST that is compliant with that of the Overture language core. The Symphony tool does, however, achieve a significant degree of reuse for the type checker and proof obligation generator [19]. One thing worth noting about the CML AST is that it consists of both VDM constructs (from the language core) and CML constructs, which have been added using the extensibility features described below. Together the VDM and CSP constructs form *hybrid trees*, which can be processed using *extended visitors* as described in section 2.3.

Contribution 1. An extensible platform for development of tool support for formal methods.

2.3 Migrating the AST to an Extensible Architecture

In order to use the Overture platform for language experiments the architecture of the language core had to be made more extensible. In consequence, the architecture of the AST was migrated from its original *encapsulated* architecture to an *extensible* one. The extensible architecture has supported the development of language extensions in several projects (see chapter 3) – most notably the COMPASS project where it was used to develop CML and the Symphony tool. Migrating to the extensible architecture has, however, introduced a loss in performance. For example, for the Overture interpreter the performance loss is estimated to be approximately 10%. Considering the advantages gained in terms of extensibility this trade-off is, nevertheless, considered acceptable. The complete performance analysis is available in [P22].

2.3.1 The Encapsulated Architecture

The encapsulated AST, shown in fig. 2.2, is based on that of VDMJ, which is designed with tool performance in mind. It consists of hand-written nodes which follow a cohesive Object-Oriented (OO) design, where tool features are implemented as methods inside the node classes: the `typeCheck` method performs semantic validation of a node, the `eval` method evaluates a node etc. A tool feature, such as the type checking, is performed by invoking the corresponding method on a node. As an example, fig. 2.3 shows the process of type checking an expression. This process starts by the `TypeChecker` invoking the `typeCheck` method on the expression, which responds by type checking its child node(s).

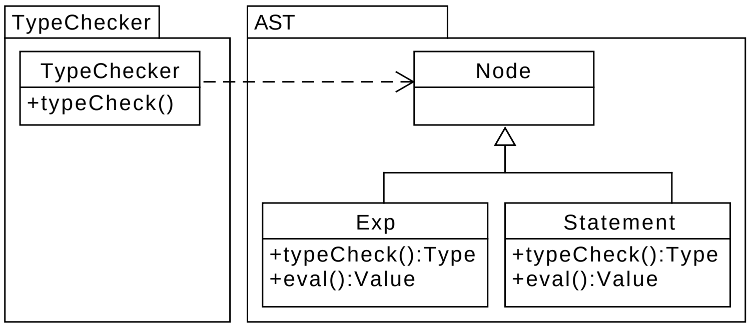


Figure 2.2: Static view of the encapsulated architecture.

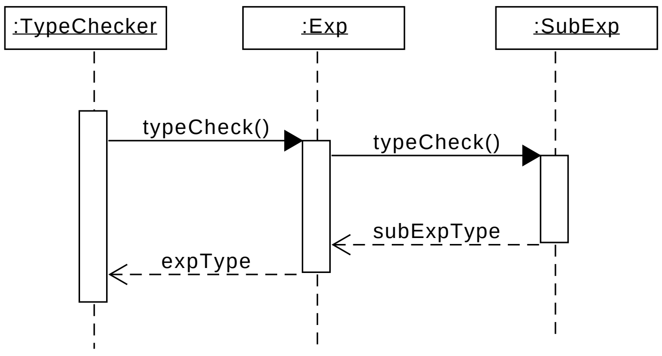


Figure 2.3: Dynamic view of the encapsulated architecture.

The main problem with this design, from an extensibility point of view, is that adding a new tool feature implies changing the AST. Modifications made to the AST to support one feature may therefore introduce problems in another feature, or conflicts in the AST implementation, due to different feature needs. Naturally, this opposes the goal of having an extensible platform. For example, adding a new code generation feature to the encapsulated AST, following the standard conventions, means implementing a `codegen` method in each node class.

2.3.2 The Extensible Architecture

To promote the extensibility of the language core the analysis functionality had to be separated from the AST [P22]. This was achieved by use of the visitor pattern. The AstCreator tool was developed to support generation of the AST, including generic base visitors, from an AST specification. To support the application of visitors to the AST, all nodes are equipped with `apply` methods. Furthermore, the generated base visitors contain a *case method* for each type of node, which is invoked upon node visitation. A tool component that needs to interact with the tree, such as the type checker, can extend one of the generated base visitors and implement the different case methods.

The structure of the extensible AST is shown using the UML package diagram in fig. 2.4, where the content of the `AST` package is generated from the AST specification using AstCreator. The process of type checking an expression under the extensible architecture is shown in fig. 2.5. Note that the application of the `TypeCheckerVisitor` to a node is followed by the invocation of a matching case method.

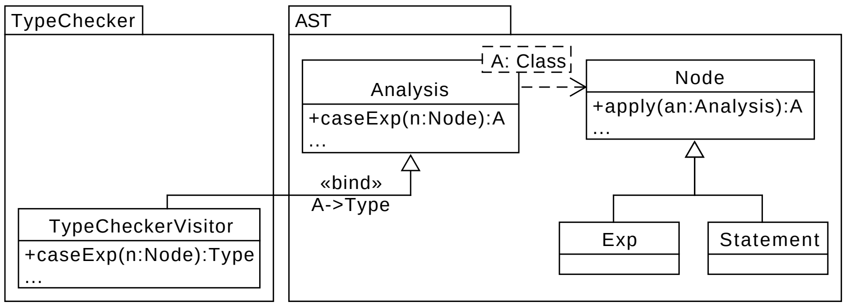


Figure 2.4: Static view of the extensible architecture.

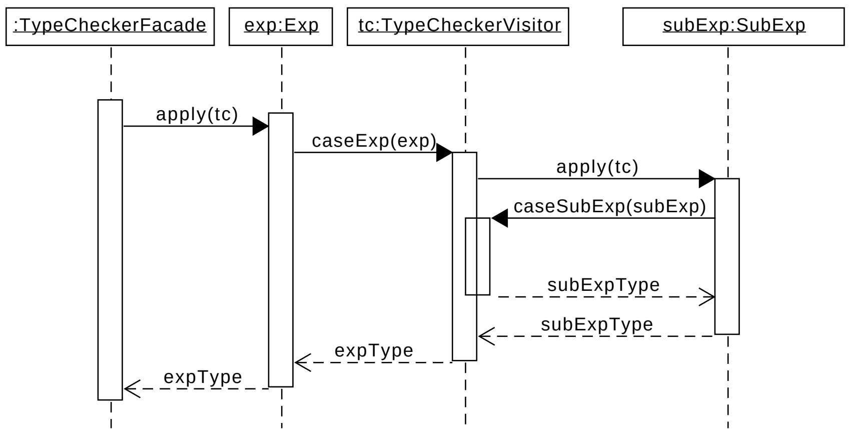


Figure 2.5: Dynamic view of the extensible architecture.

AstCreator allows existing ASTs to be extended with new nodes and fields without affecting the nodes and visitors of the base language. Based on a specification of the extension, AstCreator generates new nodes, extended nodes and extension-aware visitors. The AST extension depends on the base AST through inheritance.

To support the application of extension-aware visitors to an AST, the `apply` method of the extension nodes detects whether a visitor originates from the base or the extended AST. When an extension-aware visitor is applied to an extension node the corresponding case method is invoked in accordance with normal visitor dispatching. However, when a base visitor is applied to an extension node the closest matching default case method is invoked. Essentially, this allows a language extension to use the functionality already developed for the base language, if desired. The application of a visitor to an extension node is shown in fig. 2.6.

Contribution 2. An extensible AST architecture to facilitate the development of language extensions.

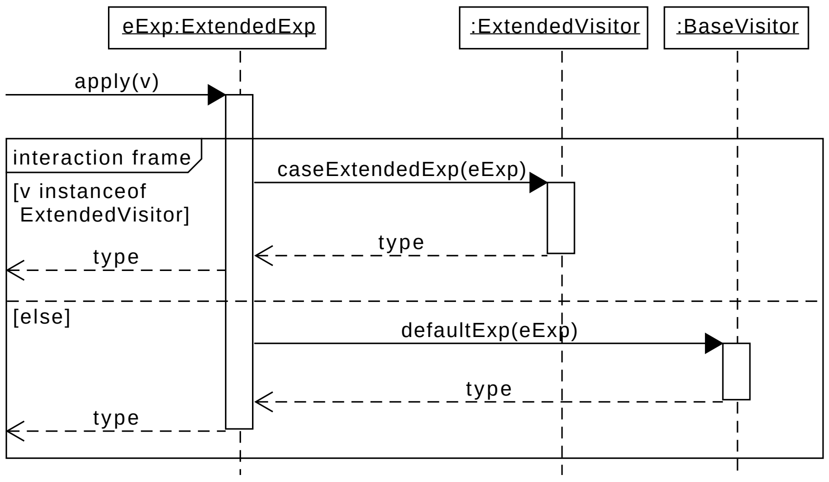


Figure 2.6: Visitor dispatching for an extended AST.

2.4 Principles for Reuse

The experiences of the COMPASS project have been distilled into a set of principles or guidelines that serve to promote reuse and extensibility for formal methods tools [P24]. These principles are supported by the AstCreator tool. The principles are described below.

Principle 1: Specification-driven ASTs. The AST is a central component that every tool feature interacts with. A language building tool should therefore provide a convenient way to maintain and extend the structure of the AST, including the different types of nodes and their individual design. A specification-driven approach focuses on the design of the AST rather than the implementation of it. This approach promotes reuse of the AST design, which in principle can be implemented in different programming languages. This principle is commonly supported by language building tools such as parser generators, which support implementation of the AST design through code generation to various programming languages.

Principle 2: Contract-based AST analysis. Any analysis that can be applied to the AST should conform to a *contract* (or interface) that is (1) parameterisable in terms of input and output and (2) leaves the implementer of the contract with the freedom to decide how the AST is traversed. A contract-based approach promotes reuse by ensuring that analyses can be used together, e.g. one analysis can extend another analysis. This principle is also key to enable support for processing of hybrid trees – ASTs that consist of nodes from both the base language and the language extension. In an OO setting, abstract classes and interfaces are usually a suitable way to define a contract. Furthermore, an OO feature such as inheritance can be advantageous in achieving the reuse and extensibility required from a contract. Similar to what is done for the AST, the contract should be derived from the AST specification.

Principle 3: Hybrid trees support. A language extension either adds new nodes, new syntactic categories or additional fields to nodes already defined by the base language. The extension should be specification-driven in accordance with the first principle. An extension consists solely of new elements that reuse the artefacts defined by the base language – without any modifications. In particular, if new fields are added to a node already defined by the language, a new node is created that extends the existing node. In an OO setting this can be achieved by making the extension node a subclass of the existing node. Together the base and the extension specifications define nodes that can be used to construct hybrid trees.

Principle 4: Base/extended analysis compatibility. In addition to the AST itself, extending a language involves extending the analyses of the base language. In accordance with the first and second principle, an extended analysis should conform to a contract derived from the base and extension specifications. An extended analysis should be able to reuse a base analysis without having to modify it. Base and extended analyses must also be *compatible* in order for them to support processing of hybrid trees. The extended contract must therefore be compatible with the contract that the base analysis conforms to. Furthermore, it should be possible for an extended analysis to redefine the behaviour of a base analysis, whenever reuse is not desired. In an OO setting this is commonly achieved using method overriding.

Contribution 3. A set of principles for reuse in formal methods tools, supported by the AstCreator tool.

3

Code Generation

The efforts needed to realise a formal specification can potentially be reduced by deriving the implementation automatically from the formal specification via code generation. This chapter presents contributions in the area of code generation that have been developed using the extensibility features of the Overture language core, introduced in chapter 2. Chapter 4 describes how these code generation contributions have been extended to support validation of the generated code against the VDM specification.

3.1 Introduction

It is beneficial to support a wide variety of implementation technologies but challenging to do so with individual code generators for each of them. One way to improve on this situation is by taking advantage of a framework-based approach in order to facilitate the construction of code generators.

This chapter presents a platform-based approach to developing code generation support for formal notations. The Code Generation Platform (CGP) uses the elements of the Overture language core to support the construction of code generators for VDM. Although the CGP is part of the Overture tool, other code generation tools may benefit from the proposed platform architecture as well. The CGP has supported the development of several code generators for VDM. This chapter studies two of them in detail, namely, Overture's Java code generator and Overture's Isabelle theory generator. Other active projects that use the CGP are mentioned in section 3.6.

The remainder of this chapter is organised as follows: The architecture of the CGP is presented in section 3.2. Next, the Java code generator, including an extension that provides code generation support for the distributed aspects of VDM-RT, are described in section 3.3. Following that, extensions for build and test automation for the Java code generator are presented in section 3.4. Next, the experiences gained from developing Overture's Isabelle theory generator are reported in section 3.5. Finally, related work on code generation for VDM is studied in section 3.6.

3.2 A Platform for Building Code Generators for VDM

The CGP provides a framework for building code generators for VDM [P55]. As shown in fig. 3.1, the CGP constructs an Intermediate Representation (IR) of the generated code from the VDM AST and passes the IR to the code generator, which translates it into *target language* code.

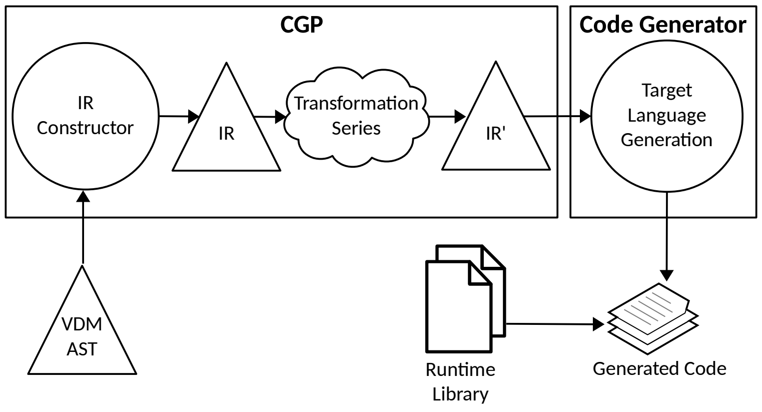


Figure 3.1: The architecture of the CGP.

In the initial version of the IR every IR node corresponds to an equivalent language construct in VDM, e.g. a statement or an expression. Subsequently, the IR is subjected to a series of behaviour-preserving transformations to replace IR nodes that are non-trivial to code generate with other IR constructs that are easier to code generate. The transformation series is configured by the code generator to conform to the peculiarities of a particular target language. When the IR has been completely transformed, i.e. it has reached a form suitable for code generation, it is passed to the code generator, which translates it into target language code.

The difficulty in implementing a code generator depends on the differences between the source and the target language. These difference are often present when the source and the target languages adhere to different language paradigms. For example, VDM contains both functional and imperative language constructs. Therefore, when VDM is translated to (say) an imperative language, there often is no single target language construct that can be used to represent a functional VDM construct. Source language constructs that are non-trivial to code generate must therefore be represented using multiple target language constructs. Similar challenges are faced when translating between other language paradigms.

The idea is for the transformations to bring the IR to a form where all the IR constructs have a mapping to a construct in the target language. Since transformations operate directly on the IR, which is independent of any target language, code generators can use and share the same transformations.

The IR is constructed with the AstCreator tool from the Overture language core (see section 2.2.1). AstCreator takes an IR specification as input and generates IR nodes and visitors. These nodes form the IR, and the visitors are used to implement transformations. One of the advantages of using AstCreator is that code generators can contribute additional IR nodes without modifying the base IR. Furthermore, since AstCreator implements the principles for reuse, described in section 2.4, the CGP supports construction and processing of hybrid trees, i.e. ASTs that consist of base and extension nodes.

The CGP provides a code emission framework, based on the Apache Velocity template engine [38], to facilitate the mapping of IR nodes into target language code. Furthermore, some code generators may use a runtime library to support the generated code. The runtime library may, for example, provide target language implementations of VDM operators or types. Although the runtime library is optional, and not a part of the CGP, it is shown in fig. 3.1 to clarify the concept of a runtime library.

Contribution 4. A Code Generation Platform for developing code generators for VDM.

3.3 Translating VDM to Java

Overture has a VDM-to-Java code generator that targets Java 7 [P55], which is an imperative OO language.[1] The Java code generator is developed using the CGP and uses several transformations to bring the IR to a form that is easier to translate into Java code.

[1] Language constructs to support the functional programming style have been added to Java 8, but the Java code generator does not rely on those.

To support the generated code, the Java code generator uses a runtime library, which provides Java implementations for some of the VDM types and operators. Sets, sequences and maps are represented using the `VDMSet`, `VDMSeq` and `VDMMap` collection classes. The `Tuple` class is used to represent and construct arbitrary tuple types and so on.

3.3.1 Use of Transformations

The Java code generator uses transformations to re-write functional constructs using imperative ones. To demonstrate this, consider the VDM function `f` in listing 3.1. This function uses a set comprehension to construct a new set from the elements of `s` for which `pred(x)` holds.

```
f : set of nat -> set of nat
f (xs) == {x | x in set xs & pred(x)};
```

Listing 3.1: Example of a VDM function that uses a set comprehension.

Java does not support set comprehensions natively. To address this, the Java code generator uses a transformation to rewrite set comprehensions to an imperative form. Hence, when the transformed version of the IR reaches the Java code generator it only contains constructs that can be translated directly to Java. The code generated version of `f` is shown in listing 3.2.

```
public static VDMSet f(final VDMSet xs) {
  VDMSet setCompResult_1 = SetUtil.set();
  VDMSet set_1 = Utils.copy(xs);
  for (Iterator iterator_1 = set_1.iterator(); iterator_1.
      hasNext();) {
    Number x = ((Number) iterator_1.next());
    if (pred(x)) {
      setCompResult_1.add(x);
    }
  }
  return Utils.copy(setCompResult_1);
}
```

Listing 3.2: The code generated version of the VDM function in listing 3.1.

Transformations are widely used by the Java code generator to support the translation from VDM to Java. Other examples of language features that are translated using transformations include pattern matching and union types, which Java does not support. Many of the transformations that initially were developed to support the Java code generator – such as the set comprehension transformation described above – have later supported the development of other code generators (see section 3.6).

3.3.2 Translation of Value Types

In Java every user-defined type is a class, hence based on reference semantics [48]. The semantics of a VDM value type can, however, be emulated in Java by representing the type as a class and then deep copying the object instances when an object is passed as an argument, appears on the right-hand-side of an assignment, or is returned as a result. To demonstrate this behaviour, consider the operation `op` in listing 3.3, which assumes the existence of a record definition of a two-dimensional vector `Vector2D`. In this listing two vectors `v1` and `v2` are created with `v2` being a copy of `v1`. Since records are value types the assignment to `v1.x` has no effect on `v2`, and therefore the operation returns 1.

```
op : () ==> nat
op () == (
 dcl v1 : Vector2D := mk_Vector2D(1,2);
 dcl v2 : Vector2D := v1; -- Copy the record value
 v1.x := 2;
 return v2.x;)
```

Listing 3.3: Example of use of value types in VDM.

Every code generated record definition is represented using a class that implements a `Record` interface, which further extends a `ValueType` interface. These interfaces are both defined in the Java code generator's runtime library. The `ValueType` interface defines the signature of a `copy` method, which takes no arguments, and returns a deep copy of the `ValueType` object. A code generated record definition implements this `copy` method and an additional `equals` method – the latter is used to compare records based on their structure. The code generated version of the VDM operation in listing 3.3 is shown in listing 3.4. In particular note that `v2` is initialised to be a copy of `v2`.

The `Utils.copy` method, used to copy `v1` in listing 3.4, checks if the argument being passed is a `ValueType` in which case it returns a copy of the argument. The copy of the argument is created by invoking the `copy` method on the `ValueType` object, i.e. `v1.copy()`. Use of the `Utils.copy`

```
public static Number op() {
  Vector2D v1 = new Vector2D(1L, 2L);
  Vector2D v2 = Utils.copy(v1);
  v1.x = 2L;
  return v2.x;
}
```

Listing 3.4: Emulation of value types in the generated code.

method to copy `ValueType` objects allows the runtime library to guard against the attempt to invoke the `copy` method on a null pointer. When **`null`** is passed to `Utils.copy`, this method simply returns **`null`**.

3.3.3 The Concurrency Extension

In [59] we extend the Java code generator to support code generation of concurrent VDM++ specifications. The concurrency extension adds additional functionality to the Java code generator's runtime library to represent and manage the concurrency constructs of VDM++ in a Java context. As an example, the `Sentinel` class is used in the generated code to control the execution of instance methods in accordance with the permission predicates, and manage history counters.

VDM++ and Java use different means to express concurrency and therefore the Java code generator uses transformations to aid the translation. As an example, the concurrency extension uses a transformation to rewrite **`mutex`** constraints using permission predicates. This transformation takes advantage of the fact that the constraint **`mutex`**$(op_1, ..., op_n)$ is semantically equivalent to having `n` permission predicates where the ith permission predicate is formulated as shown in listing 3.5.

```
per opᵢ => #active(op₁) +...+ #active(opᵢ₋₁) +
           #active(opᵢ₊₁) +...+ #active(opₙ) = 0
```

Listing 3.5: Permission predicate for op_i.

Contribution 5. A VDM-to-Java code generator supporting a large subset of VDM.

3.3.4 The VDM-RT-to-Java Code Generator Extension

In [41] we extend the VDM++-to-Java code generator to support the distributed aspects of VDM-RT (see section 1.2.2.1). In particular, this extension supports code generation of VDM-RT objects that communicate between connected CPUs.

3.3.4.1 Java RMI

The code generator uses Java Remote Method Invocation (RMI) [40] to aid the translation from VDM-RT to Java. Java RMI is a middleware technology that enables communication between objects operating under different address spaces. This technology is used to achieve transparent communication between distributed objects, which is similar to how remote communication works in VDM-RT. More specifically, the code generator is extended to produce additional boilerplate code that enables communication between distributed objects.

For a system that uses Java RMI, Java Virtual Machines (JVMs) store and obtain references for remote objects via the RMI registry. In order to pass a distributed object by reference the corresponding class must subclass Java's `UnicastRemoteObject` and implement a remote contract that specifies which methods may be invoked from a remote JVM. The implementation of a class used to create Java RMI remote objects is shown in fig. 3.2 (i.e. `UserClass`).

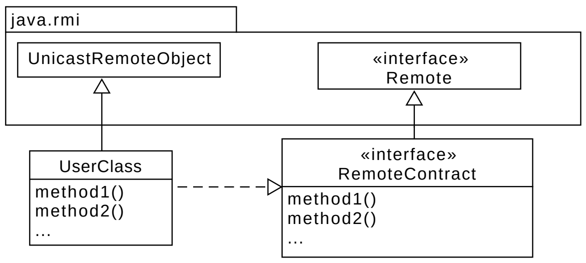


Figure 3.2: Defining remote communication using Java RMI.

3.3.4.2 Translating VDM-RT to Java

In VDM-RT an operation of an object deployed on one CPU may be invoked from a different CPU. Therefore, the generated code represents each VDM class as a Java class that subclasses `UnicastRemoteObject` and implements a remote contract. The remote contract contains the signatures of the public methods of the code generated version of the VDM class since these are the only methods that may be invoked by a remote client.

The **`system`** class is analysed statically in order to determine the architecture of the distributed system (how CPUs are connected) and the deployment of objects. The code generator determines which objects are local and remote with respect to a given CPU, since this influences the generated Java RMI code. An object that is deployed to a CPU is considered local with respect to this CPU. A remote object is an object that is deployed to a connected CPU. The code generator determines, for each VDM-RT object `o` in the **`system`** class `S`, whether `S`o` is a local or remote object with respect to a given CPU. Based on this, each CPU is represented using a JVM that, in addition to the generated user classes, has its own version of the code generated **`system`** class. Each field reference `S.o` (corresponding to `S`o`) in this class is set by the generated Java RMI code according to whether the object is a local or remote object.

Unlike VDM-RT, middleware technologies distinguish between the types used to represent local and remote objects. Therefore, a transformation replaces local class types with their equivalent remote contract type. This transformation is sound since the VDM-RT extension of the code generator ensures that every class implements a remote contract. In particular, this transformation enables remote objects to be passed as arguments and returned as results. Based on fig. 3.2, this means replacing all occurrences of type `UserClass` with type `RemoteContract`. In addition, the IR was extended with new nodes used to represent generic concepts related to distributed object communication such as remote registries and remote contracts. This extension was enabled using the extensibility features of AstCreator and the principles for reuse described in section 2.4. Templates were added for the new nodes to support the generation of Java RMI code.

3.3.4.3 Initialising and Executing the System

Before the code generated version of the system can start executing, each JVM performs the initialisation algorithm visualised in fig. 3.3. First, each JVM registers its own local objects in the RMI registry, and waits until every distributed object, specified in the **`system`** class, has been registered. Next, each JVM acquires the references for all its remote objects via the RMI registry and proceeds by submitting a special synchronisation token to the RMI registry. This token is used to indicate to the other JVMs that the submitter of the token has finished initialising. Each JVM then waits until the other JVMs have submitted their synchronisation token, and finally the system starts executing.

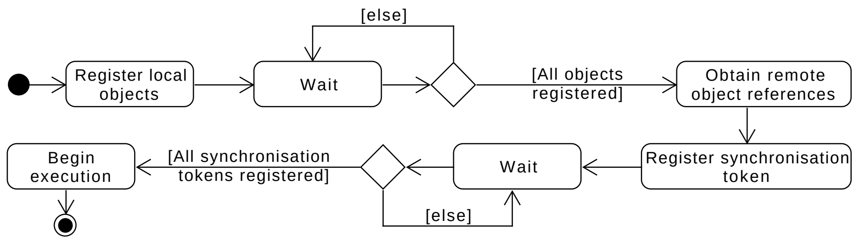

[else]
Register local objects
Wait
[All objects registered]
Obtain remote object references
Register synchronisation token
Wait
[All synchronisation tokens registered]
[else]
Begin execution

Figure 3.3: Initialisation algorithm used by every JVM.

The virtual CPU that executes the entry point of the VDM-RT specification is not represented in the generated code since it is mostly used to store objects used to represent the environment. Instead, the entry point of the system is executed on one of the JVMs used to represent a user-defined CPU. As a consequence, the VDM-RT specification cannot use objects that are deployed on the virtual CPU. This is not considered a significant limitation since these objects can be moved to a user-defined CPU in the model. To enable code generation of the VDM-RT specification, the entry point must therefore only access objects that are accessible from the CPU that executes the entry point.

Contribution 6. Extension of Overture's VDM-to-Java code generator to support the distributed aspects of VDM-RT.

3.4 Integrating Real System Components in MBD

As motivated by the case study work (see chapter 5), this section presents an extension of the Java code generator that enables automated realisation and validation of VDM specifications [P21]. This extension supports realisation of modelling setups that include real system components – a concept referred to as the *delegate* – as well as testing of the generated code using code generated VDMUnit tests (see section 1.2.2.2).

3.4.1 Build and Test Automation Technologies

Support for build and test automation has been achieved by integrating the Java code generator with Maven – a well-established build automation system for managing and building Java-based projects [70]. This extension allows the code generation features of the Java code generator to be exposed via a Maven plugin. Maven offers several features such as execution of unit tests and dependency management. Furthermore, Maven uses plugins to execute build tasks and supports development of custom plugins to intercept and control the build process.

The Maven integration also involved adding two additional transformations to the Java code generator's transformation series (see section 3.2): The first transformation enables code generation of modelling setups that integrate real system components in the simulation. The second transformation translates VDMUnit tests (see section 1.2.2.2) to JUnit4 tests, which are used to validate the system realisation. Both the delegate and the test generation features are implemented using the CGP, described in section 3.2.

Using the Maven plugin, the part of the system that is modelled in VDM can be implemented using code generation and integrated with the other system components in order to form the final version of the system. Subsequently, the code generated VDMUnit tests can be executed to validate the final version of the system.

3.4.2 The Delegate and Test Generation Features

The delegate and test generation features enable a certain style of development, which is presented below. As shown in fig. 3.4, system development is centred around the `VDM Model`, which interacts with the `External Component` via the `Bridge`. The `VDM Model` is validated using `VDM Tests`, which are written using VDMUnit. The Java code generator is invoked via the Maven plugin to produce the `Code Generated System`, which is integrated with the `External Component` using the `Delegate` mechanism. Furthermore, the `VDM Tests` are translated to JUnit4-based `System Tests` and used to validate the system realisation. The integration of the system components, as well as the test execution, is fully automated, and follows the Maven build cycle.

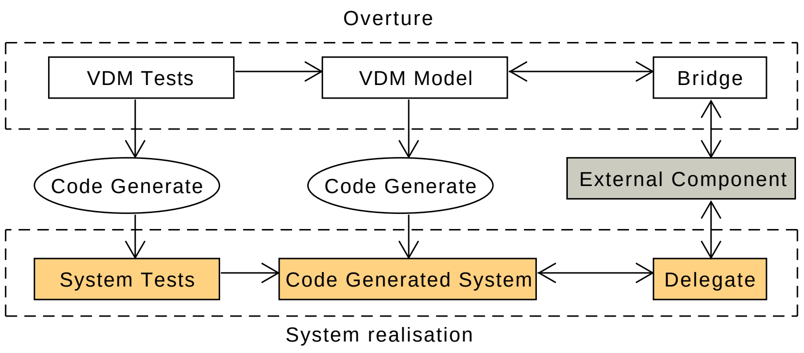


Figure 3.4: Overview of the delegate and test generation features.

3.4.3 Using an External Component in VDM

A VDM specification interacts with an external component via operations or functions that use the Java bridge (section 1.2.2.2). To demonstrate this consider the Java bridge operation in listing 3.6 which takes an argument `i` of type `I` as input, and produces a result of type `R`.

```
class Bridge
operations
op : I ==> R
op (i) == is not yet specified;
end Bridge
```

Listing 3.6: VDM operation that uses the Java bridge.

The integration of the VDM specification and the external component is enabled by Overture's `Value` runtime library (see section 1.2.2.2). Listing 3.7 shows a conceptual implementation of the Java bridge operation from listing 3.6. The implementation first converts the input paramters to a format that can be processed by the `ExternalComponent`. The converted input is then processed using the `ExternalComponent`, and the result of that is converted to a `Value` that is returned to the interpreter.

```
import org.overture.interpreter.values.Value;

class Bridge {
  public Value op(Value i) {
    I convInput = convert(i);
    R res = ExternalComponent.op(convInput);
    Value convRes = convert(res);
    return convRes;
  }
}
```

Listing 3.7: Java bridge implementation of the VDM operation in listing 3.6.

3.4.4 Using the Delegate

The `Value` runtime library enables interaction between the VDM model and the external component, but it also introduces a significant overhead due to all of its dependencies and the conversion operations that are needed (see listing 3.7). When the VDM model is realised, the dependency on the `Value` runtime library should be removed to eliminate this extra overhead. This is achieved using the delegate.

In order to use the delegate feature, each VDM class (or module) that uses the Java bridge must have a corresponding Java class that acts as a delegate. This delegate class is implemented by the user, and must include one method for each VDM operation in the VDM class that uses the Java bridge. Furthermore, each bridge-delegate pair must be configured using Maven - an example of this is shown in listing 3.8. In this listing, `Bridge` is the name of the VDM class that uses the Java bridge, and `Delegate` is the fully qualified name of the associated Java class that implements the delegate. Furthermore, to translate VDMUnit tests to JUnit4 tests, the `genJUnit4Tests` parameter must be set to `true`, as shown in listing 3.8. This listing only shows the part of the Java code generator's configuration that is directly related to the delegate and test generation features. A complete tutorial that demonstrates how these features can be used is available via [25].

During the code generation process, the delegate transformation replaces the body of each operation in the IR that uses the bridge with direct calls to the corresponding delegate method. As a result, the `Bridge` class in listing 3.6 is generated to the code shown in listing 3.9. In this listing, `I` and `R` are the

```
<configuration>
  ...
  <delegates>
    <property>
      <name>Bridge</name>
      <value>Delegate</value>
    </property>
  </delegates>
  <genJUnit4Tests>true</genJUnit4Tests>
</configuration>
```

Listing 3.8: Configuration of the Java code generator Maven plugin.

code generated versions of the input and result types of the VDM operation that they originate from.

To enable the integration of the `Bridge` and the `Delegate`, the latter must conform to the interface of the `Bridge`. Essentially this means that each method in the `Delegate` must have the same signature as the associated method in the code generated version of the `Bridge`. Therefore, knowledge of how the Java code generator represents the different VDM types in the generated code is needed. The mapping between VDM and Java types, used by the Java code generator, is described in [65].

```
class Bridge {
  ...
  public R op(I i) {
    return Delegate.op(i);
  }
}
```

Listing 3.9: Code generated version of the VDM operation in listing 3.6.

An example of a simple implementation of the `Delegate` is shown in listing 3.10. In this listing, the `Delegate` simply relays the call to the `ExternalComponent`. Although this achieves the primary purpose of the `Delegate`, i.e. to integrate the `ExternalComponent` into the generated code, the implementation of the `Delegate` may also perform other tasks such as accessing state or converting between code generated and user-defined types.

The application of the delegate transformation removes all the boiler-plate code that originally was added to enable co-execution of VDM and Java (see listing 3.7) and inserts direct invocations to the `Delegate` (see listing 3.9).

```
class Delegate {
  ...
  public R op(I i) {
    return ExternalComponent.op(i);
  }
}
```

Listing 3.10: Implementation of the `Delegate`.

This completely removes the dependency on the `Value` runtime library. The implementation of the delegate may, however, still introduce some overhead if it needs to convert between code generated types and types used by the external component.

Contribution 7. Tool support for automated realisation and validation of models that integrate real system components.

3.5 The Isabelle Theory Generator

The applications of the CGP presented so far focus on generating code towards a system realisation. This section presents a CGP-based code generator [P20] that targets an *embedding* of VDM in Isabelle – a framework for creating and working with logical formalisms [75]. In Isabelle definitions are grouped using theories. Furthermore, an embedding is a set of Isabelle theories that formalise the semantics of some language. The aim of converting a VDM specification into code of the VDM embedding is to unlock the verification features of Isabelle in a VDM/Overture setting.

The Isabelle theory generator takes as input a VDM specification, including the proof obligations generated by Overture, and generates a set of theories and proof goals, which can be analysed using Isabelle. In particular this allows Isabelle to be used to discharge proof obligations, which is something that Overture cannot currently do.

Almost every VDM construct exists in the VDM embedding. Therefore, most of the nodes in the initial version of the IR can be translated directly into Isabelle code. Furthermore, the syntax used by the VDM embedding closely resembles that of VDM with the following exceptions: All constructs in the VDM embedding are enclosed by `"` to mark them as user-defined Isabelle code. Variable names are enclosed by `^` to mark them as model variables.

Types are prefixed by `@` to mark them as model types and finally, string literals are enclosed by `''`. To exemplify this, different VDM constructs are compared to their counterpart in the Isabelle embedding in table 3.1. To translate VDM constructs that exist in the Isabelle embedding, a code mapping is specified using a template.

VDM	**Isabelle embedding**
`x`	`"^x^"`
`int`	`"@int"`
`f(1)`	`"f(1)"`
`"foo"`	`"''foo''"`
`if b then s1 else s2`	`"if ^b^ then ^s1^ else ^s2^"`

Table 3.1: VDM constructs and their counterpart in the Isabelle embedding.

There are two differences between VDM and the Isabelle embedding that are best addressed using transformations. First, Isabelle does not allow forward referencing of definitions (use of a definition before it is declared), which VDM does. To address this, a transformation is used to order definitions according to their dependency relations. The transformation processes the definitions of the IR module (the top-level node), constructs a dependency graph based on the definitions, and computes a topological sort of the graph [17].

Secondly, Isabelle requires mutually recursive functions and operations to be explicitly grouped. In the embedding this is handled by enclosing these definitions using the keywords **`begin_mutrec`** and **`end_mutrec`**. VDM, on the other hand, does not require such explicit grouping. To address this difference, a transformation computes a dependency graph for functions and operations and applies an algorithm to find strongly connected components [52]. To support this transformation and facilitate the translation of mutually recursive functions and operations, the IR was extended with a node to represent a group of mutually recursive definitions. This IR extension was made using the extensibility features of AstCreator.

Contribution 8. A code generator for translating VDM-SL specifications into theories of a VDM-SL Isabelle embedding.

In addition to the CGP-based translation, there exists an earlier version of the theory generator, which is based on a manual implementation [37] that does not use the CGP. For the rest of the section, the manual implementation of the theory generator is referred to as the *original* translation. The original translation was developed to support an Isabelle embedding of CML (see section 2.2.3), but this section only focuses on the VDM subset that it supports. To assess what was gained by using the CGP, the source code for both the original and the CGP-based translations were compared in terms of code volume, measured in Lines of Code (LoC). Although LoC does not truly reflect development efforts, it gives an idea of what can be gained by using a platform-based approach.

The measurements for the original and CGP-based translations are summarised in table 3.2. The measurements do not cover implementation components of the original translation used to translate constructs that are exclusive to CML, i.e. not included in VDM. The measurements are divided into three groups: *Data* refers to code that implements the intermediate data representation between the source and target languages. *Process* refers to code that processes or analyses the intermediate data representation and finally, *Syntax* refers to code that defines target language code.

	Manual [LoC]	CGP [LoC]	ΔLoC_{abs} [LoC]	ΔLoC_{rel}
Data	981	27	954	97.25%
Process	2427	538	1889	77.83%
Syntax	1395	86	1309	93.84%
Total	4803	651	4152	86.45%

Table 3.2: Volume comparison of the two theory generators measured in LoC.

The absolute and relative differences in LoC are computed as $\Delta LoC_{abs} = Manual - CGP$ and $\Delta LoC_{rel} = LoC_{abs}/Manual$. In total the CGP-based translation reduces the volume by 86.45% compared to the original translation. The largest reduction of 97.25% is measured for *Data*. The reason for this reduction is that the CGP-based translation reuses the IR and only needs to specify the IR extension, whereas the original translation uses a hand-written data structure. The reduction of 77.83% for *Process* is due to most of the machinery used to construct and process the IR is being handled by the CGP (applying transformations). Finally, the reduction of 93.84% for *Syntax* is because the CGP uses a template-based approach for specification of code mappings.

The take-away message is that tasks common to most code generators can be handled by the platform. As demonstrated in this section, this approach has potential to reduce development efforts when building code generators.

3.6 Related Work on Code Generation for VDM

VDMTools has supported code generation from VDM to Java and C++ since the nineties and in 1999 the Java code generator was extended to support code generation of concurrent VDM++ specifications [78]. VDMTools is a closed-source project[2] and there is no scientific literature available to document how these code generation features are designed.

In 2011 Maimaiti made the first attempt to introduce a VDM++-to-Java code generator in Overture [68]. This code generator was limited in terms of the number of supported VDM constructs and it never reached the necessary maturity to be included in a release. Maimaiti's code generator was based on the encapsulated version of the VDM AST (see section 2.3.1) which was not designed to support feature extensions such as a code generation.

In 2013, after the introduction of Overture's extensible VDM AST (see chapter 2), a new attempt was made to introduce a VDM-to-Java code generator. In 2014 the first version of the Java code generator was released with version 2.1.0 of the Overture tool. At the time of this publication, this Java code generator is under active maintenance and development. There are some significant differences between Overture's Java code generator and that of VDMTools that are worth mentioning. First, Overture's Java code generator contains support for build and test automation. As described in section 3.4, this is achieved by exposing the Java code generator as a Maven plugin. Secondly, Overture's Java code generator can be configured to translate invariants, type constraints and pre- and post conditions in VDM-SL to JML annotations that are added to the generated code (see section 4.2). Finally, Overture's Java code generator also supports code generation of traces for the VDM-SL dialect (see section 4.3).

Code generation for Overture is an active area of research. As of 2016, a VDM-RT-to-C code generator is under development as part of the INTO-CPS project [33] – a continuation of the DESTECS project (see section 2.2.3). The VDM-RT-to-C code generator was made available in version 2.3.6 releases of Overture and it is the first C code generation feature developed for VDM. Overture also has a prototype VDM++-to-C++ code generator, which has not

[2] There are, however, plans to make VDMTools open-source in 2016.

yet reached the maturity to be released with the tool [P55]. Both the VDM-to-Java, the VDM-RT-to-C and the VDM++-to-C++ code generators use many of the same transformations (without modifying them) to change the IR into a form that is easier to translate into code. It is the similarities between Java, C and C++ that enable reuse of transformations. For example, Java, C and C++ do not currently support pattern matching or collection comprehensions natively, and the code generators must therefore replace these constructs with other constructs that are easier to translate into code. This is achieved using transformations.

As of June, 2016, two Master's thesis projects about VDM code generation were completed. In [44] Holst et al. present their work on a VDM++-to-TypeScript code generator that is developed using the CGP. TypeScript is a multi-paradigm scripting language based on JavaScript that supports language features such as object-orientation, dynamic and static typing, and higher-order functions. Some of the functional constructs of VDM++ can therefore be represented directly using TypeScript code. A more detailed comparison of Overture's Java code generator and the TypeScript code generator is provided in [44, section 6.2.3]. In [28] Diswal presents his work on a CGP-based VDM-SL-to-C# code generator, which uses Microsoft Code Contracts [79, chapter 15] to represent VDM's invariants, type constraints, and pre and post conditions in the generated code. Diswal's code generator can therefore be seen as a .NET-based version of Overture's Java code generator. Microsoft Code Contracts is comparable to JML, although the semantics of the contract-based elements, as defined by these technologies, differ significantly.

In 2016, a VDM-SL-to-Smalltalk code generator was added to ViennaTalk [77]. ViennaTalk's Smalltalk code generator supports a large subset of VDM-SL, including full support for pattern matching. This code generator works by constructing a VDM AST from which it emits Smalltalk code directly. This approach is different from that of a CGP-based code generator, which generates code from an IR using a code emission framework. The design rationale of the Smalltalk code generator is to make the generated code as natural as possible. The Smalltalk code generator uses a special invariant method to implement checking of state invariants. This method is invoked on an object when the instance variables of the object change. Support for checking of type invariants and pre- and post conditions is currently limited.

4

Contract-based Validation

The contract-based elements of a formal specification describe properties desired by the final version of the system. This chapter presents contributions that extend some of the code generation features in chapter 3 to leverage these contract-based elements to validate the implementation of the formal specification. Chapter 5 describes how the contributions presented in chapters 2 through 4 have supported the case study work of this PhD.

4.1 Introduction

VDM uses pre- and post conditions to specify intended behaviour, and invariants to constrain data. These constraints, or contracts, support system analysis, and describe desired system properties. Contract-based specification supports reasoning about the system under development and validation of the system's behaviour against its contracts. For example, when the VDM specification is analysed by animation, the VDM interpreter checks that the contracts are met, and reports errors if violations occur.

This chapter describes how the constraints defined by a VDM-SL specification can be used to validate the corresponding implementation. This is achieved by translating these constraints into JML annotations that can be used to check the generated Java program for correctness (see section 1.2.3). The translation of the different VDM constructs are presented as a set of rules. These rules are implemented as an extension of the VDM-to-Java code generator [C5] to make this approach fully automated. It is further demonstrated how the generated JML annotations can be exercised exhaustively using code generated VDM-SL traces.

This chapter is structured as follows: First the rules for translating VDM constraints to JML annotations are described in section 4.2. Finally, an approach to enhancing test automation for VDM specifications using JML-based code generation is proposed in section 4.3.

4.2 Automated Translation of VDM to JML annotated Java

In [P81] we present the rules used to translate VDM constraints to JML annotations. The translation rules, including the extension of the Java code generator, form the most significant scientific contribution of this PhD. The complete definition of the translation is not included in this summary due to space limitations. The purpose of this section is instead to clarify the main ideas of the translation, and provide the information needed to understand the remainder of this summary. This section therefore only covers a subset of the translation rules, in particular those related to checking of pre conditions and some of the VDM types. Details related to checking of post conditions, state invariants, atomic execution etc. have been omitted from this section. The extension of the Java code generator, referred to as the JML translator, produces a JML annotated Java program that can be checked for correctness using JML tools.

The JML annotated Java programs produced by the JML translator can be executed using the OpenJML runtime assertion checker (see section 1.2.3). This is the only runtime assertion checker that we are aware of that currently supports both Java 7 and the JML subset produced by Overture's JML translator.

4.2.1 The ATM Model

Throughout this section the translation will be demonstrated using a small case study model of an Automated Teller Machine (ATM). Excerpts from the model are shown in listing 4.1. The ATM model uses a state component to manage information about customer `accounts`. In particular, the system stores information about debit cards that are considered valid (`validCards`), the accounts these cards are associated with, and the debit card that is currently inserted into the ATM (`currentCard`), if any. The `pinOk` flag indicates whether a valid PIN code has been entered, for the credit card currently inserted into the ATM. Finally, the model defines several operations for activities such as money withdrawal and depositing – most of which have been omitted from listing 4.1.

4.2.2 Translating VDM-SL Contracts to JML

Using the ATM model, each translation rule, included in this section, is demonstrated by example, and afterwards the rule is generalised, and presented using a grey "rule" box. The idea of the "rule" box is to emphasise and

```
module ATM
definitions
state St of
 validCards : set of Card
 currentCard : [Card]
 pinOk : bool
 accounts : map AccountId to Account
 ...
operations
...
Withdraw : AccountId * Amount ==> real
Withdraw (id, amount) == ...
end
```

Listing 4.1: Excerpts from the ATM model.

summarise a point made previously. The rules are presented as they appear in [P81] with the exact same rule number.

Pre- and post conditions are specified using **`pre`** and **`post`** clauses for the function or operation they guard. From these clauses, function definitions are derived for the pre- and post conditions. The details of these derived functions are described in the VDM-SL ISO standard [5]. The derived function definitions are not a visible part of the model, but used internally by the interpreter, to check the consistency of the model. As an example, consider the `Withdraw` operation in listing 4.2.

```
Withdraw : AccountId * Amount ==> real
Withdraw (id, amount) ==
let newBalance = accounts(id).balance - amount
in (
 accounts(id).balance := newBalance;
 return newBalance;
)
pre
currentCard in set validCards and pinOk and
currentCard in set accounts(id).cards and
id in set dom accounts
```

Listing 4.2: Operation for money withdrawal, guarded by a pre condition.

In order to withdraw money from an account, the credit card inserted into the ATM must be valid, a correct PIN code must be provided, and the associated bank account must exist. This requirement is expressed using the pre condition in listing 4.2 from which the function definition in listing 4.3 is derived. The purpose of listing 4.3 is to clarify the relationship between the **pre** clause and its derived function definition, shown in this listing. The pre_Withdraw function is an internal definition, and not a visible part of the model. This function receives all the input parameters of the Withdraw operation, including a copy of the state component, and performs the pre condition check.

```
pre_Withdraw: AccountId*Amount*St +> bool
pre_Withdraw (id, amount, St) ==
 St.currentCard in set St.validCards and St.pinOk and
 St.currentCard in set St.accounts(id).cards and
 id in set dom St.accounts
```

Listing 4.3: Derived pre condition function for the Withdraw operation.

The Withdraw operation and the pre_Withdraw function are represented using methods with the same name in the generated code. Since the pre_Withdraw method originates from a function, which cannot access state directly, this method is declared **static** and marked using JML's **pure** modifier. In general this applies to all function definitions whether they are explicitly defined, or derived from pre- or post conditions or invariants.

2. Translation of functions
Any function – whether it is defined by the user or derived, e.g. from a **pre** or **post** condition clause – code generates to a **static** Java method that gets annotated with the **pure** modifier.

In order to ensure that the pre condition of the Withdraw operation is met, the pre_Withdraw method is invoked from the **requires** clause of the Withdraw method as shown in listing 4.4. The pre_Withdraw method receives all the input parameters of the Withdraw method, including a copy of the state component, since the pre condition method is allowed to reason about module state.

```
//@ requires pre_Withdraw(id,amount,St);
public static Number Withdraw(final Number id, final Number
     amount) {...}
```

Listing 4.4: Code generated version of the `Withdraw` operation.

3. Translating the pre condition of an operation

Let `op` be a method code generated from a VDM-SL user-defined operation and let the signature of `op` be:
`static` `R op(`I_1 i_1`,...,`I_n i_n`)`
Then `op` has a code generated pre condition method `pre_op` that is **`pure`** and which in addition to the parameters of `op` also takes the state component `s` as an argument, i.e.
`/*@` **`pure`** `@*/` **`static boolean`**
`pre_op(`I_1 i_1`,...,`I_n i_n`,S s)`
To ensure that the pre condition check gets performed, we annotate `op` with the following **`requires`** annotation:
`//@` **`requires`** `pre_op(`i_1`,...,`i_n`,s);`

Rule 3 assumes the existence of the state component of the module enclosing the pre condition function. For situations where the state component is not defined, this rule is changed such that the pre condition method does not include the state parameter `s`.

4.2.3 Checking VDM-SL Types using JML

The JML translator uses JML annotations to check the constraints imposed by VDM types in the generated code. This is achieved using a function `Is(v,T)` that takes as input a Java value `v` and a VDM type `T` and produces a predicate that checks whether `v` represents a value of type `T`. The purpose of this function is to produce a check that when added to the generated code checks that a Java value or object reference remains consistent with the VDM type that it originates from.

`Is(v,T)` is defined as a recursive function over the different classes of VDM types. This function defines 19 cases, and hence introduces the same number of rules. The complete definition of `Is(v,T)` is given in [P81]. Another aspect of checking VDM type constraints across the translation is *where* in the generated code the type checks must be added. A detailed description of how the JML translator handles this is provided in [P81, section 6.1].

In the most simple case `Is(v,T)` is used to check constraints imposed by VDM's basic types. This is necessary, since VDM has a more fine-grained representation of numbers using rational and natural numbers, reals and integers. Java, on other hand, only uses a single data type to represent integers. Therefore, whenever the ATM model uses a positive integer to represent an amount of money, using VDM's **`nat1`** type, the generated code must ensure that the corresponding Java value meets this type constraint. As an example, consider the VDM fragment in listing 4.5, which inserts an amount of money, required to be a positive integer, into some account. If this constraint is violated Overture will report a runtime error.

```
let amount : nat1 = expense - profit in
  Withdraw(accId, amount);
```

Listing 4.5: Explicit type annotation used to ensure that a valid amount is being withdrawn.

The generated code, shown in listing 4.6, checks this constraint using the `Utils.is_nat1` method available via the Java code generator's runtime library. This method is invoked from a JML assertion to ensure that `amount` is a positive integer.

```
Number amount = expense.longValue() - profit.longValue();
//@ assert Utils.is_nat1(amount);
return Withdraw(accId, amount);
```

Listing 4.6: JML used to check that a valid amount is being withdrawn.

11. Checking of the `nat1` type
Let `v` be a value or object reference in the generated code that originates from a variable or pattern of type **`nat1`** and further define `Is(v,`**`nat1`**`) = Utils.is_nat1(v).` To ensure that `v` represents a value of type **`nat1`**, generate a JML check to ensure that `Is(v,` **`nat1`**`)` holds.

Checking of basic types is similar in all cases, and these types constitute the base cases of `Is(v,T)`. The JML annotations used to check collection-based, user-defined and union types, for example, are more complicated. To demonstrate this, consider the *named invariant type* `Amount` in listing 4.7. For this particular example, it is said that `Amount` is based on the *domain type* **`nat1`**. In addition, since the ATM cannot dispense amounts larger than 2000, values of this type are constrained accordingly using an invariant.

```
types
Amount = nat1
inv a == a < 2000;

operations
Withdraw : AccountId * Amount ==> real
Withdraw (id, amount) == ...
```

Listing 4.7: Money to be withdrawn modelled as a named invariant type.

From the named invariant type, a function is derived, which is used to check the consistency of values of type `Amount`. This is similar to how functions are derived from **`pre`** and **`post`** clauses, except that the invariant function only takes a single argument – the value that is validated against the invariant predicate. The code generated version of the invariant function is shown in listing 4.8.

```
...
/*@ pure @*/
public static Boolean inv_ATM_Amount(final Object check_a){
 Number a = ((Number) check_a);
 return a.longValue() < 2000L;
}
```

Listing 4.8: The invariant method for `Amount`.

As shown in listing 4.9, on entering the `Withdraw` method, a JML assertion is used to check that the amount, passed as input to this method, is consistent with the VDM type `Amount`. The JML assertion first checks that `amount` is a valid domain type value, and secondly it checks that the invariant predicate is met. Essentially, this corresponds to checking that `amount` is a positive integer (left part of the conjunction) that is less than 2000 (right part of the conjunction).

It is important to note that meeting the invariant predicate does not imply being a valid domain type value, and vice versa. For example, -1 meets the invariant predicate since it is smaller than 2000, but it is not of type **`nat1`**, i.e. a legal domain value. Conversely, 2001 is of type **`nat1`** but it does not meet

```
public static Number Withdraw(final Number id, final Number
     amount){
...
//@ assert Utils.is_nat1(amount) && inv_ATM_Amount(amount);
...
}
```

Listing 4.9: JML used to check that `amount` is consistent with the VDM type `Amount`.

the invariant predicate. Since the invariant function and method assume that the input is a legal domain type value, the generated JML assertion must first and foremost ensure that `amount` is a positive integer. If that condition is met, it is checked that `amount` meets the invariant predicate. Since the JML assertion in listing 4.9 is evaluated using short-circuit or McCarthy semantics [71], the invariant method is only invoked if `amount` represents a legal domain value. Therefore, it is safe to cast the input argument of the invariant method, before performing the invariant check, as shown in listing 4.8.

16. Checking of named invariant types

Let `v` be a value or object reference in the generated code that originates from a variable or pattern of the VDM named invariant type `T` based on the domain type `D` and constrained by invariant predicate `e(p)`, i.e. `T` is defined as

```
types
T = D
inv p == e(p)
```

Then `T` has an invariant method, responsible for running the code generated version of the `e(p)` check, with a signature defined as:

```
public static boolean inv_T(Object o)
```

Further define `Is(v,T) = Is(v,D) && inv_T(v).`

To ensure that `v` represents a value of type `T`, generate a JML check to ensure that `Is(v,T)` holds.

The invariant method `inv_T` in rule 16 uses Java's `Object` class to represent the type of its input parameter. This allows the invariant method to take any Java type as input. The type of the parameter could in principle be represented using a smaller type, for example using the code generated version of the domain type of `T`. However, this would in some situations lead to type casting in the generated JML annotations that invoke the invariant method. In particular for situations where the value passed to the invariant method is masked as a union type. Instead, the task of narrowing the type of the input parameter is handled by the invariant method.

Contribution 9. Automated translation of VDM-SL to JML annotated Java.

4.3 Enhancing Test Automation for VDM Models

This section demonstrates how a code generated version of a VDM-SL model, produced using the JML translator, can be tested using combinatorial testing. This is achieved by code generating a trace and executing it against the JML annotated Java program, generated from the VDM-SL model, as described in [P80]. The trace must be executed using a JML tool in order to enable checking of the generated JML constraints.

Later in section 5.4 it is demonstrated how code generated traces have been used to analyse the properties of an algorithm used to obfuscate IDs of retailers. This section focuses on describing how the trace is code generated, i.e. how the different trace operators are represented in the generated code, and how the code generated version of the trace can be executed. The aim of using code generated traces is twofold: First, it allows one to obtain a higher degree of confidence in the correctness of the generated code. Secondly, it has the potential to allow a larger number of tests to be executed faster since the tests are executed as compiled code rather than using a VDM interpreter. The latter is expected to be particularly beneficial for traces that expand to large test sets. Such test sets may be impossible or intractable to execute using an interpreter due to a large memory consumption or execution time.

Internally, Overture uses an AST to represent a trace. This AST consists of nodes that constitute the different trace constructs such as **`let`** bindings, and calls to functions and operations. This trace, or AST, describes a pattern that Overture *expands* into a collection of tests that are *executed* against the VDM specification. The idea of this work is to take advantage of code generation and use compiled code to perform the expansion and execution.

Similar to Overture, the JML translator represents the trace as an AST that is built using nodes from the Java code generator's runtime library. Most of these nodes represent the different trace operators (see section 1.2.2.2). In VDM the `|` operator and **`let be st`** binding are both represented using the `Alternative` trace node. The `||` operator is represented using the `Concurrent` trace node. The repetition operator is represented using the `Repeat` trace node. The `Sequence` trace node expands to the sequencing of all the tests of its child nodes. The `Statement` trace node expands to a single test, i.e. the invocation of the `Call` statement that it is associated with. The `Call` statement represents the invocation of some function or operation. Objects of this class constitute the leaves of the trace AST.

When a trace is code generated the JML translator produces code that when executed constructs the trace AST, expands it, and executes the generated tests. To demonstrate the complete process of expanding and executing a code generated trace consider the trace in listing 4.10. The non-deterministic choice between `op1(x)` and `op2(x)` expands to two tests, i.e. `op1(x); op2(x)` and `op2(x); op1(x)`. Repeating `op3(x)` up to two times further produces two tests, namely, `op3(x)` and `op3(x); op3(x)`. Finally, the trace produces tests for each binding of `x` leading to a total of eight tests as shown in listing 4.11.

```
let x in set {1,2} in (
  ||(op1(x),op2(x)) | op3(x){1,2}
)
```

Listing 4.10: Example of a trace.

```
x = 1; op1(x); op2(x);    /* Test 1 */
x = 1; op2(x); op1(x);    /* Test 2 */
x = 1; op3(x);            /* Test 3 */
x = 1; op3(x); op3(x);    /* Test 4 */
x = 2; op1(x); op2(x);    /* Test 5 */
x = 2; op2(x); op1(x);    /* Test 6 */
x = 2; op3(x);            /* Test 7 */
x = 2; op3(x); op3(x);    /* Test 8 */
```

Listing 4.11: The tests generated from the trace in listing 4.10.

When a trace AST has been constructed it forms a tree of objects from which the tests will be derived. The runtime representation of the trace in listing 4.10 is shown using the UML object diagram in fig. 4.1. The execution of the trace is handled entirely by the runtime library, as visualised in fig. 4.2. Initially the `execTests` method is invoked to expand and execute the trace. This method is passed four arguments, i.e. `ast`, `module`, `store` and `testAcc`. The first argument, `ast`, is a representation of the trace AST such as that shown in fig. 4.1. `module` is the code generated version of the module enclosing the trace. `store` is used to manage and reset the system's state between each test run in order to allow tests to be executed independently of one another. `testAcc` is used to record or accumulate test results. For each test that is executed the `testAcc` is notified using the `registerTest` method. This call has been omitted from fig. 4.2 for simplicity. Test accumulators are implemented using the strategy pattern [39], which allows test results to be accumulated in different ways. For example, one strategy may write the results to a file, while another strategy may print the results directly to a console.

Before the tests are executed the module class, enclosing the code generated trace, is registered in the `store`, as shown in fig. 4.2. If more modules exist, these are also registered in the `store` as they might have their state changed during test execution. After the store has been configured, the tests are obtained from the trace AST, using the `getTests` method. This method returns a `TestSequence` that contains the trace tests. Afterwards, the tests are executed, one by one, and the system's state is reset between each test run.

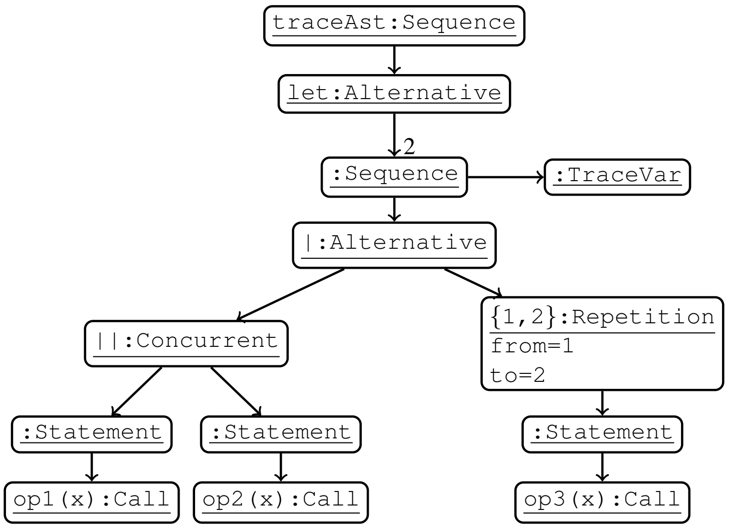


Figure 4.1: A code generated trace shown using a UML object diagram.

The leaves of the trace AST – the `Call` statement objects – are all implemented and instantiated using anonymous classes based on the abstract `Call` statement class. These anonymous classes are implemented by the code generator according to the particular VDM function or operation call that the `Call` statement represents. The interface of the `Call` statement node, shown in fig. 4.3, defines three methods: The `isTypeCorrect` method determines whether the arguments passed to the `Call` statement meet the type constraints of the formal parameters of the VDM function or operation that they originate from. The `isTypeCorrect` method returns **`true`** by default, which corresponds to the situation where it can be guaranteed using static analysis that the `Call` statement is type correct for all tests. For such situations the implementation of the `isTypeCorrect` method can be omitted by the code generator. If the `Call` statement is not considered type correct, i.e. the `isTypeCorrect` method returns **`false`**, then the current test is registered as `INCONCLUSIVE`. The implementation of the `Call`

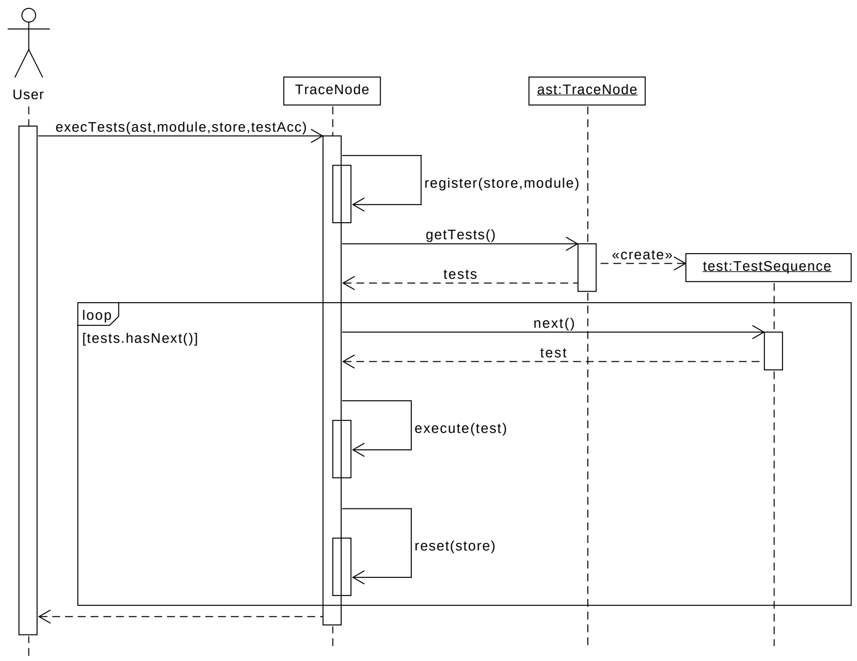


Figure 4.2: Execution of a code generated trace.

statement for the `op3(x)` call from listing 4.10 is shown in listing 4.12. For this example, it is assumed that `op3` defines a single input parameter that is a natural number (of type **`nat`**). As shown in listing 4.12, this argument is accessed from a scope surrounding the `isTypeCorrect` method, and validated using a JML assertion. Note that the `isTypeCorrect` method assumes that the JML annotations are compiled using Java assertions, which is possible using OpenJML. Therefore, if the compiled code is executed using the OpenJML runtime assertion checker with Java assertions enabled, a JML violation will produce an `AssertionError`.

In general a `Call` statement is type correct if all the input arguments $\texttt{a}_1, \ldots, \texttt{a}_n$ represent values that meet the constraints imposed by the types of the formal parameters $\texttt{T}_1, \ldots, \texttt{T}_n$ of the VDM function or operation that the `Call` statement represents. That is, for a `Call` statement to be type correct $\texttt{Is(a}_1\texttt{,T}_1\texttt{)} \ \&\& \ \ldots \ \&\& \ \texttt{Is(a}_n\texttt{,T}_n\texttt{)}$ must hold. This condition is checked using JML as shown in listing 4.12.

If a `Call` statement is considered type correct the runtime library proceeds by checking that the pre condition is met. This check is performed using the `meetsPreCond` method. This method returns **`true`** by default, which corresponds to the situation where the pre condition is either omitted or can be guaranteed to be true for all tests. Listing 4.12 shows how the `meetsPreCond` method is implemented for `op3`. If the pre condition is met, the runtime library proceeds by executing the `Call` statement, using the `execute` method, and returning the result to the runtime library. The `execute` method is responsible for invoking the code generated version of the function or operation that the `Call` statement is associated with. However, if the pre condition is not met the test is considered `INCONCLUSIVE`. The implementation of the `execute` method for `op3` is shown in listing 4.12.

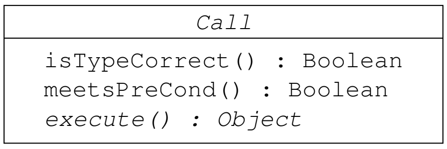


Figure 4.3: The interface of the `Call` statement.

Contribution 10. Enhanced test automation for VDM models.

```
Call callStm_3 = new Call() {
   public Boolean isTypeCorrect() {
     try {
       //@ assert Utils.is_nat(x);
     } catch (AssertionError e) {
       return false;
     }
     return true;
   }
   public Boolean meetsPreCond() {
     return pre_op3(x);
   }
   public Object execute() {
     return op3(x);
   }
   public String toString() {
     return "op3(" + Utils.toString(x) + ")";
   }
 };
```

Listing 4.12: Implementation of the `Call` statement for `op3`.

5

Case Studies

Tool support for formal methods must be properly integrated into the software development life cycle to support cost-effective development of high-quality software. This chapter describes how the contributions presented in the previous chapters have supported three case studies. Chapter 6 evaluates the contributions produced during the PhD, concludes the dissertation, and presents future work.

5.1 Introduction

The contributions presented in the previous chapters have been used in three case studies to develop systems of a mission-critical nature. The feedback received from the case study work has helped improve the existing contributions and introduce new ones. As an example, the build and test automation features [C7] originate from the unforeseen needs encountered during the case study work.

The first case study, presented in this chapter, concerns the development of Functional Mock-up Interface (FMI) [26] support for Overture. Essentially, this feature enables co-execution of VDM and other formalisms that can be exported in Functional Mock-up Units (FMUs). The second case study uses the delegate and test generation features [C7] to develop a system that supports farmers in optimising harvest operations using simulation-based predictions. The last case study uses the JML generator [C9] and code generated traces [C10] to validate the properties of an algorithm used to obfuscate Financial Accounting District (FAD) codes, which are six digit numbers used to identify branches of a retailer. In particular, this case study investigates the performance that is gained by using code generated traces.

This chapter is organised as follows: The experiences gained from developing FMI support for Overture are reported in section 5.2. Next, the development of the harvest planning system using the delegate and test generation features is described in section 5.3. Finally, the performance of code generated traces is analysed using the FAD code case study in section 5.4.

5.2 The FMU Orchestration Engine Rule Checker

In the INTO-CPS project (see section 3.6) a number of simulators, supporting different formalisms, are being extended with functionality to export models in FMUs. FMUs implement the interface requirements described in the FMI standard, which enable sub-systems, modelled using different formalisms, to communicate in a co-simulation. For example, a CPS may be analysed using several models that use heterogeneous formalisms to model different parts of the system. If these constituent models are exported in FMUs, via different tools, then the FMUs can in principle interact in a co-simulation. It is the responsibility of the *orchestration engine* to coordinate the execution of the individual FMUs and handle the communication between them.

5.2.1 The Rule Checker Model

The case study described in this section [42] uses MBD to develop a component of a co-simulation orchestration engine called the *rule checker*. This component is used to perform a static validation of the information that will be exchanged between FMUs. FMUs communicate by means of *scalar variables*, which are characterised by their causality (e.g. input or output), variability (e.g. discrete or continuous), type and initial value. As an example, a FMU may use a scalar variable to represent a sensor reading that serves as input to other FMUs.

Initially it was attempted to implement the rule checker directly in Java. However, the interface requirements, described in the FMI standard, are based on tables and natural language descriptions that gave rise to discussions about the scalar variable semantics. This led us to believe that the development of the rule checker was well-suited to be addressed using formal analysis. The purpose of the analysis was to improve our understanding of the FMI standard, as well as to identify the parts of the standard that are relevant to the rule checker. This case study therefore demonstrates traditional usage of formal analysis to address the ambiguity of an informal specification, which potentially would lead to problems in the implementation. The rule checker is specified using VDM-SL, which is well-suited for analysing the interface constraints by animation, and implemented using the Java code generator [C5].

The rule checker model is used to determine whether a scalar variable is valid, and reports a meaningful error message if it is not. The scalar variables are validated once by the orchestration engine before the co-simulation is started. If there are multiple problems with a scalar variable, the rule checker will report the problem that is considered most severe. For example, the rule checker will first and foremost require that a scalar variable of continuous variability is of type real. In the VDM model, the scalar variable is represented using the record type, `SV`, shown in listing 5.1.

```
SV ::
  causality   : [Causality]
  variability : [Variability]
  initial     : [Initial]
  type        : Type;
```

Listing 5.1: A scalar variable represented using a record type.

The rule check is performed for each FMU that participates in the co-simulation. As a first step, the rule checker provides default values for the scalar variable properties that have been legally left undefined. Afterwards, the rule checker validates the scalar variables against the FMI standard using a function called `Validate`. If a scalar variable is valid, this function returns **`true`**. Otherwise, the function returns **`false`** and produces an error message that describes why the scalar variable is invalid.

5.2.2 Realising the Rule Checker

When a formal specification, such as the rule checker model, is implemented using code generation, the generated code will typically constitute one of several components that altogether form the system realisation. The generated code will therefore need to interact with other system components. In this case study, the `RuleChecker` is, for example, a sub-system of the `OrchestrationEngine` as shown in fig. 5.1.

From a general perspective, the generated code may need to be integrated with other system components such as user interfaces, hardware, third-party libraries or other dependencies. Seamless realisation of a formal specification therefore requires tools that support the build environments that integrate all the system components. The experiences gained from developing the rule checker therefore inspired us to extend the Java code generator with build

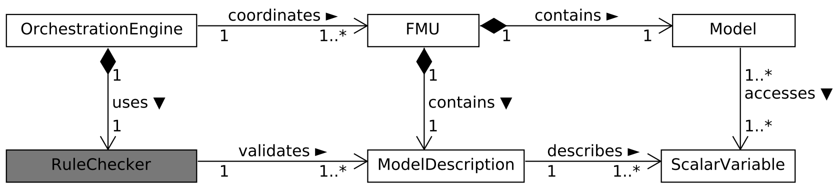


Figure 5.1: Overview of the FMI case study.

automation as described in section 3.4. In this way Maven can be used to integrate all the system components when the system is realised.

The code generated version of the rule checker is validated with manually implemented tests. These tests help ensure that the rule checker responds as expected and reports appropriate error messages for invalid scalar variables. It is worth noting that the formalisation of the static semantics of scalar variables helped us identify an inconsistency in the FMUs generated by the commercial modelling tool, Dymola 2015 [30]. The problem was encountered when we tried to execute a co-simulation where one of the connected FMUs was exported using Dymola 2015. More specifically, our orchestration engine rule checker reported an error for one of the scalar variables, defined in the model description of this FMU. The scalar variable, causing the error, was defined as an output variable (the causality) of fixed variability. However, according to the FMI standard, version 2.0, this is not a legal combination of variability/causality.

Contribution 11. A Functional Mock-up Unit orchestration engine rule checker specified in VDM-SL and translated to Java using the VDM-to-Java code generator.

5.3 The Harvest Planning System

The delegate and test generation features have been developed to support an externally funded project on harvest optimisation in the area of precision agriculture [23, P21]. One of the goals of this project is to develop a harvest planning system that helps farmers optimise harvest operations using simulation-based predictions.

The part of the system that is used to simulate the harvest operation is partly modelled in VDM and realised using Overture's Java code generator. The code generated version of the harvest simulator is one of several system components that form an *offline planning tool*, which enables farmers to make predictions about harvest operations. The project also aims to develop tools that address other aspects of harvest optimisation. However, the remainder of this dissertation only concerns the development of the offline planning tool, especially the MBD aspects.

5.3.1 The Harvest Simulator Model

A domain model [31] that describes the relationships between the domain elements of the harvest planning system is shown in fig. 5.2. For a given field, and a selection of resources (harvesters, grain carts and a storage facility), the systems simulates the harvest according to the chosen optimisation algorithms [11, 51, 74]. The field is divided into work rows and headland segments, where the latter surround the field and allow the vehicles to manoeuvre during the harvest. Based on the user's configuration, the harvest simulator calculates instructions that describe the actions taken by the vehicles during the harvest simulation. These instructions constitute a plan for how the harvest may be performed in a realistic setting, and – when executed – the instructions provide an estimate of the time the harvest will take.

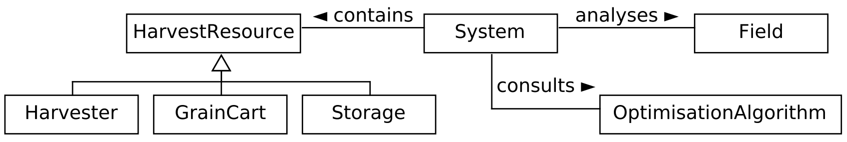


Figure 5.2: Harvest planning system domain model.

The plan produced by the harvest simulator may be inspected using a visualiser as illustrated in fig. 5.3. For the scenario visualised in this figure, the system has a harvester, a grain cart, and a storage facility available. Furthermore, for this scenario, the system is configured to perform unloading and harvesting concurrently – an optimisation algorithm referred to as "on-the-go" unloading. Due to this choice of optimisation algorithm, the harvester and grain cart are driving side-by-side in adjacent rows, while the unload is being performed. The grain cart thus becomes responsible for transferring the yield to the storage facility, shown at the left of fig. 5.3. Alternatively, unloading can, for example, be performed using the “static unloading” optimisation algorithm. When this optimisation algorithm is used the harvester and grain cart perform unloading at a designated unload point – without moving.

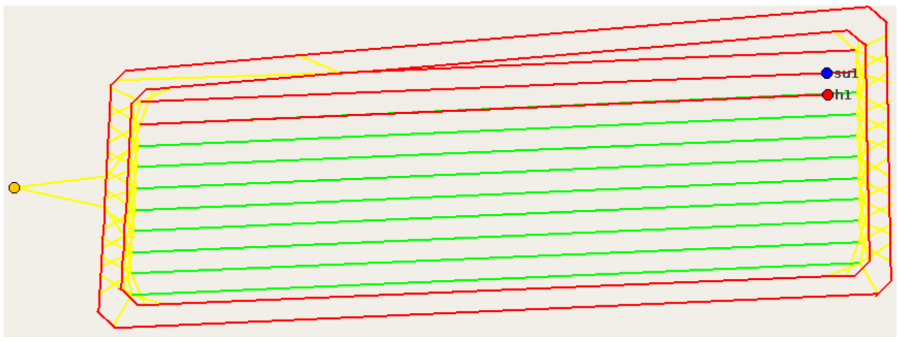


Figure 5.3: Visualisation of a harvest plan.

5.3.2 Employing the Delegate

The model used to analyse the harvest operation exists in two versions: The first version [23] is specified solely using VDM, while the new version [P21] uses the Java bridge to analyse the field representation. The Java bridge was introduced to address some of the performance issues with the first version of the model. In that version of the model the data structures and algorithms used to analyse the field are based on non-standard designs that do not scale well in terms of field size. This makes it either impractical or impossible to use the first version of the model to analyse realistic-sized fields: In [P21] we report that for a relatively large field, consisting of 16 work rows and 2 headlands, the old versions of the model and the system realisation did not complete the simulation within eight hours. For comparison, the new version of the system, performs the simulation in approximately 183.8 seconds ($\approx$ 3 minutes) using the model and in approximately 1.0 seconds using the system realisation. A more detailed analysis and discussion of the system’s performance is provided in [P21].

The new version of the system uses a Java-based graph representation of the field. In that way the field can be analysed using standard algorithms, implemented using well-established libraries. More specifically, the new version of the model uses the Java bridge and the JGraphT library [52] to handle the field representation. As shown in fig. 5.4, the graph representation of the field therefore constitutes the *external component* of the harvest planning system. The JGraphT library provides implementations of some of the common algorithms used to analyse the field. As an example, to find the shortest path between two points in the field, the new version of the model uses Dijkstra's algorithm [27]. The shortest path algorithm can, for example, be used to compute the route a grain cart must take in order to service a harvester that needs to unload at a designated unload point.

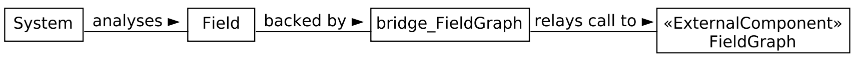


Figure 5.4: Integration of the external component of the harvest planning system. `System`, `Field` and `bridge_FieldGraph` are defined using VDM, whereas `FieldGraph` is implemented in Java.

Every operation in the `bridge_FieldGraph` that uses the Java bridge is defined using the **`is not yet specified`** statement to indicate that the operation is executed in an external environment. As an example, the `shortestPath` operation, shown in listing 5.2, takes as input the IDs of two points or vertices in the field, and returns the shortest path between them. In the model, the path is represented as a sequence of pairs, where each pair consists of an edge ID and the direction (`<Standard>` or `<Reverse>`) in which the edge must be traversed. The introduction of the `Field` class, in-between the `System` and the `bridge_FieldGraph`, allows additional processing of the data returned by the `bridge_FieldGraph` to be performed, if needed.

In the new version the model the shortest path algorithm is implemented in the `FieldGraph`, i.e. the external component. Whenever the `System` needs to determine the shortest path between two vertices, it invokes the `shortestPath` operation on the `Field`. The `Field` then relays the invocation forward to the `bridge_FieldGraph`, which uses the `FieldGraph` to compute the shortest path. This scenario is visualised in fig. 5.5.

The model is primarily tested using the visualiser and the VDMUnit framework (see section 1.2.2.2). The visualiser has been used to identify potential problems with the optimisation algorithms, while they were being developed. VDMUnit has been used to test small parts of the system and to write integration tests that perform complete simulations, and make assertions

```
class Field
operations
public shortestPath : int * int ==> seq of (int * Global`
    Direction)
shortestPath (pFromV,pToV) ==
  bridge_FieldGraph`shortestPath(pFromV, pToV);
...
end Field

class bridge_FieldGraph
operations
public static shortestPath : int * int ==> seq of (int *
    Global`Direction)
shortestPath (pFromV,pToV) ==
  is not yet specified
...
end bridge_FieldGraph
```

Listing 5.2: Example of a field operation that uses the Java bridge.

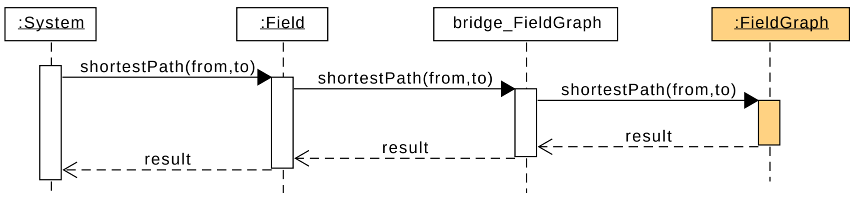


Figure 5.5: The model computes the shortest path using a Java-based external component (highlighted in the figure).

about the simulation outcome. These tests enable regression testing of the system as new functionality is being developed.

5.3.3 Realising the Harvest Planning System

When the system is realised, the Java code generator is invoked via its Maven plugin to translate the VDM model and the VDMUnit tests into code and JUnit4 tests. Next, the generated code is validated using the JUnit4 tests to obtain increased confidence in the correctness of the system realisation. If the system passes the JUnit4 tests, Maven continues by integrating the code generated version of model with the remaining system components in order to construct the final version of the offline planning system. The build process is fully automated and handled entirely by Maven.

Figure 5.6 shows how the code generated version of the VDM model – the `HarvestSimulator` highlighted in the figure – is integrated with the remaining system components. Essentially, the offline planning tool adds a UI on top of the `HarvestSimulator` that allows the user to configure the harvest operation by defining the boundaries of the field, and selecting resources and optimisation algorithms. Furthermore, the offline planning tool uses a `FieldPartitioningTool`, implemented using MATLAB [69], to convert the `Field` into a graph representation that can be processed by the `HarvestSimulator`. Finally, the offline planning tool uses the visualiser to enable inspection of the harvest plan, once it has been calculated.

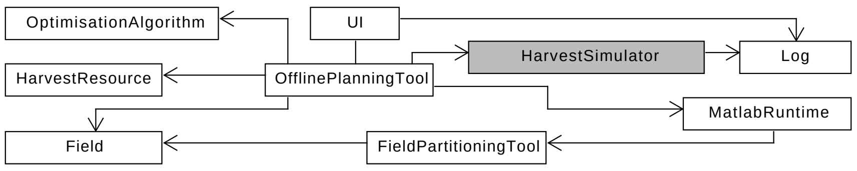


Figure 5.6: The structure of the offline planning tool.

Once the Java code generator has been configured, the delegate and test generation features enable fully automated system realisation and validation of VDM models that integrate real system components. Hence, the harvest simulator model and the system realisation can both be developed using the development practices of Continuous Integration (CI) [29]. That is, for every change made to the model, the harvest simulator is code generated and tested using the generated JUnit4 tests. The build and test process is handled by a Jenkins [50] integration server. This helps to ensure that the system works as intended and avoids problems caused by developers that lack the discipline to run the tests.

Contribution 12. A harvest planning system developed using the delegate and test generation features.

5.4 FAD Code Obfuscation

Code generated VDM traces have potential to allow a larger number of tests to be executed since the tests are run as compiled code rather than being interpreted. This section presents excerpts from the case study described in [P80]. The goal of this case study is to analyse the properties of an algorithm used to obfuscate FAD codes as well as to study the performance of code generated traces. The investigation therefore compares the execution times of code generated traces to those obtained by executing the same trace using Overture and VDMJ.

One of the interesting aspects of the FAD code case study is that the analysis of the algorithm involves generating and executing one million trace tests from a VDM-SL model, which code generated traces enabled us to do. However, concurrently with our work, the trace expansion algorithm used by VDMJ was significantly improved, which allows traces to be executed much more efficiently. VDMJ is the only VDM interpreter that currently supports this expansion algorithm.

This case study investigates a scenario from the industry where a customer asked to consider a way to obfuscate FAD codes such that the generated codes (1) were still six digit numbers (2) remain unique per branch, and (3) the entire 0-999999 range was still available. Although this is the same as creating a permutation on the list of all the FAD codes and using it as a look-up table, the obfuscation had to be done using a lightweight calculation. Based on this, the designers of the algorithm thought that the obfuscation could be defined by means of an injective mapping of the digits 0-9 onto themselves, but without any digit mapping to itself. The idea was to use this mapping to transform the individual digits of a FAD code. Although it was believed that this would meet the requirements of the algorithm, it was decided to analyse this approach using a VDM-SL model. Relevant parts of the FAD code model are shown in listing 5.3. The VDM-SL model defines an arbitrary digit mapping that is used to obfuscate FAD codes using the `convert` function. The `AllDifferent` trace is used to check that the set of obfuscated FAD codes meets the requirements when the injective map is applied to every possible FAD code.

Although the trace in listing 5.3 has no combination of cases it still produces one million tests when `SIZE` – the number of FAD code digits – is set to six. A test passes if the digit map `DM1` invariant holds, and when the map is applied to a FAD code, the obfuscated FAD code must be a different value, as required by the `convert` post condition.

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

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

Listing 5.3: Excerpts from the FAD code model.

5.4.1 Performance Results

To study the performance gained by code generation, the trace in listing 5.3, was executed for different FAD code sizes (by changing the `SIZE` value) using code generated traces, Overture and VDMJ. After each execution time had been measured the scenario was repeated to confirm that the tool did not suffer from memory starvation, as this would produce a misleading execution time. A complete description of how the experiment was carried out is provided in [P80]. Some of the details have been omitted from this section due to space limitations.

The execution times, for the different VDM tools and FAD code sizes, are listed in table 5.1 and visualised using a logarithmic plot in fig. 5.7. The

Table 5.1: Execution times from the FAD code analysis.

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

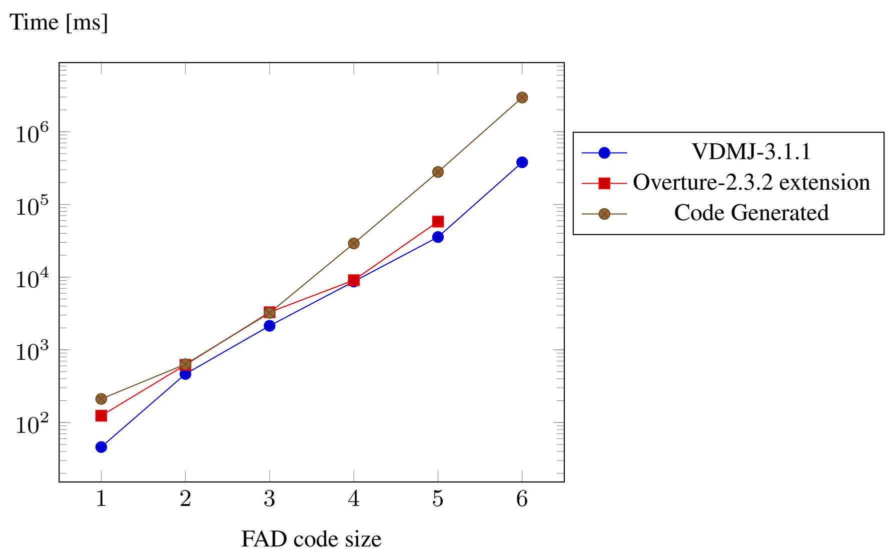


Figure 5.7: Logarithmic data plot visualising the execution times in table 5.1.

one scenario that did not complete, due to the tool running out of memory, is specified as “failed” in table 5.1.

5.4.2 Discussion of the Performance Results

As shown in table 5.1, Overture ran out of memory and failed to expand the one million tests that are produced for six digit FAD codes. The code generated version of the trace, on the other hand, was capable of executing all one million tests in over $2,900$ s, or 49.22 minutes. When comparing the execution times of code generated traces to those obtained using Overture, the results show that Overture executes the tests faster for FAD codes that consist of five digits or less. The reduced memory consumption is therefore the only advantage that is currently gained by running the code generated version of the trace. Naturally, this does meet the expectation that the compiled version of the trace would run faster. Comparing the execution times of code generated traces to those obtained using Overture provides a fair indication of the performance gain since both tools use the same algorithm to expand the trace.

VDMJ is the fastest tool to execute the one million tests. It achieves this in over 379 s, or 6.33 minutes. Unlike Overture, VDMJ manages to execute the one million tests because it uses an algorithm that is more memory efficient. This algorithm was released with the newest version of VDMJ (version 3.1.1), concurrent with our work. Older releases of VDMJ use an algorithm similar to that used by Overture.

For the reasons given in section 4.2, the code generated traces were executed using the OpenJML runtime assertion checker, version 0.6.3. To investigate the performance overhead introduced by OpenJML, the code generated version of the trace was compiled and executed as a plain Java program, i.e. without using the OpenJML compiler and runtime assertion checker. When the trace is executed in this way none of the JML annotations are checked at runtime. The point of doing this is to completely remove any overhead caused by OpenJML. The execution of the trace as a plain Java program only takes 33.94 seconds. If the expansion algorithm used by the code generator was changed to that used by VDMJ the execution time would mostly likely be significantly reduced. This is expected since the current results indicate that VDMJ scales better than Overture when the number of tests increases.

As another experiment, all the JML annotations were removed from the generated code. Afterwards the annotation-free code was compiled using OpenJML and the trace was executed using the OpenJML runtime assertion checker. The purpose of this is to remove the overhead directly associated with checking the JML annotations, while focusing solely on the overhead caused by OpenJML. It is worth noting that even though the JML annotations are removed, OpenJML still guards against variables and fields that hold the value **`null`**, which is not allowed by default. When the JML annotations are removed from the generated code the OpenJML runtime assertion checker executes the one million tests in 11.15 minutes. This is surprising as one would expect the execution time to approach that obtained by running the code generated version of the trace as a plain Java program (33.94 seconds).

These experiments lead to believe that the disappointing performance results are heavily influenced by the OpenJML tools.

The conclusion drawn in [P80] states that two things must be changed to improve the performance of code generated traces. First, the expansion algorithm used by the code generator must be replaced by that used by VDMJ, which is available as open-source. Secondly, the technology used to validate the generated code against the VDM constraints must be changed to one that performs better. Section 6.5.3 provides a more detailed discussion of the future plans for the trace code generator.

Contribution 13. Performance analysis of code generated traces executed using OpenJML.

6

Conclusion

This chapter concludes the dissertation and outlines future work. The research contributions presented in chapters 2 through 5 are evaluated to assess whether the research objectives, described in chapter 1, have been achieved.

6.1 Introduction

This dissertation presents a number of contributions that enhance system realisation in formal model development. The contributions take a starting point in the challenges of realising a formal specification and develop tool support to achieve integration of formal methods tools into the different phases of the software development life cycle. The tools have supported three case studies from externally funded projects. The feedback received from the case study work has helped to improve these tools and develop new ones.

This chapter summarises and evaluates the research contributions, produced over the course of this PhD, against the research criteria described in section 1.6. Based on the evaluation, the hypothesis is revisited to assess whether the research objectives have been achieved, and future plans for improving the research contributions are also discussed.

Following this section, an overview of the different contributions is provided in section 6.2. Subsequently, the contributions are evaluated in section 6.3. Next, it is discussed to what extent the objectives of the PhD have been achieved in section 6.4, and finally future plans are presented in section 6.5.

6.2 Research Contributions

The contributions produced over the course of this PhD are grouped into the four main areas *The Overture Platform*, *Code Generation*, *Contract-based*

Validation and *Case Studies* each of which corresponds to a chapter with the same name.

The contributions are shown organised into these four categories in fig. 6.1. This figure further shows how the different contributions relate and contribute to one another (as indicated by the arrows). All contributions contribute either directly or indirectly to Overture, which is why [C1] is highlighted in fig. 6.1. Based on the four categories, a summary of the contributions is provided below.

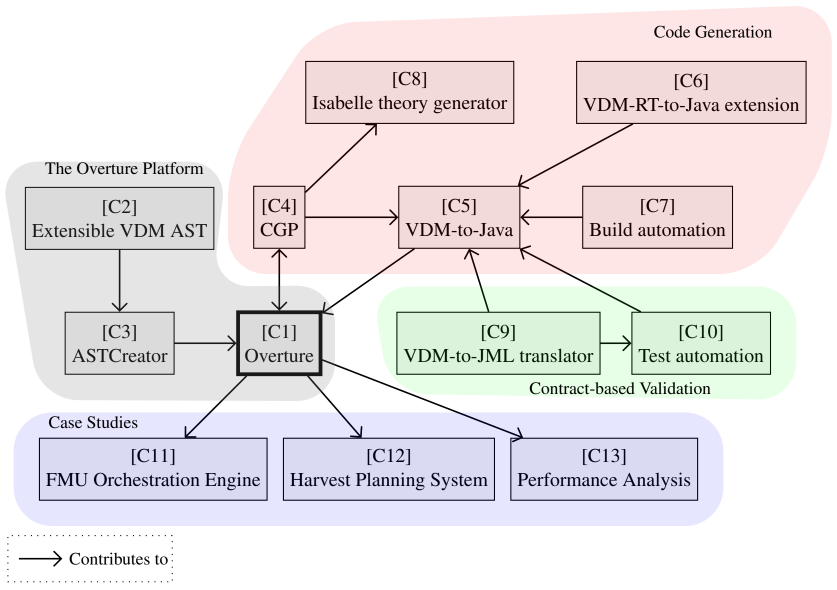

Code Generation
[C8] Isabelle theory generator
[C6] VDM-RT-to-Java extension
The Overture Platform
[C2] Extensible VDM AST
[C4] CGP
[C5] VDM-to-Java
[C7] Build automation
[C3] ASTCreator
[C1] Overture
[C9] VDM-to-JML translator
[C10] Test automation
Contract-based Validation
Case Studies
[C11] FMU Orchestration Engine
[C12] Harvest Planning System
[C13] Performance Analysis
Contributes to

Figure 6.1: Overview of the contributions.

6.2.1 The Overture Platform

Chapter 2 presents three contributions that cover architectural changes that promote extensibility and reuse for formal methods tools. The first contribution is the Overture platform, which supports the development of formal methods IDEs [C1]. The development of the Overture platform has involved re-designing Overture's AST in order to improve the extensibility and reuse of the underlying formal language [C2]. The re-design of the AST has enabled the development of CML and the Symphony tool in the COMPASS project. The experiences gained from using the extensible AST in this project have been distilled into a set of principles for reuse in formal methods tools, supported by AstCreator [C3]. The architectural changes made to the Overture platform have enabled the development of the remaining contributions produced during this PhD.

6.2.2 Code Generation

Chapter 3 presents five contributions that add code generation features to Overture. Common to all the code generators developed in connection with this PhD is that they use the CGP – an infrastructure for developing code generators for VDM [C4]. The CGP provides a framework (based on AstCreator) that offers functionality commonly used by code generators. Experiences show that this approach has the potential to reduce the efforts needed to develop new code generation features (see section 3.5).

Overture's Java code generator is the largest code generator project developed so far that uses the CGP [C5], although several other projects are currently under development (see section 3.6). The Java code generator forms the basis for other code generation related projects. For example, the VDM-RT-to-Java code generator extension adds support for the distributed aspects of VDM-RT [C6].

Motivated by the case study work (see chapter 5), the Java code generator was extended with functionality to enable automated realisation of a VDM specification and testing of its associated realisation. This was achieved by integrating the Java code generator into the Maven build automation system using a Maven plugin. The Java code generator Maven plugin was further enhanced to support code generation of VDM specifications that include real system components – a technique referred to as the *delegate* [C7].

The development of the Isabelle theory generator [C8] demonstrates other usages of the CGP. Whereas other projects that use the CGP focus on code generation towards a system realisation, the Isabelle theory generator is used to export VDM specifications to Isabelle to take advantage of the functionality that this tool has to offer. Exporting VDM specifications to Isabelle has been used to discharge proof obligations – something that cannot currently be done using Overture alone.

6.2.3 Contract-based Validation

Chapter 4 presents two large contributions that build on top of the Java code generator. Rules for translating VDM contracts to JML annotations were proposed with the purpose to bridge the gap between an abstract specification, specified using VDM, and its Java implementation [C9]. This is considered the most significant scientific result of this PhD. In addition, tool support was developed as an extension of the Java code generator to automate the translation from VDM to JML. The JML translator produces a JML annotated Java program that can be validated using JML tools. To enhance test automation for VDM specifications, the Java code generator was extended to support code generation of traces in order to exercise the generated JML annotations [C10]. This approach therefore leverages the contracts of the VDM specification to enable combinatorial testing of code generated VDM specifications.

6.2.4 Case Studies

Chapter 5 presents three case study contributions. In the FMU orchestration engine case study from the INTO-CPS project, a rule checker was developed to validate the data exchanged between the FMUs interacting in a co-simulation [C11]. In the attempt to understand the FMI standard, the rules for data exchange between the FMUs were specified using VDM-SL, and implemented via code generation. To integrate the code generated version of the rule checker and the orchestration engine, the Java code generator was extended with Maven support.

The harvest planning system is the most ambitious case study developed using the tools produced during this PhD [C12]. This system is developed to assist farmers in optimising harvest operations using simulation-based predictions. The first version of the model was specified solely using VDM, which led to performance issues with the system. This was partly because the data structures and algorithms used to analyse the field did not scale well in terms of field size. To address these performance issues, the model was updated to represent the field as a graph and use well-established Java libraries, based on standard algorithms, to analyse the field. This resulted in a setup consisting of both model and real system components. The delegate and test generation features were used to produce the system realisation and the tests used to validate the generated code. This approach enabled the practice of CI to be applied to the model as well as the generated code.

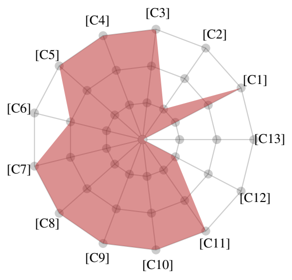
In the last case study, the performance of code generated traces, executed using OpenJML, was studied. Code generated traces enabled us to analyse the properties of an algorithm used to obfuscate FAD codes by execution of one million tests, derived from a trace. We did not manage to execute all one million tests using Overture because this tool ran out of memory during the expansion process. However, recent advances in trace expansion techniques, implemented by VDMJ, allow traces to be executed more efficiently than using other approaches. Since code generated traces are executed as compiled code rather than using a VDM interpreter, they were expected to run significantly faster. However, this expectation was not met, and there are indications that the poor performance results are related to OpenJML (see section 5.4.2). In section 6.5.3 future plans for how to improve the performance of code generated traces are discussed.

6.3 Evaluation of the Contributions

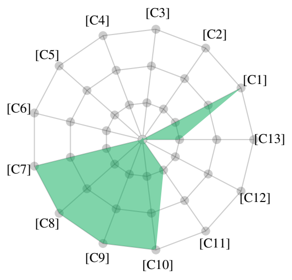
In this section the contributions are evaluated using the criteria described in section 1.6. The assessment is illustrated in fig. 6.2. Each of the individual evaluation criteria is represented using a spider chart to visualise the informal assessment. The evaluation of the contributions is a subjective assessment performed by the author of this dissertation. The spider charts show to what extent the criteria are fulfilled – the closer the scale is to the edge of the spider web, the better the contribution is considered to be. Figure 6.2e overlays fig. 6.2a, fig. 6.2b, fig. 6.2c and fig. 6.2d to provide an overall assessment for all criteria. The fulfilment of the individual criteria is discussed below.

6.3.1 Tool automation

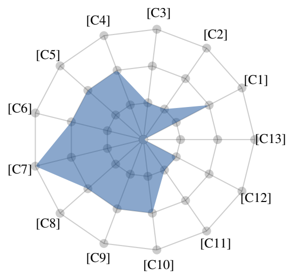
Most of the contributions enhance tool automation in formal model development. The VDM-RT-to-Java code generator [C6] adds support for the distributed aspects of VDM-RT, but the tool support for this particular contribution lacks maturity and is still under development. The development of FMI support for Overture allows VDM specifications to be exported in FMUs that can participate in FMI-based co-simulation [C11]. The feedback from the harvest planning system case study [C12] gave rise to the development of the delegate [C7] and test generation features. The extensible AST [C2] is regarded as a design that is suitable for AST generation but it does not contribute tool automation per se. Although the extensible AST is tightly connected with the AstCreator tool [C3], the latter is a separate contribution. The automation contributed by the tools produced as part of this PhD project, as illustrated in fig. 6.2a, therefore leads to the belief that this criterion is fulfilled to a high degree.



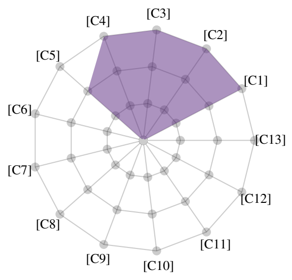
(a) Tool automation



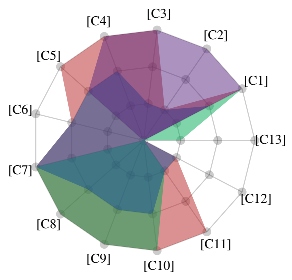
(b) System validation



(c) Tool integration



(d) Extensibility



(e) Combined criteria

Figure 6.2: Assessment of the contributions based on the evaluation criteria.

6.3.2 System validation

Overture is a formal modelling tool that includes several features for system validation [C1]. The Isabelle theory generator can be used to translate a VDM-SL specification and the associated proof obligations into Isabelle theories [C8]. In that way Overture can benefit from the verification features of Isabelle. In relation to the case study work, the orchestration engine rule checker [C11] supports system validation through co-simulation.

Improving the Java code generator in the context of build automation partly involved developing support for code generating VDMUnit tests to JUnit4 to achieve full reuse of the model tests. The code generated tests can subsequently be executed via the Maven build automation system [C7]. The JML translator enables validation of the code generated version of the system against the JML annotations derived from the formal specification [C9]. Code generated traces further allow the generated JML annotations to be exercised exhaustively [C10]. Finally, in the case study work it was shown how code generated traces can be used to analyse a system using combinatorial testing [C13]. The development of these contributions, especially those covered by chapter 4, therefore leads to the belief that this criterion is fulfilled to an acceptable degree.

6.3.3 Tool integration

The Overture platform promotes tool integration via its plugin-based architecture [C1]. Furthermore, the extensibility features of the AST [C2] and AstCreator [C3] enable integration of other formal languages.

The CGP [C4] supports integration with other technologies through code generation. The Java code generator [C5], the VDM-RT extension [C6] and the Isabelle theory generator [C8] all use the CGP for this purpose. Similarly, the JML translator [C9] and the trace code generator [C10] build on top of the Java code generator in order to achieve integration with Java and JML.

The extension of the Java code generator to support Maven [C7] is considered the most significant contribution in terms of tool integration due to the value it brings to the FMU orchestration engine and the harvest planning system case studies. The integration with Maven was developed to (1) expose existing code generation features via Maven and (2) add new functionality to support CI and code generation of modelling setups that integrate real system components. The inspiration for the Maven integration comes from the needs identified during the case study work. The contributions have demonstrated that they promote tool integration, and the corresponding criterion is therefore considered fulfilled to a satisfactory degree.

6.3.4 Extensibility

The architecture of the Overture platform is plugin-based, which means that new functionality can be added without changing existing components [C1]. Overture was used in the COMPASS and DESTECS projects to develop the Symphony and Crescendo IDEs (see section 2.2.3). VDM language extensions are enabled through the extensible AST [C2] supported by AstCreator [C3]. In the COMPASS project CML was developed as an extension of VDM using AstCreator.

The CGP [C4] uses AstCreator to generate the IR in order to take advantage of the extensible AST design. The advantages of using the CGP were demonstrated by analysing the Isabelle theory generator in terms of LoC. The analysis showed that a framework-based approach has the potential to significantly reduce the efforts needed to develop code generators.

Transformations enable one to add new functionality to code generators. The delegate concept and test generation features were both developed as separate transformations that were added to the transformation series of the Java code generator [C5]. The Java code generator is itself designed with extensibility in mind – in particular to support the development of the JML translator and the VDM-RT-to-Java code generator extension. The extensibility of the remaining tool contributions [C6] through [C10] have not been investigated. Most of the contributions covered by chapter 2 and chapter 3 have, however, demonstrated that they support the development of new tool extensions, and the corresponding criterion is therefore considered fulfilled to an acceptable degree.

6.4 Revisiting the Hypothesis

The hypothesis of this PhD, presented in section 1.5, stated that:

The automation of the steps involved in realising and validating a system based on a formal specification can be improved through use of properly designed tool support that seeks to (1) improve the integration of formal methods tools into the software development life cycle and (2) leverage the system properties described by the formal specification.

Based on the challenges inherent to realisation of formal specifications, a number of contributions were presented with the purpose to enhance system realisation in formal model development. The contributions to code generation and build automation specifically help improve the integration of tools into the software development life cycle. Based on the feedback from the case study work, the tool contributions were integrated into build environments to support the automation of the steps involved in developing the FMU orchestration engine and the harvest planning system case studies. This approach enables seamless integration of all the system components, including the code that is generated from the formal specification, in order to construct the final version of the system.

The contributions related to system validation leverage the properties described by the formal specification to improve validation of the system realisation. The contributions achieve this by allowing these properties to be used to test the generated code – for example using combinatorial testing. In the case study work, the possibility to use the model tests to validate the generated code, was considered the most useful validation feature, contributed by this PhD.

From a general perspective the hypothesis is difficult to validate in a PhD project within a three-year period. In particular because the hypothesis only has been tested using three main case studies, where the author of the dissertation has been involved. It would have been a different situation had the contributions been used by external practitioners who work under different circumstances. However, under the circumstances that this PhD project has been carried out, the evaluation of the contributions leads to the belief that the hypothesis is valid. It is the hope that the results produced during this PhD project will inspire other researchers and assist practitioners in the area of formal modelling, and MBD development in general, in improving their development practices.

6.5 Future Work

This section describes future directions for some of the contributions.

6.5.1 Re-designing Transformations

The IR used to represent the code generated version of the VDM specification is independent of any target language. Transformations operate on the IR and can therefore be reused among code generators. As mentioned in section 3.6, the VDM-to-C code generator uses some of the transformations that initially were developed to support the VDM-to-Java code generator. Although this approach has potential to reduce development efforts when implementing code generators, it is faced by several challenges.

One thing that challenges reuse of the transformations currently available is that they generally try to solve too many problems at once. For example, some transformations are too extensive in terms of the number of transformed language constructs. Ideally a transformation should only treat a small problem, i.e. the transformation should be *atomic*. As an example, an atomic transformation could transform implications on the form $A \Rightarrow B$ into the semantically equivalent expression $\neg A \vee B$. This transformation could be used to eliminate implications whenever this operator is not supported by the target language – which is the case for languages such as Java and C.

Other challenges related to reuse of transformations are caused by transformations making assumptions about the structure of the IR – for example by assuming that certain language constructs will (or will not) exist in the IR.

That is, when a transformation is applied to the IR the transformation sometimes assumes that other transformations have been applied to the IR previously. Currently there exists no good way to resolve dependencies among transformations.

Naturally, it becomes increasingly difficult to reuse a transformation as it makes more assumptions and becomes less atomic. This future work item proposes re-designing transformations to make them atomic and to find a better way to resolve dependencies among transformations.

6.5.2 Improving the Integration between Overture and Isabelle

The current version of the Isabelle theory generator [C8] produces a set of theories and proof goals that can be analysed using Isabelle. Extending the Isabelle embedding to cover VDM++ and VDM-RT, in addition to VDM-SL, is among the improvements that are currently under development in the INTO-CPS project [33].

One way to improve the integration between Overture and Isabelle is by making the communication between the two tools transparent to the user. This can be achieved by having Overture pass the theories and proof goals to Isabelle, which communicates the proof results back to Overture. The idea is to have Overture handle all user interaction and presentation of proof results. In addition to developing the communication between the two tools, this work-flow involves translating the proof results back to VDM.

6.5.3 Improving the Performance of Code Generated Traces

The performance figures obtained by running the code generated traces are disappointing, and there are things that indicate that the poor performance is related to OpenJML (see section 5.4.2). It is therefore worthwhile considering other ways to represent VDM constraints in the generated code. One way is to use Java assertions – without relying on a particular DbC technology such as OpenJML. However, it is not possible to directly use all the expressions generated by the JML translator in Java assertions. As an example, some of the generated JML annotations used to check VDM type constraints rely on certain JML constructs that are not available in Java natively.

Many JML tools, including OpenJML, do not support current versions of Java, which makes generating JML annotations less useful. However, it still requires a significant amount of work to keep updating the JML tools as the Java language and standard libraries evolve. Still, the fact that JML tools lack support for current versions of Java, makes it worthwhile considering use of other DbC technologies.

Another technology related to JML is Microsoft Code Contracts [79, chapter 15], which is used to express assumptions in .NET programs. Microsoft Code Contracts originates from Spec# – a language that extends C# with contract-based elements and a convenient DbC notation. To support all languages within the .NET framework, Code Contracts offers its features via tools and library functionality. Naturally, this leads to program specifications that are less concise than those written using a dedicated notation such as Spec# or JML. The Code Contracts project has become open-source and is being actively maintained, which makes it an appealing technology from the perspective of this PhD. Therefore, it would be interesting to translate VDM specifications to Code Contracts decorated C# programs and exercise them using C# code generated traces – especially to compare the execution times to those obtained using the trace code generator developed in this PhD project [C10]. The development of a VDM-SL-to-C# code generator that uses Code Contracts to represent invariants, type constraints, and pre- and post conditions has been developed in connection with a Master's thesis project (see section 3.6). However, code generation support for traces is work in progress for this code generator.

6.5.4 Updating the Harvest Planning System to Model Distribution

The current version of the harvest planning system [C12] assumes a global view where every resource (vehicles, storage facility etc.) participating in the harvest operation has access to global information about the field and the other resources. A new version of the system will model the distributed aspects of the system by taking into account that resources form constituent systems that communicate with one another through a network in order to establish a local view on how the harvest is progressing. In a realistic setting several things can go wrong during a harvest. For example, vehicles may break down and become unavailable, which necessitates re-planning.

So far the global view approach has been modelled using VDM++, and code generated using the VDM-to-Java code generator [C5]. To model the distributed aspects and move away from a global view, the model will be updated to use VDM-RT, which gives rise to new challenges related to both modelling and system realisation. The VDM-RT model can in principle be code generated using the VDM-RT-to-Java code generator extension [C6], although things such as deployment and testing of the system realisation also need to be taken into account. This may result in the development of new code generation or build automation features.

Part II

Publications

7

Towards Enabling Overture as a Platform for Formal Notation IDEs

This paper has been accepted as a peer-reviewed workshop paper.

[P18] Luís Diogo Couto, Peter Gorm Larsen, Miran Hasanagić, Georgios Kanakis, Kenneth Lausdahl and Peter W. V. Tran-Jørgensen. *Towards Enabling Overture as a Platform for Formal Notation IDEs*. 2nd Workshop on Formal-IDE (F-IDE), June, 2015.

8

Migrating to an Extensible Architecture for Abstract Syntax Trees

This paper has been accepted as a peer-reviewed conference paper.

[P22] Luís Diogo Couto, Peter W. V. Tran-Jørgensen, Joey W. Coleman and Kenneth Lausdahl. *Migrating to an Extensible Architecture for Abstract Syntax Trees*. 12th Working IEEE / IFIP Conference on Software Architecture (WICSA 2015), May, 2015.

9

Principles for Reuse in Formal Language Tools

This paper has been accepted as a peer-reviewed conference paper.

[P24] Luís Diogo Couto, Peter W. V. Tran-Jørgensen and Kenneth Lausdahl. *Principles for Reuse in Formal Language Tools*. 31st Annual ACM Symposium on Applied Computing (SAC), April, 2016.

10

A Code Generation Platform for VDM

This paper has been accepted as a peer-reviewed workshop paper.

[P55] Peter W. V. Jørgensen, Luís Diogo Couto and Morten Larsen. *A Code Generation Platform for VDM*. 12th Overture Workshop, June, 2014.

A Code Generation Platform for VDM

Peter W. V. Jørgensen, Morten Larsen, and Luis D. Couto

Department of Engineering, Aarhus University, Denmark
{pvj,mola,ldc}@eng.au.dk

Abstract. In this paper we describe ongoing work on a code generation platform that simplifies the construction of code generators for VDM in the Overture tool. The platform represents the code generated model as an Intermediate Representation (IR) and assists a code generator in transforming the IR into a structure that is easier to code generate. Since the IR is independent of any target language, a code generator can choose the transformations it needs to obtain the IR it desires. Based on the code generation platform a VDM++ to Java code generator has been developed[1], while early work is currently being made on a C++ code generator. Implementing the Java and C++ code generators has provided useful feedback for the architecture of the code generation platform. This has helped us to generalise the platform structure in order to make it a stronger foundation to use for constructing code generators.

Keywords: VDM, code generation, language paradigms, intermediate representations, tree transformations, extensibility, Java, C++

1 Introduction

When resources have been invested into modelling a system it is desirable to code generate the software implementation or parts of it from the system model to reduce the efforts needed to realise the system. Code generation therefore supports efficient transitioning to the realisation phase. However, most importantly it minimises the chances of introducing inconsistencies in the software implementation that makes it deviate from the system specification due to manual translation of the model into code.

With the existence of many popular target languages it is common for code generators to provide support for multiple target languages in order to target a larger group of users. This can, however, easily lead to duplication of efforts when implementing code generators — especially if the target languages follow the same paradigms such that the rules used to code generate a source language are the same.

Ideally it should be possible to reuse the transformations used to code generate constructs of a source language. As an example, consider the VDM set comprehension {`x|x` **`in set`** `S & pred(x)`}, which constructs a new set from the elements of `S` for which `pred(x)` is true. In imperative languages such as Java and C++ this language construct is non-trivial to code generate since Java and C++ do not have similar constructs included. The same functionality can be obtained in those languages, but it

[1] The Java code generator is available in Overture releases 2.1.0 onwards

requires use of multiple language constructs for iterating over a set, evaluating a predicate on each set member, adding elements to a resulting set and so on.

The potential for different *backends* (a code generator that extends the code generation platform) to use the same transformations is particularly good when the target languages belong to the same paradigm (e.g. they are object-oriented or functional in style). In that case they will have many language constructs in common and thus face many of the same challenges with respect to code generation. When the same transformation can be used by different backends to code generate a source language construct it is beneficial to apply the transformation to the code generator input before it reaches the backend in order to obtain a transformed structure that is easier for a backend to code generate. The idea is therefore to structure a code generator such that it is possible to select the transformations that will lead to a structure that requires the least effort for a backend to code generate. In this paper we explore this approach to code generation in order to reduce the efforts needed to implement code generation for multiple backends.

The paper is structured as follows. Section 2 describes how IRs support the implementation of code generators or backends. Section 3 explains how tree transformations simplifies the implementation of backends. Section 4 provides an overview of the code generation platform used for implementing the Java and C++ backends. Section 5 and Section 6 describes challenges encountered for the implementation of the Java and C++ backends, respectively. Section 7 describes future plans for the code generation platform. Section 8 describes related work and finally section 9 concludes our work.

2 Intermediate Representations

One approach adopted by compiler developers is to transform the Abstract Syntax Tree (AST) specified in the source language into an IR that preserves the semantics of the input and from which the backend generates code in the target language. The IR helps managing the complexity of the compilation process by being independent of details specific to the source language and the target language. An IR obtained from the VDM AST serves a similar purpose by mitigating the complexity of generating code from a VDM model. This would, for example, enable the code generator to unify VDM functions and operations into the concept of a method as seen in a programming language such as Java. Then the backend only needs to treat a single (language) construct without having to distinguish between functions and operations.

Code generating a VDM construct is easier if equivalent or similar constructs appear in the target language. For example, a set comprehension in VDM is more likely to have an equivalent construct in a functional programming language, which would simplify the task of code generating it. However, for a target language that does not support set comprehensions, code generating this construct is non-trivial. This is not surprising since Java is an imperative language and a set comprehension is a functional concept. Similarly, code generating the object-oriented concepts of VDM to a functional language will require constructs to be code generated that are not naturally expressed in terms of the target language. In general it is difficult to code generate across paradigms since a construct with a strong relation to one paradigm will not be present in other paradigms thus requiring a strategy to translate that construct, which potentially needs to make widespread changes to the IR.

In this work we address the challenges of code generating constructs where no obvious mapping exist. We do this by translating the VDM AST into an IR to which a series of transformations are applied. By transforming language constructs that are difficult to code generate into new (possibly larger) tree structures, based on concepts that are easier to code generate, the implementation of a backend can be simplified. If the backend provides support for code generating the replacement constructs used by the transformations then it follows that the backend already supports code generation for the complex construct. In that case the complexity of the code generation process is comprehended entirely using tree transformations. The advantage of this approach is that the transformations can be made such that they are independent of the target language. This enables other backends to benefit from the same transformations when code generation is implemented for other languages.

3 Tree transformations

In order to show the usefulness of applying transformations to the IR, before handing it to the backend, we will consider a set comprehension as an example of a construct to code generate. Since the set comprehension is a functional concept many target languages, such as those that are imperative in style, need to use several different constructs to obtain the equivalent functionality.

3.1 Code generating the set comprehension

The VDM snippet in Listing 1 shows an example where a set comprehension is used to construct a new set obtained by iterating over the set `S` and selecting the elements for which `pred(x)` is true. Therefore collection comprehensions provide a convenient notation to construct collections from other collections which would otherwise require use of several different constructs in an imperative language such as Java.

```
public f : () -> set of nat
f () ==
let a = {x | x in set S & pred(x)}
in g(a,a);
```

Listing 1. Example of a set comprehension in VDM

Without using the set comprehension the equivalent functionality can be obtained by rewriting the function in Listing 1 into a VDM operations that explicitly specifies the semantics of the set comprehension using an imperative style of writing as shown in Listing 2. Due to the expressiveness of the set comprehension Listing 1 obtains the same functionality as that of Listing 2 using fewer lines of VDM. Although the function in Listing 1 and the operation in Listing 2 are semantically equivalent, the listings represent different challenges for a code generator. The reason for this is that the two VDM snippets use different constructs to obtain the same result.

```
public op : () ==> set of nat
op () == (
dcl setCompResult : set of nat := {};
for all x in set S do
  if pred(x) then
    setCompResult := setCompResult union {x};
(dcl a : set of nat := setCompResult;
 return g(a,a)));
```

Listing 2. Imperative specification of the VDM set comprehension in Listing 1

Therefore, the difficulty of code generating a VDM model depends on the style of modelling and the target language. VDM is a multi-paradigm modelling language, using constructs of both the object-oriented and the functional paradigm, and therefore backends will experience situations where a construct does not have a one-to-one mapping into the target language.

3.2 Transforming language constructs

One approach to simplifying code generation of a VDM model is to have the modeller refine the model such that it uses constructs that are easier to code generate. However, eliminating constructs that are problematic to a code generator using model rewriting, limits the modeller to use only a subset of the source language. This also clutters the model with details used to assist the backend in generating code from the model, thus going against the point to have a model that abstracts away details that do not contribute to obtaining the insight needed.

A more sophisticated approach is to have this kind of model refinement done at a later stage to make it transparent to the modeller and avoid restricting modelling to only a subset of the source language. This could be done by applying transformations to the IR such that constructs that are problematic to code generate get replaced with other IR constructs in order to obtain a *simplified IR* that is easier to code generate.

The use of transformations is part of a larger platform architecture that is used to construct backends. In section 4 the architecture of this code generation platform is detailed to make it clear how it facilitates the construction of code generators.

4 Architecture of the code generation platform

The code generation platform, shown in Figure 1, takes a VDM++ model as input and use it to construct an IR that represents the generated code. After the IR has undergone a transformation process it is input to a backend that translates it into source code in a target language. To further detail the approach taken to construct code generators this section describes the architecture of the code generation platform and how it interacts with the backend of a target language.

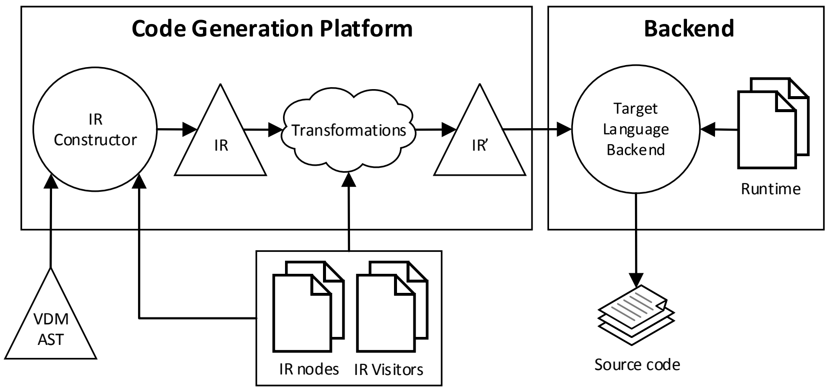


Fig. 1. An overview of the code generation platform architecture

4.1 The intermediate representation life cycle

The IR as first constructed from the VDM AST represents a slightly simplified and extended version of the VDM model. For example, records in the IR are allowed to have methods (unlike records in VDM). The purpose of this will become more clear in subsection 5.1 where it is discussed how VDM records are code generated to Java.

The IR simplifies the tree structure by eliminating or rewriting use of certain operators by replacing them with use of other operators. For example, writing a logical implication on the form $A \Rightarrow B$ where A and B are propositions, is convenient in a mathematical language such as VDM, but since it is a derived and, not elementary operation of boolean logic, this operator is rarely seen in a programming language. Therefore the expression $A \Rightarrow B$ is represented as $\neg A \vee B$ in the IR, which is semantically equivalent.

Afterwards, code generation enters the transformation process where constructs that are difficult to code generate (or even unsupported by the backend) are translated into new tree structures that can be code generated. The IR before and after it has undergone the transformation process is denoted `IR` and `IR'` in Figure 1, respectively. Finally, the simplified IR is input to the backend that translates it into a target language.

4.2 The design of the intermediate representation

The IR nodes are generated using the ASTCreator tool [1], which is a SableCC [9] inspired tool. As shown in Figure 2 the ASTCreator takes a description of the AST as input and outputs nodes from which concrete ASTs can be constructed. The generated AST structure uses bidirectional node relations which make it easier to search the tree both upwards (e.g. finding the enclosing class of a node) and downwards (e.g. looking up type information of child nodes). The nodes also have functionality for making changes to the tree structure, which is needed when nodes must be replaced with new tree structures during the transformation process.

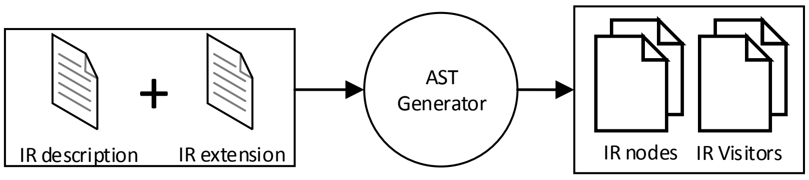


Fig. 2. The ASTCreator produces the IR nodes and visitors based on the IR description

The ASTCreator also produces functionality to traverse the AST. Tree walkers or visitors are implemented using the visitor pattern and play an important role in the current AST architecture used in the Overture tool where they, for example, are used to implement the type checker and the interpreter [6]. Similarly, the `IR Constructor` is a visitor that traverses the VDM AST and constructs the `IR` from it.

The ASTCreator is designed such that it allows optional AST extensions to complement an existing AST description and therefore it is possible to add new nodes to the IR. This design benefits the extensibility of the code generation platform since extensions to the IR can be made to support the implementation of additional transformations.

4.3 The backend

The final step of the code generation process translates IR constructs into source code of the target language. When transformations have simplified the IR then ideally these mappings should be trivial. The process of mapping IR constructs into source code of the target language for the Java backend is done using the template based technology, Apache Velocity [11]. Optionally, the generated code can make use of a runtime. As an example, the Java backend includes a runtime to represent VDM types and implementation for some of the VDM operators such as the sequence modification.

5 Code generating Java from VDM++

Java has fewer language constructs than other object-oriented languages such as C# and C++, which makes it simpler, but also less expressive as a language. Such languages are difficult to code generate since fewer constructs in a target language implies less ways to code generate a source language. This has led to some instructive experiences when implementing the Java backend, some of which we will discuss in this section.

5.1 Code generating value semantics

In VDM records, tuples and collections have copy-by-value semantics (referred to as “value semantics” throughout the remainder of the paper), which is the behaviour where a variable is copied when it is passed as a parameter or appear on the right-hand-side of an assignment. This is also the semantics used for structs in programming languages such as C++ and C#. In Java where there is no direct support for structs (or something similar) the equivalent can be obtained by representing a value type using a class and then have the instances explicitly copied as needed – normally by invoking a method on the instance that does the copying. Therefore, code generating value types to Java require careful attention. To demonstrate this, consider the VDM snippet in Listing 3, where the record type `Vector2D` is used to represent two-dimensional vectors. In this listing two vectors `v1` and `v2` are created with `v2` being a copy of `v1`. Since records have value semantics subsequent modifications to `v1` have no effect on `v2` and therefore the operation would return 1.

```
public op : () ==> nat
op () == (
dcl v1 : Vector2D := mk_Vector2D(1,2);
dcl v2 : Vector2D := v1; -- Copy using value semantics
v1.x := 2;
return v2.x;)
```

Listing 3. Value semantics in VDM demonstrated using records

Had the vectors in Listing 3 been modelled using a class rather than a struct then `v2` and `v1` would have been object references pointing to the same object. Therefore, subsequent modifications to the underlying object using any of the two object references would effect the same object and in that case the operation in Listing 3 would return 2.

5.2 Obtaining the effect of value semantics in Java

To obtain the effects of value semantics using a Java class one can provide a `clone` method and invoke it on the instances when they need to be copied. Therefore, code generating the VDM operation in Listing 3 yields the Java code shown in Listing 4. Note, how the backend invokes the generated `clone` method in order to ensure that the copy of `v1` respects the rules of value semantics. Since the responsibility of the `clone` method is to copy the fields of the associated class it must be generated specifically for each record in the IR.

```
public Number op() {
  Vector2D v1 = new Vector2D(1L, 2L);
  Vector2D v2 = v1.clone();
  v1.x = 2L;
  return v2.x;}
```

Listing 4. Java code generated from Listing 3 demonstrating how the Java backend obtains the effects of value semantics

A record in the IR can have methods (unlike records in VDM, which only have fields) and therefore the clone method can be added as a child to the record node. Similarly, the Java backend adds a method for record comparison based on structural equivalence (field-wise comparison), a method for calculating the hash code of a record (such that it is suited for use in collections) and a method that computes the string representation of the record. Since these methods are added as extensions to records in the IR they are specified solely using IR nodes.

5.3 Code generating functional concepts in Java

The approach of applying transformations to the IR has enabled the Java backend to code generate complex constructs without being aware of their presence in the VDM model. The reason for this is that these constructs are transformed into tree structures composed of IR nodes that are simpler to code generate. In that sense code generating the functional concepts required no extra effort for the implementation of the Java backend since code generation for the simpler constructs was already supported. The result of code generating the set comprehension in Listing 1 is shown in Listing 5.

```java
public static VDMSet f() {
  VDMSet setCompResult_1 = SetUtil.set();
  VDMSet set_1 = S.clone();
  for (Iterator iterator_1 = set_1.iterator();
    iterator_1.hasNext();) {
    Number x = ((Number) iterator_1.next());
    if (pred(x)) {
      setCompResult_1 = SetUtil.union(
        setCompResult_1, SetUtil.set(x));
    }
  }
  VDMSet a = setCompResult_1;
  return g(a, a);}
```

Listing 5. Java code generated from the VDM function in Listing 1

5.4 Iterating over collections

Iterating over collections in a target language is often done using library classes specific to that language. The Java backend does this using the `java.util.Iterator` class, as shown in Listing 5, whereas the C++ backend uses the C++ standard library iterators (`std::iterator`). Since transformations may produce new tree structures that iterate over collections the code generation platform enables transformations to be configured with language specific ways to iterate over collections. Iteration strategies, as they are termed, have been added to the code generation platform as a result of the feedback from implementing the C++ backend, and the Java backend has been updated accordingly such that it also uses the iterator strategies.

Language iterators must implement a language iterator interface that requires implementation of methods to

1. Initialize the iterator (or counter) used to perform the iteration
   – e.g. `Iterator iterator_1 = set_1.iterator();`

2. Build the expression used to determine whether there are more elements to process
   – e.g. `iterator_1.hasNext();`
3. Increment the iterator (or counter) and read the next element
   – e.g. `Number x = ((Number) iterator_1.next());`

The language iteration strategies allows incrementation of the iterator and reading the next element (the third item) to be done in separate steps (increment the iterator and then read the next element), but in Java and C++ it is common to do this in a single step – at least when using the built-in iterator classes. Each method constructs a tree to express the generated code related to that method using IR nodes. Since the trees generated by these methods may want to represent types that are external to the code generation platform the IR offers nodes to represent external constructs of the target language. For example, in order to allow easier integration with a target language the IR offers a construct to represent external types (e.g. the `Iterator` class in Java).

6 Code generating C++ from VDM++

The point of using a code generation platform is that backends of similar target languages can use the same functionality in order to reduce the efforts needed to implement code generation. The implementation of a C++ backend has provided useful feedback for the architecture of the code generation platform and given rise to some future plans that will be covered in section 7. In this section we describe some of the interesting challenges encountered for the implementation of the C++ backend and relates it to the work on the Java backend described in section 5.

6.1 Code generating reference semantics

In VDM classes use reference semantics, and therefore two object references are considered equal if they point to the same object. The same applies for object references in Java, but in C++ the objects must be referred to using pointers in order to obtain reference semantics. A C++ object allocated on the stack, on the other hand, uses structural equivalence (field-wise comparison) to determine equality.

When an object must be shared among multiple methods it is common to put it on the heap and access the object via a pointer or reference. In C++ memory that is allocated on the heap must also be deallocated explicitly by the programmer since C++ does not support garbage collection. In order to address this issue, the C++ backend implements an object reference using a shared pointer from the standard library (i.e. `std::shared_ptr`), which provides reference counting and automatic deletion when no more references for the underlying object exist. In Java, garbage collection is a language feature, and therefore the Java backend does not take memory deallocation into account.

6.2 Code generating functional concepts in C++

The C++ backend uses the same transformations as the Java backend to transform functional VDM language constructs (collection comprehensions, quantified expressions etc.) before they are code generated. Configuring the visitor that performs the transformations of the functional concepts with an iterator strategy and applying it to the set comprehension in Listing 1 yields the generated C++ code shown in Listing 6. Since the transformation takes care of expressing the algorithmic part of evaluating a set comprehension, and this can be reused directly by the C++ backend, the efforts needed for code generating the set comprehension in Listing 1, is only a matter of providing the iterator strategy – given that the simpler constructs can already be code generated by the backend (e.g. the if-statement).

```
vdm::set<int> f() {
  vdm::set<int> setCompResult_1 = vdm::set<int>::
      from_list();
  vdm::set<int> set_1 = S;
  for (vdm::set<int>::iter iterator_1 = set_1->begin();
      iterator_1 != set_1->end();  ){
    int x = *iterator_1++;
    if( pred(x) ){
      setCompResult_1 = vdm::set<int>::set_union(
          setCompResult_1, vdm::set<int>::from_list( x));
    }
  }
  vdm::set<int> a = setCompResult_1;
  return g(a,a);};
```

Listing 6. C++ code generated from the VDM set comprehension in Listing 1

6.3 Representing VDM records in C++

Record values in VDM are copied when assigned from, passed as argument (to a function or an operation), or returned as a value. This is also the behaviour for a class in C++, and therefore this construct is used to represent a VDM record. However, for a C++ class to create values from another class, the declaration of that class must be visible to the compiler (such that the size of the value can be computed), otherwise the class can only be pointed to (using a fixed size pointer). For example, consider a class `R1` that has a field of the class type `R2`. The declaration of `R2` must appear before the declaration of `R1` otherwise the C++ compiler raises an error. A partial solution to this, is to sort the dependencies using a topological sort. This does, however, require a graph with no directed cycles. Another solution is to treat records as classes and use the `std::shared_ptr` type and generate additional code to obtain the effects of value semantics as it was done by the Java backend described in subsection 5.2. Currently the C++ backend uses the first approach where topologically sorted C++ classes are used to represent VDM records.

6.4 Representing VDM collections in C++

The C++ standard library include lists, sets and maps but it lacks some of the functionality needed to fully represent the VDM collections. Therefore, the C++ backend uses a runtime that includes classes to represent the VDM collections (e.g. `vdm::set`) and operations on them. For example, the set union operator is implemented as a method, `set_union`, and used, for example, in the transformation of a set comprehension as shown in Listing 6. To code generate the includes needed in the generated code to access the runtime, the C++ backend uses the external type construct of the IR.

To ensure that the value semantics for VDM collections are preserved, the runtime collections overload the assignment operator and the copy constructor such that a collection gets copied correctly when assigned from, passed to a method or returned as a value. The advantage of this approach is that the C++ compiler becomes responsible for generating the code that copies the collection object the places where it is needed. This is different from the approach used by the Java backend, which needs to analyse the IR in order to find out where the `clone` method needs to be invoked. However, since Java classes use different semantics than C++ classes and Java does not allow operator overloading nor does it use copy constructors, the approach used by the C++ backend cannot be used.

7 Future plans

Looking forward, there are several immediate improvements that can be made to the code generation platform, in terms of expanding the coverage of VDM and adding support for additional target languages. There are also possibilities of looking into how the extensibility and the reuse of transformations can be improved.

7.1 Adding support for new target languages

We may wish to generate code for different languages of different paradigms to further validate the code generation platform architecture by implementing backends based on target languages that are different from Java and C++. Since Java and C++ are both imperative languages that use object-oriented principles, the work presented in this paper focuses on reusing the existing transformations. Adding support for a new language of a different paradigm would provide feedback for the code generation platform, which would lead to further improvements in terms of its extensibility.

7.2 Atomic transformations

One way to add support for new target languages is to continue expanding the code generation platform, by adding transformations and altering the existing ones to facilitate the support for new target languages. However, there are issues with this approach: the constant maintaining of existing functionality to support new target languages indicates a poor platform extensibility. Instead it should be possible to extend the existing functionality that the code generation platforms offers without affecting it.

In order to address this issue, the code generation platform must be re-designed with respect to the way transformations are applied. At the moment each transformation is large and extensive. For example, when transforming the functional elements in preparation for Java code generation, all elements are removed in the same visitor. This occurs at code level where all these transformations are implemented in one visitor.

To understand the consequences of this, consider the case where we wish to code generate for a language `JavaEQ` that is like Java in every regards, except that it contains support for existential quantifiers. In the current architecture this requires either subclassing and overriding the methods in the Java visitor (which immediately locks the new transformations in the Java hierarchy) or duplicating and changing code.

Therefore, we propose use of fine-grained *atomic transformations* that allow constructs in the IR to be transformed one at a time. These transformations would then be grouped in libraries in terms of which types of constructs they replace rather than which programming language they target. From here, we can define composite transformations that support specific programming language as combinations of the atomic transformations. For example, if we consider a transformation to be a relation from IR to IR then we would say that $JavaTrans = ExistsTrans \circ SetCompTrans \cdots$.

7.3 Extensibility of the code generation platform

Use of atomic transformations would also make it easier to maintain existing backends. For example, Java supports lambda expressions as of the recent Java 8 release[5]. In terms of atomic transformation, updating the Java backend to support Java 8 is as simple as removing the lambda expression transformation from the sequence of transformations. This would be significantly simpler (and shorter) than editing the visitor code to remove the transformation. In addition, if this visitor is being used to code generate for another language without lambda expressions, then the code must be split.

In order for this approach to be viable, the atomic transformations would have to respect various properties. Namely any two atomic transformations should be compatible with each other. This means that they alter independent parts of the tree while preserving anything else. An initial approach to this might be to ensure that no two transformation operate on the same node.

As certain transformations push the tree in a particular direction, other transformations no longer become available. This is acceptable since one is not interested in combining transformations arbitrarily but rather do so always with the goal of getting closer to a particular target language. There may also be issues with the order of the transformations. Again, it is not essential that all transformation can be combined in arbitrary ways. Only that there is one way to combine all the desired transformations.

There are some implementation challenges to the atomic transformation approach. Particularly since multiple inheritance is not available in Java (the language in which the transformations are implemented) and therefore the combination of multiple transformations in a single visitor is non-trivial. Finally, there may also be performance considerations when performing multiple small transformations versus a single larger one.

7.4 Code generating trace definitions

Overture supports automated test generation and execution of large collections of test cases that are derived from a trace definition [7]. The trace definition uses a short-hand notation and can be thought of as a regular expression that expands to test cases or sequences of operation calls that match the pattern of the trace definition. When the expansion process is complete the VDM interpreter executes the tests one by one and optimises the process by filtering out tests based on the outcome of other tests (e.g. a test will fail if it starts with a sequence of operation calls that it known to cause a test to fail). In addition, Overture offers different techniques to make reduced sets of tests as representative as possible – a technique known as shape reduction.

This future plan item aims to produce and execute code generated from a trace in order to make test execution more efficient. This can be done by expanding the trace into tests that are code generated and executed, instead of having them executed in the VDM interpreter. However, for this to be of any value it should make up for the time spent producing and executing the code generated tests. This approach would also allow use of existing techniques available for test filtering and shape reduction. Another approach is to code generate the trace directly such that when the generated code is executed it will expand and execute all the tests matching the pattern of the trace. However, this approach needs new ways to do test filtering and shape reduction, since it must be done during execution of the code generated trace.

8 Related work

This section describes existing work on code generation for VDM and approaches to constructing code generators for cases where expressing the source language in terms of the target language is non-trivial.

8.1 VDM code generation

VDM code generation was developed for VDMTools [10] in the nineties with support for both Java and C++, and has primarily been used to code generate prototype implementations rather than final production code. In the late nineties the Java code generator was extended to support code generation for the concurrency mechanisms of VDM++ [8]. Unfortunately, there is no scientific literature available to document the approach used to construct the VDMTools code generation feature.

VDMTools supports code generation for a larger subset of VDM compared to the current code generation platform described in this paper. However, VDMTools does, for example, not support code generation for a functional concept such as a lambda expression, which is non-trivial to express in earlier versions of Java where this is not supported. Code generation of lambda expressions (for earlier versions of Java) is achieved by the Java backend described in this paper by applying transformations to the IR.

8.2 The DMS Software Reengineering Toolkit

The DMS Software Reengineering Toolkit is a commercial set of tools for program analysis and transformations [3]. It contains tools for lexer and parser generation, functionality to pretty print an AST and specify program transformations, termed "transforms", using source rewrite rules. Rules are written in the DMS's Rule Specification Language and typically have the form *LHS* $\rightarrow$ *RHS* **if** *condition*. A rule is interpreted such that when a part of the program matches *LHS* it gets replaced by *RHS* if the condition is true. For example, a rule can be specified such that it replaces an assignment statement on the form `v = v+1` with the incrementation `v++`. To support the specification of rules, patterns use language syntax categories (e.g. both *LHS* and *RHS* must be of the statement syntax category) and it also allows use of metavariables to match with variables and expressions in the source language.

Transforms and source rewrite rules work at the concrete syntax level and therefore this approach differs from that used by the code generation platform described in section 4. Here transformations are applied to the IR at the abstract syntax level using visitors generated using the ASTCreator tool, where a case must be implemented to match a constructs in the IR. For example, a case can be implemented to match a set comprehension but there are also cases that allows categories of IR nodes such as statements, expressions and numeric binary expressions to be matched. Visitors have the potential to make changes all over the IR including adding new types and definitions as done by the Java backend during code generation of records as explained in subsection 5.2.

8.3 Code generating a logic language

Research has been done on translating logic languages such as Prolog [4] into imperative languages. In logic languages programs are expressed as logical formulas or horn clauses. Queries can be made about a program from which the interpreter will try to construct a proof. An example of a Prolog to Java translator is found in Prolog Café [2].

In order to be able to execute a code generated Prolog program, there must exist a runtime to support the generated code, and take over the role of the Prolog interpreter. This runtime is therefore responsible for trying to construct a proof that meets a given query. Since the Prolog interpreter is based on a highly efficient implementation and makes use of sophisticated algorithms to implement the traversal of the search tree, the implementation of such a runtime (or at least one that is efficient) is a complicated task.

The approach to use transformations to mitigate the complexity of code generating a logic language to a language such as Java can be expected to be of limited value compared to the case where code generation is more naturally done by expressing a source language using an IR. The reasons for this is that most of the challenges of code generating a logic language such as Prolog to Java involves the implementation of the runtime that constructs the proof.

9 Conclusion

The code generation platform presented in this paper supports construction of different backends and reduces the efforts needed to code generate a source language to multiple target languages. It achieves this by representing the generated code as an IR and subjects it to a series of semantic preserving transformations in order to obtain a tree structure that is easier for a backend to code generate. Since the IR is independent of any source and target language backends facing similar challenges during the code generation process, can use the same transformations to simplify their implementation.

To validate the architecture of the code generation platform a VDM++ to Java backend has been developed for the Overture tool, and work is currently being made on a VDM++ to C++ backend. Java and C++ are imperative languages and therefore both backends can use the same transformations in order to obtain an IR that is easier to code generate. To demonstrate this, it was shown how a set comprehension was transformed into a larger tree structure based on IR constructs of an imperative nature.

Applying transformations to an IR has proven useful for implementing the Java and C++ backends, but the the approach also supports code generation for other source languages. Therefore we hope that the work presented in this paper will be useful for others working with code generation.

Acknowledgements

This work is partially supported by the EU Framework 7 Integrated Project "Comprehensive Modelling for Advanced Systems of Systems" (COMPASS, Grant Agreement 287829). For more information see `http://www.compass-research.eu`.

References

1. The ASTCreator website (2014), `https://github.com/overturetool/astcreator`
2. Banbara, M., Tamura, N., Inoue, K.: Prolog Cafe : A Prolog to Java Translator System. Lecture Notes in Computer Science, vol. 4369, pp. 1–11. Springer (2005)
3. Baxter, I.D., Pidgeon, C., Mehlich, M.: DMS: Program Transformations for Practical Scalable Software Evolution. In: Proceedings of the 26th International Conference on Software Engineering. pp. 625–634. ICSE '04, IEEE Computer Society, Washington, DC, USA (2004)
4. Blackburn, P., Bos, J., Striegnitz, K.: Learn Prolog Now!, Texts in Computing, vol. 7. College Publications (2006)
5. Gosling, J., Joy, B., Steele, G., Brach, G., Buckley, A.: The Java Language Specification Java SE 8 Edition. Oracle America, Inc. (March 2014)
6. Jørgensen, P.W., Lausdahl, K., Larsen, P.G.: An Architectural Evolution of the Overture Tool. In: The Overture 2013 workshop (August 2013)
7. Larsen, P.G., Lausdahl, K., Battle, N.: Combinatorial Testing for VDM. In: Proceedings of the 2010 8th IEEE International Conference on Software Engineering and Formal Methods. pp. 278–285. SEFM '10, IEEE Computer Society, Washington, DC, USA (September 2010), `http://dx.doi.org/10.1109/SEFM.2010.32`, ISBN 978-0-7695-4153-2
8. Oppitz, O.: Concurrency Extensions for the VDM++ to Java Code Generator of the IFAD VDM++ Toolbox. Master's thesis, TU Graz, Austria (April 1999)
9. The SableCC website (2014), `http://www.sablecc.org/`
10. SCSK: VDMTools website. *http://www.vdmtools.jp/en/* (2007)
11. The Apache Velocity website (2014), `http://velocity.apache.org/`

11

Integrating Real System Components in Model-Based Development

This draft paper is planned to be submitted for publication (venue to be found).

[P21] Luís Diogo Couto and Peter W. V. Tran-Jørgensen. *Integrating Real System Components in Model-Based Development*. Draft paper planned to be submitted for publication (venue to be found).

Integrating Real System Components in Model-Based Development

Luis Diogo Couto
Department of Engineering, Aarhus University
Current affiliation: UTRC-I
CoutoLD@utrc.utc.com

Peter W. V. Tran-Jørgensen
Department of Engineering, Aarhus University
pvj@eng.au.dk

ABSTRACT

In Model-Based Development (MBD), system analysis is sometimes carried out through combined execution of model and real system components. Tool support for seamless realisation of these modelling setups is limited. To address this, we propose the delegate concept – when a model is realised via code generation, the connection mechanisms between the model and the external component are automatically replaced by the delegate that connects the generated code to the external component. The delegate has been developed as an extension of the code generator of the Overture tool for the Vienna Development Method (VDM). This includes integration with the Maven build system which unlocks automated realisation and validation of the system. The delegate has been employed in a VDM project on harvest field planning where, for realistic fields, it has increased system performance by over 3000%.

CCS Concepts

•Computing methodologies → Modeling methodologies; Simulation tools; •Software and its engineering → Model-driven software engineering; Software testing and debugging; *Formal methods;*

Keywords

System realisation; Build automation; Code generation; Test automation; Continuous Integration; Formal methods

1. INTRODUCTION

In Model-Based Development (MBD), we sometimes find ourselves in situations where development is carried out on both a model and real system components. When executing simulations of such setups, the model is executed together with the real system components. This is similar to the concept of software-in-the-loop modelling and simulation [33]. However, whereas software-in-the-loop is normally used to test software within a simulated context, our situation refers to active system development where certain parts of the system are represented by models and certain parts are already realised – we call the latter *external components* since they are either used as is, or developed using an implementation technology in parallel with the modelling activities.

When simulating models with external components, whenever the simulation reaches a point where the functionality of the external component is needed, the model invokes it through a connection mechanism called the bridge. Subsequently, the external component processes the result in order to continue with the simulation.

The problem we seek to address is a lack of support for automated realisation of models with external components. This lack of automation is problematic because it limits the ability to perform automated analysis and validation of the model and its associated realisation. When code generating the model to produce the system realisation, the connection mechanism between the model and the external component is lost and the generated code does not function correctly. Manual integration of the external component in the system realisation is possible, but this makes it challenging to maintain both the model and the system realisation.

In this paper, we present an approach to a fully automated integration of external components with the realised system via code generation [19]. This automation further extends to test generation and execution, thus enabling automated analysis and validation of model-based systems with external components. Our approach is based on the concept of a *delegate* that replaces the software-in-the-loop bridge and more directly connects the generated code to the external component. Whenever the model makes a call across the bridge, the generated code instead makes a call to the delegate, which then processes the call by forwarding it to the external component. This allows direct reuse and integration of the external component in the realised system. In this way, it becomes possible to carry out MBD to design and develop a system that integrates external components and ensure that the external components can still be used effectively in the system realisation.

The delegate mechanism can be fully automated in a way such that code generation proceeds without user intervention. In this manner, the practice of Continuous Integration (CI) can be applied not only to the model itself but to the generated code as well. This enables the generation and execution of tests as well as subsequent validation of the generated code for every modification made to the model. In addition, since code generation is performed every time a change is made to the model, subtle errors in the system that occur only when moving from model to code, can be detected sooner and more easily.

The delegate mechanism has been successfully employed in the MBD of a system for planning harvest operations in the field of precision agriculture. By using the delegate and a handwritten external component we were able to greatly increase the performance of the model and reduce simulation times by multiple orders of magnitude. In addition, the use of continuous integration, as enabled by

ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

the fully automated nature of the delegate, was essential in ensuring deployment of the various system prototypes.

The remainder of this paper is structured as follows: section 2 presents background information necessary for this paper; in section 3 we present our main contribution — the delegate mechanism and its usage; section 4 presents the harvest planning system case study; results for the application of the delegate to the case study are reported in section 5 and discussed in section 6, together with a more general discussion of the delegate; finally we describe related work in section 7 and conclude in section 8.

2. BACKGROUND

2.1 The Overture Tool

Our approach is based on the Overture tool and the formal modelling notations of the Vienna Development Method (VDM), one of the most mature Formal Methods [18, 10]. VDM focuses on the development and analysis of system models expressed in a formal language. The formal language enables a wide range of analysis of the model including testing and mathematical proof. A particularly noteworthy feature of VDM is its executable subset which makes VDM models highly amenable to simulation and automated execution to enable CI [31, chapter 7].

VDM contains three dialects: the most basic dialect (VDM-SL), standardised by ISO [23] supports modelling of the functionality of sequential systems. Additional dialects support object-oriented modelling and concurrency (VDM++) [11], and real-time computations and distributed systems (VDM-RT) [28, 38, 37]. All these dialects of VDM are supported by the Overture tool [22, 3, 32].

The Overture tool is structured around a common core, with various plug-ins providing different kinds of features including type checking, model execution and proof obligation generation. Two features worth noting for this paper are the VDMUnit library which provides support for developing and executing unit tests of VDM models, and the VDM-Java bridge [29] which enables VDM models to be combined with executable Java code.

The VDMUnit library enables unit testing of VDM models in a way that is similar to the JUnit library [20]. VDMUnit provides various classes and operations such as `TestCase` and `Assert` which allow the user to write unit tests for VDM models. Tests can be organised into suites which can be executed in a single execution session by the Overture tool.

The Java bridge enables two different kinds of connections between VDM and Java. First, it allows VDM models to interact with an external Java library. This corresponds to execution of the external component we have described in section 1. The external Java library feature allows VDM models to call functionality provided by Java jar files. This is achieved using the **`is not yet specified`** statement or expression. When these are encountered, the Overture interpreter will automatically attempt to call the respective functionality on the Java side, provided it is present and named according to convention. Although the Java bridge is normally used to call software functionality, it can also be used to route calls to a piece of hardware in order to perform hardware-in-the-loop simulation [15].

The other feature of the Java bridge is the ability to allow a Java program to remote control a VDM model. This allows a Java program to directly control and issue commands to the Overture interpreter by means of a `RemoteControl` Java interface. This feature is most commonly used for implementing user interfaces for VDM models. However, it is not relevant to the delegate technique and as such will not be discussed further. Further details about the Java bridge including implementation details can be found in the Overture manual [24, chapter 15].

2.2 Realisation of VDM Models

The delegate concept is implemented using Overture's Code Generation Platform (CGP), which provides a framework for building code generators for VDM [19, 36, 4]. We have used the features of the CGP to extend Overture's VDM-to-Java code generator with functionality for code generating models that combine VDM with executable code.

The CGP uses an Intermediate Representation (IR) to represent the code generated version of the VDM model. In the initial version of the IR each of the IR nodes correspond to a language construct in VDM such as an expression or a statement. The IR is subjected to a series of semantics-preserving transformations to replace IR constructs that are non-trivial to code generate with other IR constructs that are easier to code generate. As an example, the Java code generator uses transformations to rewrite VDM language features such as pattern matching and quantified expressions, which Java does not support. When the IR reaches its final form, i.e. the IR is fully transformed, the IR constructs are translated into concrete syntax or code in the target language, which finalises the code generation process.

Transformations provide a convenient way to access and manipulate the IR before code is emitted. In particular, changing a code generator's transformations allows a code generator to be extended with new functionality. The delegate principle and the JUnit4 [20] test generation features are both implemented as transformations that have been added to the Java code generator's transformation series.

The Java code generator uses a runtime library to implement some of the VDM operators and types used in the generated Java code. As an example, this library uses the `VDMSet`, `VDMSeq` and `VDMMap` classes to represent sets, sequences and maps, respectively. As we shall see in section 3, knowledge of the type mappings used by the Java code generator is needed in order to implement a delegate. The complete set of mappings are described in the Overture tool's user manual [25, section 11.5].

The Java code generator can be invoked as a Maven plugin, which allows VDM models to be code generated using the Maven build system [27]. Use of this Maven plugin enables VDM models to be included in a development environment and used in a MBD fashion. The delegate concept and JUnit4 test generation features are included in the release of the Maven plugin and available for public use. We use this plugin to realise our case study. In section 3 we demonstrate how the Maven plugin works and provide references for materials and examples that demonstrate how the plugin can be employed in a development environment.

Extending the Java code generator Maven plugin with the delegate concept and the JUnit4 test generation features allows the model tests to be used to automatically validate the code generated version of the model which enables the principle of CI [8] – executing tests every time changes to the model are committed to the source control repository. This is one of the advantages of integrating the Java code generator with the Maven build automation system.

3. APPROACH

The delegate concept enables the code generated version of the VDM model to be connected with the external component such that the Java code generator can produce the final version of the system. In this section we explain the structure of the delegate concept and demonstrate usage of this feature.

3.1 Overview of the Delegate Concept

The delegate concept is shown in fig. 1. In this figure the development is based on a `VDM Model`, which is validated using `VDM Tests`. In order to enable the `VDM Tests` to be translated to JUnit4 tests the `VDM Tests` must be written using the VDMUnit test framework, as described in section 2. Combined execution of the `VDM Model` and the `External Component` is enabled through the `Bridge`.

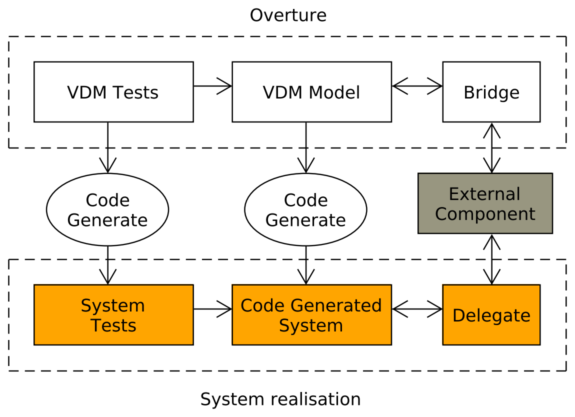


Figure 1: Overview of the delegate concept and test generation features.

From the `VDM Model` and the `VDM Tests` the code generator produces the `Code Generated System` and the `System Tests`, respectively. In order to enable the integration of the `Code Generated System` with the `External Component` the delegate feature processes the IR, produced by the Java code generator, and replaces the bodies of the `Bridge` operations and functions with direct invocations to the `Delegate`. This is explained in more detail in section 3.3.

3.2 Using the Java Bridge

As explained in section 2 a bridge operation (or function) uses the **`is not yet specified`** keyword to indicate that the operation is executed in an external environment. To demonstrate how the code generator treats these operations, consider the VDM bridge operation `op` in listing 1, which maps the input type `I` to the result type `R`.

```
class Bridge
operations
op : I ==> R
op (i) == is not yet specified;
end Bridge
```

Listing 1: An example of a VDM operation that uses the Java bridge.

The Java bridge uses Overture's `Value` runtime library to represent the input parameters and the result of the VDM operation. This runtime library enables the VDM interpreter to invoke the Java bridge method from a VDM environment. A conceptual implementation of the Java bridge operation in listing 1, is shown in listing 2. This Java method first converts the input parameter to a format that can be processed by the `ExternalComponent` (line 5). Then it uses the `ExternalComponent` to process the converted input (line 6), and finally the result is returned as a `Value` to the VDM interpreter (line 8).

```
1   import org.overture.interpreter.values.
       Value;
2
3   class Bridge {
4     public Value op(Value i) {
5       I conI = convert(i);
6       R res = ExternalComponent.op(conI);
7       Value convRes = convert(res);
8       return convRes;
9     }
10  }
```

Listing 2: Java bridge implementation of the VDM operation in listing 1.

3.3 Setting up the Delegate

Although the `Value` runtime library enables VDM and Java to be co-executed, it introduces a significant overhead due to all of its dependencies and required conversion operations (see listing 2). This overhead is not desired in the final version of the system, and therefore the Java bridge, or the `Value` runtime library, must be removed from the final version of the system. This is the responsibility of the delegate.

Each operation that uses the Java bridge has a corresponding delegate method defined in the delegate class. To use the delegate feature, the Java code generator plugin must be configured with each of the bridge-delegate pairs. An example showing how to use Maven to associate the VDM `Bridge` and `Delegate` class is given in listing 3. In this listing `Bridge` is the name of the VDM side of the bridge (a class or a module) and `Delegate` is the fully qualified name of the corresponding Java delegate class. The configuration further shows how to enable JUnit4 test generation by setting the `genJUnit4Tests` flag to `true`. This is the only configuration that is needed in order to use the delegate concept and the test generation features. When the configuration is completed everything is fully automated, which is key for CI.

Note that listing 3 omits the remaining Java code generator plugin configuration since the purpose of this paper is to show how to enable the delegate and test generation features. Instead we provide a complete example in [5] that documents and demonstrates how to use these features.

```
<configuration>
  ...
  <delegates>
    <property>
      <name>Bridge</name>
      <value>Delegate</value>
    </property>
  </delegates>
  <genJUnit4Tests>true</genJUnit4Tests>
</configuration>
```

Listing 3: Configuration of the `Bridge` and the `Delegate` for the Java code generator Maven plugin.

As shown in listing 4 this configuration file enables the code generated version of the `Bridge` operation to relay the call to the corresponding method in the `Delegate` class. In listing 4 `I` and `R` represent the code generated versions of the input type and the result type for the VDM operation in listing 1.

```
class Bridge {
  ...
  public R op(I i) {
    return Delegate.op(i);
  }
}
```

Listing 4: The code generated version of the VDM operation in listing 1.

The implementation of the `Delegate` must conform to the interface of the code generated `Bridge` class in order to enable their integration. That is, each delegate method must have the same signature as the corresponding method in code generated version of the `Bridge`. The signatures of the `Bridge` methods can be determined using the mapping between the VDM types and the code generated types described in section 2.2.

The user's implementation of a delegate method normally involves interaction with the `ExternalComponent` in some way. In our conceptual example, the `Delegate` simply relays the call to the `ExternalComponent` as shown in listing 5. However, in a realistic scenario it might do other things such as converting between code generated and user-defined types.

```
class Delegate {
  ...
  public R op(I i) {
    return ExternalComponent.op(i);
  }
}
```

Listing 5: The `Delegate` interacts with the `ExternalComponent`.

Combining the `Bridge` together with the `Delegate` and the `ExternalComponent` in this way results in the execution behaviour shown using the Unified Modeling Language (UML) sequence diagram in fig. 2. This diagram shows how the invocation to the code generated `Bridge` gets relayed to the `Delegate`, which is responsible for invoking the `ExternalComponent`. This approach has the advantage that it completely removes the `Bridge` boiler-plate code from the final version of the system, and only introduces a small overhead caused by the relayed invocation to the `ExternalComponent`.

4. CASE STUDY

The delegate concept was developed in part to support the work reported in this case study. This case study consists of a planning system for harvest operations, which has been developed using MBD. The system consists of 4975 lines of VDM specification and additionally 9361 lines of Java/Scala code, which have been used to develop and integrate the external component and user interface. This adds to a total of 14336 lines of specification and code (excluding the generated code). We provide only a brief high-level description of the system and focus mostly on the technical aspects that are relevant to motivating the need for the delegate concept and to explain its value.

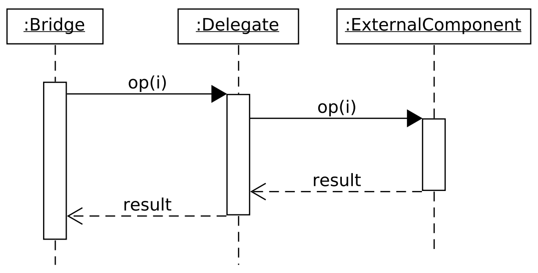


Figure 2: Delegate dispatching shown using a UML sequence diagram.

The planning system takes as input a field specification and a set of resources (harvesters, supporting grain carts and storage facility). The field is divided into parallel work rows and a surrounding area –the headlands – where vehicles typically manoeuvre (see fig. 3). In the system, the field is modelled by a graph-like structure consisting of the work rows, headlands segments and the connections between them. The resources are modelled by their relevant parameters such as capacity and speed. Figure 4 shows a high level overview of the system.

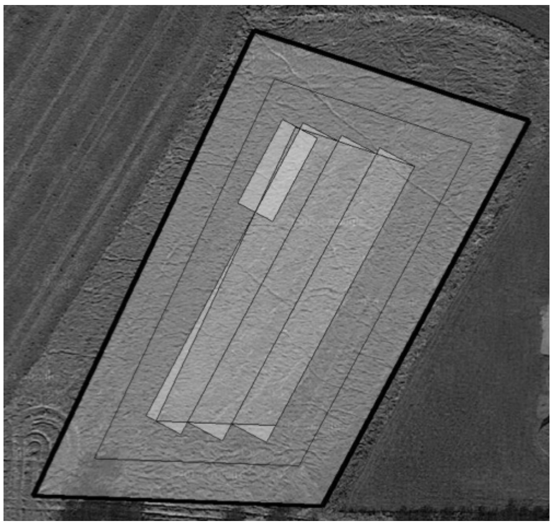


Figure 3: A field divided into 4 work rows and 2 headland laps.

Broadly speaking, the system works as follows: a field specification (already divided into rows) and a set of resources are fed into the system. Then, according to whichever optimisation algorithms [2, 16, 30] have been selected, instructions are calculated for the harvesters and the grain carts so that the harvesters harvest the field and the grain carts service the harvesters whenever they need to unload.

After the harvest plan has been calculated, the system executes a simulation of the harvest according to the plan, in order to assess how much time the harvest takes and to help the user visualise the harvest process. Figure 5 shows a sample harvest visualisation, with the `h1` and `h2` dots indicating harvesters, the `g1` and `g2` dots

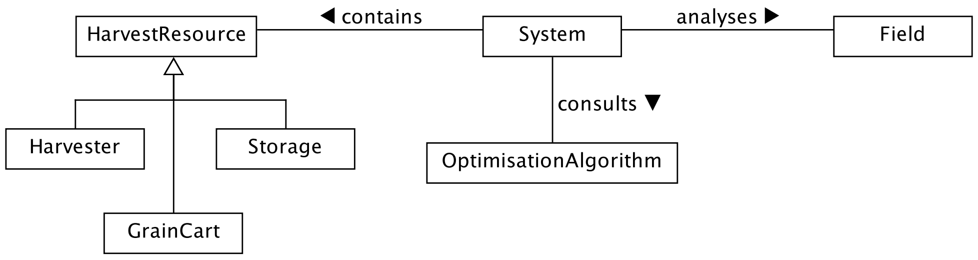


Figure 4: A domain model of the harvest planning system illustrated using a UML class diagram.

indicating grain carts and the dot outside the field indicating the storage facility. Harvested rows are painted red, unharvested rows are painted green and the connections between rows are painted yellow.

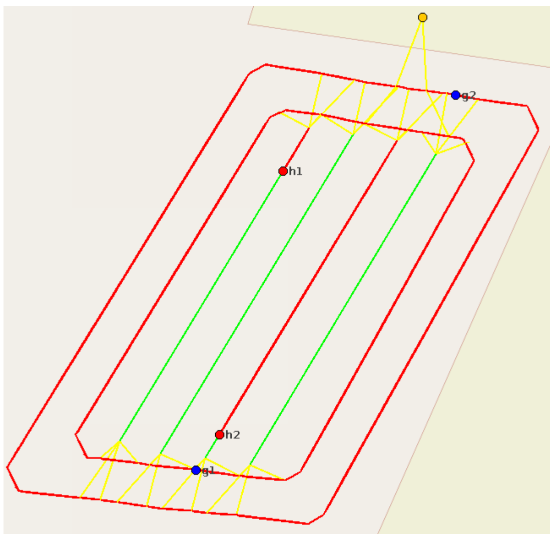


Figure 5: Harvest simulation visualisation.

One of the most important aspects of the model is how the field is represented. Although the harvest takes place in physical space, two-dimensional coordinate systems such as GPS are not particularly well-suited. This is because the harvesters and grain carts are significantly constrained in the way they move. They may only move along a row and they can only navigate from one row to another at the headlands (at either end of a workrow). Due to the restrictions on movement of the vehicles, a data structure such as a graph (where edges correspond to rows and headland segments and vertices connect them together) is more appropriate to represent the field.

As part of model execution, certain operations on graphs are executed frequently. For example, consider the following scenario: a harvester has become full and a grain cart needs to meet it at a designated unload point. It is necessary to compute the shortest path for the grain cart to reach the unload point. Afterwards, it is necessary to compute the shortest path from the unload point to the next destination of the grain cart and so forth.

Because of the frequency of these kinds of graph operations, it is important that the graph representing the field as well as the commonly used graph operations and the optimisation algorithms are efficient. Although it is possible to model all of these directly in VDM (indeed, this was attempted in the initial version of the model – version *m1*), it is challenging to do so with high efficiency as that is not a primary focus of the VDM languages and their associated tools.

As such, it was eventually decided to move away from a pure VDM approach and rely on Java and the JGraphT [17] library to handle field representation and provide efficient implementations of common graph operations such as shortest path, which is implemented using Dijkstra's algorithm [6].

The Java-based field representation is an external component that connects to the system model via the Overture Java bridge. This connection was made at the level of the `Field` class, as shown in fig. 6. The `Field` class wraps all calls across the bridge so that, from the perspective of the rest of the model, the Java implementation is invisible. An example call, for the shortest path operation, is shown in listing 6.

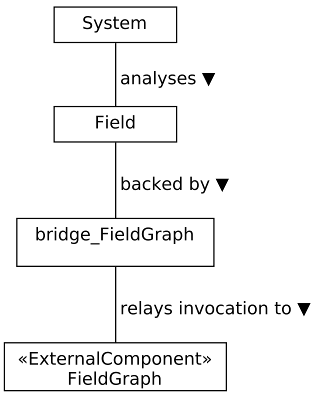


Figure 6: The external component of the harvest planning system.

```
class Field
operations
public shortestPath : int * int ==> seq
```

```
    of (int * Global`Direction)
shortestPath (pFromV,pToV) ==
  bridge_FieldGraph`shortestPath(pFromV,
      pToV);
...
end Field

class bridge_FieldGraph
operations
public static shortestPath : int * int
    ==> seq of (int * Global`Direction)
shortestPath (pFromV,pToV) ==
  is not yet specified
...
end bridge_FieldGraph
```

Listing 6: A Field class operation which is redirected to the Java bridge.

The calls from listing 6 are visualised using a UML sequence diagram in fig. 7. An operation for computing the shortest path is initially invoked on the field class. This call is then routed via the Java bridge to the Java-based field graph implementation.

By utilising the Java bridge, we were able to achieve an efficient implementation of the field representation and associated common graph operations (see section 5 for more). However, an important part of the project was the realisation of the planning system. This was done through the use of the code generator of Overture and the delegate mechanism.

In this project, the realised system consists of an application that enables users to select fields and resources, plan and simulate harvest operations and visualise the outcomes. The core of the system (the data model and operation planning and simulation features) is developed entirely via MBD. Once the code is generated, that code is used and integrated directly with two other components in order to produce the system realisation. One of the components is the graph-based field representation that has already been discussed; the other is a simple handwritten user interface module targeting the generated code.

By utilising the delegate, we were able to automatically generate the core of the planning system as well as its tests. We were also able to leverage the Maven build system and the Jenkins integration server to enable CI of not only the VDM model, but also the realised system itself by generating the code from the model and then integrating all components in order to build the system. Figure 8 shows an example chart from the CI server that tracks the duration of builds of the code generated system. As part of these builds, the code generation plugin is invoked so that code is generated from the latest version of the model. In addition, the code generated VDMUnit tests are executed to ensure that the core of the system is working correctly. It is possible to see in the chart, a build failure for build #56.

The application of CI to the project provided several other benefits. The fact that we built the realised system so frequently made it easy to detect problems with incorrectly specified or deployed dependencies (build #56 is an example of this). We expect this will be even more relevant when combining code generated from multiple sources, which we did not do. In addition, system deployment was easier since working builds were available at any time. Another advantage was the detection of subtle errors when developers lacked the discipline or time to run the full test suites every time. In addition, the project-wide notification of test failures helped motivate

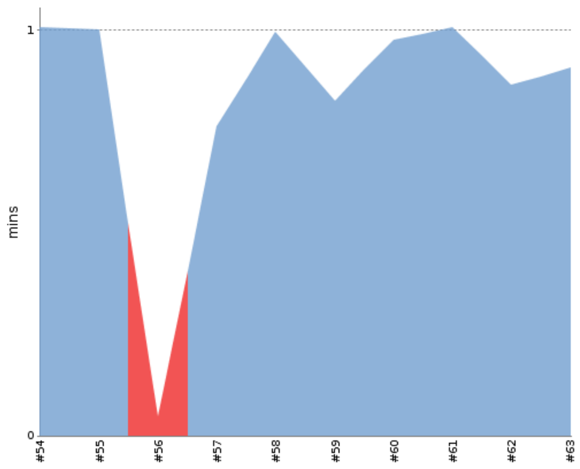


Figure 8: Sample chart from the CI server showing the build trends for the realisation of the harvest planning system.

developers to improve their discipline. Finally, we were able to detect subtle problems in the code generation plug-in when using certain VDM idioms. Our setup also enabled us to quickly react to these issues and deploy code generator fixes. Some of these are expected benefits of applying CI but the salient point is that the delegate concept enables the use of CI and, through that, the aforementioned advantages.

Significant time and effort were spent setting up the CI infrastructure for the project, but maintenance costs have been low since then. Although the CI server configuration can be complex, we believe it was time wisely spent since the benefits reaped are significant. Furthermore, the integration between the Overture tools and CI server is minimal, forcing us to carry out most of the CI work with handwritten scripts. There is clear room for improvement here and in the future it may be worthwhile to develop a VDM plug-in for CI. On the other hand, the fact that it was possible for us to set up the infrastructure without specialised tool support indicates that it may be possible for other formal notations to attempt to implement similar setups with their existing tool kits.

5. RESULTS

In this section, we present test results for the harvest planning case study described in section 4. These results seek to showcase the differences in performance between the two versions of the model — before and after the utilisation of the delegate — for both model simulations and executions of the system realisation.[1]

These tests were carried out by applying the model to plan harvests for fields of increasing size. Table 1 shows the results for model executions and table 2 shows the results for executing the system realisation.

For both tables, column one describes the field in terms of the number of rows and headland laps. In table 1 columns two and three show the simulation times for both versions of the model (T_{m1} for the original model version and T_{m2} for the version that

[1]Due to limitations with the original setup, we were unable to generate additional fields for testing and thus our results are somewhat disperse in terms of field size. As the purpose of this paper is not to provide a comprehensive set of results for field harvesting, but rather to demonstrate the performance improvements of the delegate, we feel the situation is acceptable.

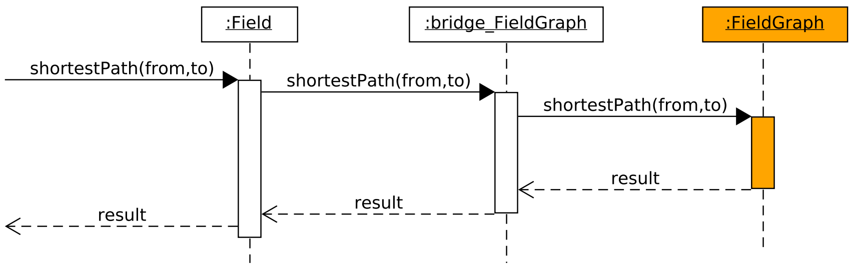


Figure 7: The shortest path operation and Java bridge call, shown using a UML sequence diagram.

Table 1: Model simulation results

Field	T_{m1} [s]	T_{m2} [s]	Gain
3 rows, 1 headland	18.621	4.976	374%
4 rows, 2 headlands	82.103	7.641	1075%
16 rows, 2 headlands	N/A (> 8H)	183.836	N/A

Table 2: System realisation execution results

Field	T_{r1} [s]	T_{r2} [s]	Gain
3 rows, 1 headland	0.183	0.048	381%
4 rows, 2 headlands	3.18	0.091	3495%
16 rows, 2 headlands	N/A (> 8H)	0.999	N/A

uses the delegate); finally, column four shows the gain in simulation speed, calculated as T_{m2}/T_{m1}. Similarly, table 2 shows the execution times for the system realisation, i.e. T_{r1} and T_{r2} denote the execution times for the first and second version of the system realisation. Likewise, the fourth column indicates the gain in simulation speed, calculated as T_{r2}/T_{r1}.

The first version of the system (both the model and the system realisation) was not able to process the largest field. In our tests, we allowed the simulation and system realisation to run overnight and in both cases it was under 10% completion after running for approximately 8 hours.

These tests were conducted on a Fujitsu LIFEBOOK U772 laptop with a 1.7GHz Intel Core i5 processor and 8Gb of memory running a Linux OS. The model simulations were executed using version 2.3.0 of Overture and the system realisation was executed on a Java 7 JVM.

6. DISCUSSION

The main problem with the first version of the harvest planning system is that the data structures and the algorithms used to process the fields do not scale in terms of field size. This is reflected by the time measurements T_{m1} and T_{r1} shown in table 1 and table 2. The scalability issues make it impossible to use the first version of the system to analyse fields of realistic sizes. The first version of the model uses non-standard data structures and algorithms to represent and analyse the field. In addition, the field analysis is specified in VDM, which generally performs significantly worse than executable code.

To circumvent these performance issues, the new version of the system represents the field as a graph and uses well-established libraries, based on standard algorithms, to analyse the field (see section 4). Through use of the Java bridge all graph analysis is performed in executable code, which improves the overall system performance. Otherwise, the overall structure and execution flow of the harvest planning system remains the same for both versions.

For the small-sized field (3 rows, 1 headland), the performance gain for the new system is 374% for the model and 381% for the system realisation. However, as the field size increases, the performance gains increase more drastically. For the medium-sized field (4 rows, 2 headlands) the gains increase to 1075% and 3495% for the model and the system realisation, respectively. The gain is more than three times higher for the system realisation, which is an indication that the system realisation scales better than the model in terms of resources needed to run a simulation.

This becomes more apparent when comparing the execution times for the new model, T_{m2}, to the execution times for the new system realisation, T_{r2}, for the medium-sized and large-sized fields. As shown in table 1 and table 2, T_{m2} increases more drastically (from 7.641 s to 183.836 s) than T_{r2} (from 0.091 s to 0.999 s) as the field becomes larger. In addition to the model being specified in an interpreted language, the VDM interpreter also performs internal consistency checks to ensure that the model has the desired properties. Naturally the number of consistency checks increases as the field becomes larger. These checks are not performed by the system realisation due to the extra performance overhead they would introduce.

For the large-sized field (16 rows, 2 headlands) the first version of the model and the system realisation do not complete the field within eight hours. The new version of the model completes the large-sized field in 183.836 seconds ($\approx$ 3 minutes), whereas the system realisation completes this field in 0.999 seconds. To summarise, what the first version of the system did not achieve in more than eight hours, the new version of the system achieves in less than a second.

7. RELATED WORK

Several technologies exist for integrating models with external components in the context of model analysis. All of these technologies would benefit from a delegate mechanism in order to automate the process of moving towards a full system realisation. In [12] Fröhlich et al. propose a method for combining VDM specifications with C++ code. A more generic approach is employed in VDMTools, which supports integration of external code using

CORBA [35]. Their approach therefore enables integration with any CORBA-supported programming language. In [13] Gallasch et al. present Comms/CPN – a Standard ML framework that provides an infrastructure for DESIGN/CPN to establish communication between Coloured Petri Net (CPN) models and external processes. A similar technology is available in CPN Tools [39] – the successor of DESIGN/CPN. Ladenberger et al. present B-Motion Studio, which uses a graphical editor to create visualisations for Event-B models [21]. Their approach uses Event-B expressions to connect the model with the visualisation.

Technologies that enable realisation and validation of models that use real system components are more limited. We are not aware of any technologies that support direct integration with a build automation system, which is an important part of our work. However, some noteworthy attempts have been made to integrate MBD into existing development environments.

TargetLink [7] is an automatic C code generator for Simulink/Stateflow [26], which uses Custom Code blocks to enable integration with code such as an external component. This is similar to what can be achieved using Overture's Java bridge. Automation and process integration is enabled using the TargetLink Application Programming Interface (API), which provides access to TargetLink properties and settings. The individual stages of the build process can be intercepted using "hook functions" and used to, for example, integrate TargetLink into an existing development environment. Our work takes a different approach to integrating code generation into a development environment. First, the Java code generator is implemented using the CGP, which makes it possible to intercept and control the different phases of the code generation process (see section 2.2). Secondly, the Java code generator is exposed as a Maven plugin and therefore fully integrated with the Maven build automation system. Finally, the Java code generator plugin, including our work, is available as open-source software, which enables a high degree of customisation.

Another technology related to TargetLink is MATLAB Coder [26], which supports integration with legacy code as well as C and C++ code generation from MATLAB models. Embedded Coder can be used together with MATLAB Coder to optimise the generated code for embedded system development. To make system realisation as seamless as possible, Embedded Coder supports integration with other popular Integrated Development Environments (IDEs) such as Texas Instruments' Code Composer Studio [34] and VisualDSP++ [1]. Our work is advantageous for IDEs that provide integration support for Maven. Examples of such IDEs include Eclipse [9] and IntelliJ [14].

8. CONCLUSION

We have presented a tool-supported methodology for automated realisation and validation of systems that consist of both model and real system components. Automated realisation is achieved through the delegate concept, which supports seamless integration of an external component in both a modelling and a system realisation context. Automated validation is achieved through reuse of model tests, which supports the development practices of CI, where every change made to the model is automatically validated against the model tests. We have implemented the delegate concept and test generation features as an open-source extension to Overture's VDM-to-Java code generator Maven plugin. This approach enables MBD using the Maven build system.

The delegate concept has been employed in a large VDM project on developing a planning system for harvest operations. In the early versions of this model all the harvest planning algorithms were specified solely using VDM. This gave rise to performance issues for harvest planning for realistic-sized fields, which necessitated a major revisioning of the model. The new version of the model takes advantage of the Java bridge and uses a graph library to compute the harvester and grain cart routes. Use of the delegate concept and the test generation features makes realisation and validation of the system a seamless task.

As shown in section 6, increasing the field size significantly degrades the performance of the old version of the system. For the large-sized field (16 rows, 2 headlands) the old version of the system was not capable of completing the harvest simulation within eight hours. The new version of the system, on the other hand, completes the harvest simulation for the same field in less than a second. This is achieved by encapsulating computationally intensive functionality in an external component, and using the delegate concept and test generation features to realise and validate the system.

Several modelling and simulation technologies (such as those described in section 7) provide support for integrating models with external components. However, the tool support for realising these models, while reusing the external component, is much more limited. We therefore believe that the delegate concept, presented in this paper, can be valuable to any modelling and simulation technology which seeks to automate the realisation and validation for these modelling setups.

9. ACKNOWLEDGMENTS

We would like to thank Rene Nilsson and Gareth Edwards for helping us produce the harvest simulation visualisation and field partition figures (fig. 3 and fig. 5). In addition, we would like to thank Nick Battle and Peter Gorm Larsen for valuable feedback on earlier versions of this paper.

The work presented here was partially carried out in the context of the Danish High Technology Foundation research project *Off-line and on-line logistics planning of harvesting processes*.

10. REFERENCES

[1] Analog Devices, VisualDSP++ website. http://www.analog.com, 2016.

[2] D. Bochtis and S. Vougioukas. Minimising the non-working distance travelled by machines operating in a headland field pattern. *Biosystems engineering*, 101(1):1–12, 2008.

[3] L. D. Couto, P. G. Larsen, M. Hasanagic, G. Kanakis, K. Lausdahl, and P. W. V. Tran-Jørgensen. Towards Enabling Overture as a Platform for Formal Notation IDEs. In *2nd Workshop on Formal-IDE (F-IDE)*, Oslo, Norway, June 2015.

[4] L. D. Couto and P. W. V. Tran-Jørgensen. Extending the Overture code generator towards Isabelle syntax. In *Proceedings of the 13th Overture Workshop*, pages 48–59, National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-Ku, Tokyo, Japan, June 2015. Center for Global Research in Advanced Software Science and Engineering. GRACE-TR-2015-06.

[5] Delegate Tutorial Github project. https://github.com/ldcouto/delegate-tutorial, 2016.

[6] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[7] dSPACE GmbH TargetLink Product Management. *TargetLink 4.1 Product Information*, February 2016.

[8] P. M. Duvall, S. Matyas, and A. Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[9] The Eclipse Foundation, Eclipse IDE website. http://www.eclipse.org, 2016.

[10] J. Fitzgerald and P. G. 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.

[11] J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef. *Validated Designs for Object–oriented Systems*. Springer, New York, 2005.

[12] B. Fröhlich and P. G. Larsen. Combining VDM-SL Specifications with C++ Code. In M.-C. Gaudel and J. Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, pages 179–194. Springer-Verlag, March 1996.

[13] G. Gallasch and L. 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.

[14] JetBrains, IntelliJ IDE website. https://www.jetbrains.com/idea, 2016.

[15] J. A. E. Isasa, P. W. Jørgensen, and P. G. Larsen. Hardware In the Loop for VDM-Real Time Modelling of Embedded Systems. In *MODELSWARD 2014, Second International Conference on Model-Driven Engineering and Software Development*, January 2014.

[16] M. F. Jensen, D. Bochtis, and C. G. Sørensen. Coverage planning for capacitated field operations, part ii: Optimisation. *Biosystems Engineering*, 139:149–164, 2015.

[17] JGraphT. https://www.jgrapht.org/, 2016.

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

[19] P. W. V. Jørgensen, L. D. Couto, and M. Larsen. A Code Generation Platform for VDM. In *The Overture 2014 workshop*, June 2014.

[20] The JUnit website. http://www.junit.org, 2016.

[21] L. Ladenberger, J. Bendisposto, and M. 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.

[22] P. G. Larsen, N. Battle, M. Ferreira, J. Fitzgerald, K. Lausdahl, and M. Verhoef. The Overture Initiative – Integrating Tools for VDM. *SIGSOFT Softw. Eng. Notes*, 35(1):1–6, January 2010.

[23] 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.

[24] P. G. Larsen, K. Lausdahl, and N. Battle. The VDM-10 Language Manual. Technical Report TR-2010-06, The Overture Open Source Initiative, April 2010.

[25] P. G. Larsen, K. Lausdahl, P. W. V. Tran-Jørgensen, A. Ribeiro, S. Wolff, and N. Battle. Overture VDM-10 Tool Support: User Guide. Technical Report TR-2010-02, The Overture Initiative, www.overturetool.org, May 2010.

[26] MathWorks, MATLAB/Simulink/MATLAB Coder/Embedded Coder website. http://www.mathworks.com, 2016.

[27] The Apache Maven Project website. https://maven.apache.org, 2016.

[28] P. Mukherjee, F. Bousquet, J. Delabre, S. Paynter, and P. G. Larsen. Exploring Timing Properties Using VDM++ on an Industrial Application. In J. Bicarregui and J. Fitzgerald, editors, *Proceedings of the Second VDM Workshop*, September 2000. Available at www.vdmportal.org.

[29] C. B. Nielsen, K. Lausdahl, and P. G. Larsen. Combining VDM with Executable Code. In J. Derrick, J. Fitzgerald, S. Gnesi, S. Khurshid, M. Leuschel, S. Reeves, and E. 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.

[30] C. Nilsson. Heuristics for the traveling salesman problem. Technical report, Tech. Report, Linköping University, Sweden, 2003.

[31] R. Osherove. *The Art of Unit Testing*. Manning, 2 edition, 2013.

[32] The Overture tool website. http://www.overturetool.org, 2016.

[33] P. F. Riley and G. F. Riley. Next Generation Modeling III - Agents: Spades — a Distributed Agent Simulation Environment with Software-in-the-loop Execution. In *Proceedings of the 35th Conference on Winter Simulation: Driving Innovation*, WSC '03, pages 817–825. Winter Simulation Conference, 2003.

[34] Texas Instruments, Code Composer website. http://ti.com, 2016.

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

[36] P. W. V. Tran-Jørgensen, P. G. Larsen, and G. T. Leavens. Automated translation of VDM to JML annotated Java. January 2016. Submitted to the International Journal on Software Tools for Technology Transfer (STTT).

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

[38] M. Verhoef, P. G. Larsen, and J. Hooman. Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM 2006: Formal Methods*, Lecture Notes in Computer Science 4085, pages 147–162. Springer-Verlag, 2006.

[39] M. Westergaard and L. Kristensen. The Access/CPN Framework: A Tool for Interacting with the CPN Tools Simulator. In G. Franceschinis and K. Wolf, editors, *Proceedings of the 30th International Conference on Applications and Theory of Petri Nets*, pages 313–322. Springer Berlin / Heidelberg, 2009.

12

Extending the Overture code generator towards Isabelle syntax

This paper has been accepted as a peer-reviewed workshop paper.

[P20] Luís Diogo Couto and Peter W. V. Tran-Jørgensen. *Extending the Overture code generator towards Isabelle syntax*. 13th Overture Workshop, June, 2015.

Extending the Overture code generator towards Isabelle syntax

Luís Diogo Couto and Peter W. V. Tran-Jørgensen

Department of Engineering, Aarhus University, Denmark
{ldc,pvj}@eng.au.dk

Abstract. Overture has a Code Generation Platform (CGP), designed with extensibility in mind but this extensibility has never been thoroughly tested before. In this paper, we explore the extensibility of the Overture CGP by developing code generation support targeting an Isabelle embedding of VDM. We compare our solution to an existing hand-coded VDM to Isabelle translation based on direct traversals of the VDM AST and show that using the CGP led to a decrease in code volume of 86%. We also report various extensibility improvements that have been incorporated into the CGP as part of our work.

Keywords: VDM, code generation, Isabelle, extensibility

1 Introduction

The Overture tool[1] for VDM [6] has a Code Generation Platform (CGP) that was originally developed targeting the Java language but was designed with extensibility in mind. The intent of the CGP is to make it easy to contribute new Code Generation (CG) support for new languages to Overture [12]. Currently, the CGP supports the original Java code generation as well as an experimental generation of C++. The extensibility features of the CGP have never been thoroughly tested since C++ generation is similar to Java generation.

In this paper, we further explore the extensibility of the CGP by developing experimental support for generation of Isabelle syntax, which differs from Java more significantly than C++ does. The reason for this is that Java and C++ are both imperative OO languages and Isabelle is not. The process for developing this translation is also generalised into a standard methodology for developing CGP extensions.

There are two reasons for choosing Isabelle: there is already a usable existing embedding of VDM in Isabelle that we can reuse and a corresponding translation that runs on Overture models [3]. This translation was handwritten and as such will provide a good basis of comparison to see if it is really worthwhile to use the CGP. The comparison shows that using the CGP leads to a code volume reduction of 86%.

The remainder of this paper is structured as follows: the code generation platform as well as the existing Isabelle embedding and translation are described in section 2. The steps taken by the developer to construct the new CG extension are described in section 3. Relevant details of the Isabelle translation are discussed in section 4. The results

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

of the work in terms of the new Isabelle translation and extensibility improvements to the CGP are reported in section 5 and evaluated in section 6. Finally, we discuss future work in section 7 and conclude in section 8.

2 Background

2.1 Isabelle Embedding

This subsection presents the target language of the translation: an Isabelle embedding of VDM. Isabelle [13] is a framework for implementing logical formalisms and the VDM embedding being targeted is one such formalism. It was originally developed for the COMPASS Modelling Language (CML) [15] in the COMPASS project [7] and is built on an Isabelle mechanisation [8] of the UTP semantics used for CML [10].

CML is a combination of VDM and CSP [9]. In particular, the types, values, expressions and functions of CML are lifted from VDM. State is similar although it is handled somewhat differently – state in CML is composed of multiple independent variables much like VDM++ rather than a single record structure. Additionally, CML does not support the **`let be st`** construct due to its non-deterministic nature. The remaining differences between CML and VDM are related to the reactive and Object Oriented (OO) features of the language. Neither are relevant for this translation.

The Isabelle embedding of CML/VDM is a deep embedding, which means that it gives an explicit semantics to each construct of CML/VDM in Isabelle. In other words, rather than translating from VDM to another formalism, each construct in VDM is defined in the embedding and then given a semantics using formalisms available in Isabelle – specifically, higher-order logic.

Furthermore, the parsing capabilities of Isabelle give significant flexibility when defining the syntax of the VDM constructs in the embedding. The end result is that the embedding has its own syntax which is quite similar to that of the VDM language itself. The primary differences lie in separator characters such as `"` to distinguish between Isabelle and VDM syntax, ^ to identify VDM variables and `@` to identify VDM types.

In addition to the syntactical similarities there is also a near one-to-one correspondence between constructs in the source and target languages which facilitates the translation process. However, while CML has OO features the embedding does not support OO so it is suitable for representing VDM-SL models only.

Finally, we briefly describe the manually written existing translation, based on the visitor framework of the Abstract Syntax Tree (AST). The translation visitors traverse the AST and produce an intermediate data structure used to store relevant translation information for each node including its syntax and dependencies. Afterwards, the data structure is used to generate the Isabelle syntax, either with direct conversion to strings or with auxiliary methods and classes for the processing of more complex nodes. Further details about the existing Isabelle translation as well as the embedding are available in [7].

2.2 Code Generation Platform

The reason for using the CGP, and what makes it a viable solution for developing code generators, is found in the way the CGP represents and works with the generated code.

From the VDM AST the CGP constructs an Intermediate Representation (IR) of the generated code, which forms a tree structure that is independent of any particular target language.

Initially, each node in the IR has a one-to-one correspondence to a node in the VDM AST. Subsequently, the IR is subjected to a series of *transformations* in order to change the tree structure into a new form that is easier for a particular code generator to produce code from. More specifically, each transformation represents a rewriting of the IR with the purpose of changing the IR into a form where each node in the resulting tree structure maps easily into the target language. One advantage of this approach is that transformations operate directly on the IR, and therefore they can be shared among code generators. As an example, the Java and C++ code generators use many of the same transformations to eliminate functional-styled constructs in the IR such as quantified expressions and collection comprehensions.

The IR is generated from an AST specification file using the *AstCreator* tool [1]. In addition to the IR nodes, the *AstCreator* also generates mechanisms to walk the tree using visitors [5] as well as functionality to change the tree structure by allowing parts of it to be replaced. Transformations are themselves implemented as extensions to the visitors generated by the *AstCreator*. What characterises a transformation is that in addition to traversing the tree structure, it also manipulates it.

After the IR has been fully transformed, it is handed over to a language-specific backend generator in order to finalise the code generation process. The CGP provides a framework for syntax generation that serves to facilitate production of code in the target language. This framework is based on the Apache Velocity template engine and used for mapping each node in the IR into concrete syntax [14]. This is handled by the *template manager*, which associates each type of IR node to a template file, that describes the code to be produced.

Code generators extending the CGP may need extra nodes in addition to those already defined by the platform. Therefore, the CGP allows new nodes to be added via the *AstCreator* extension mechanism [4]. This mechanism allows the *AstCreator* to produce nodes and visitors that allow construction and traversal of hybrid trees, .i.e. tree structures composed of both IR nodes defined within the CGP and new nodes contributed via an AST specification extension file. In addition to adding new nodes, the CGP also allows existing IR nodes to be extended to include new fields. Finally, the template manager can be redefined to support syntax generation of new nodes added by the user.

3 Methodology

Based on the description of the CGP in subsection 2.2 we now outline the steps used to develop the Isabelle syntax generator. These steps constitute a general methodology for development of code generation support in Overture using the CGP. Others who want to use the CGP to develop code generation support for another target language may benefit from following these steps.

We start out by listing the steps to be carried out by the developer and afterwards we elaborate on each of them.

1. Set up the CGP extension
2. Add new nodes
3. Transform the IR
4. Generate syntax
5. Validate the translation

The first step in the process is only necessary once. The remaining steps are done in an iterative manner. The approach is to start with a very small VDM example and go through the steps until the example is completely translated. Afterwards, the example should be expanded as little as possible and the steps repeated. This is done iteratively until the new CGP extension is complete.

Step 1 - Set up the CGP extension: Broadly speaking, the setting up of the CGP extension consists of subclassing the base code generator class – `CodeGenBase` – that is the common extension point of the CGP. The base code generator is responsible for driving the code generation and providing access to the IR and various settings. It is also responsible for storing data used and generated throughout the code generation process.

Next, it is necessary to construct a new template manager for the extension. This can be done by subclassing the base template manager. This will provide access to the basic CGP template structure which manages an initial collection of template locations. If additional template locations are necessary, the template manager can be used to configure them.

Finally, it is worth setting up a basic test infrastructure to drive the development process. This test infrastructure is responsible for processing a VDM source, passing the respective AST to the code generator and validating the translation outcome.

Step 2 - Add new nodes: If the target language construct being translated is sufficiently different from those of the base IR, then it is likely that a code generator needs extra nodes. If necessary, these can be provided by extending the IR as described in subsection 2.2. Once the extension is defined, the *AstCreator* tool must be invoked in order to generate the extension nodes.

Step 3 - Transform the IR: Constructs that are not supported by the code generator need to be transformed away, using either base IR nodes or extended nodes generated in the previous step. This is done by implementing one or more necessary transformations. It is recommended that transformations be as small as possible so that each transformation only changes the IR in terms of one concept such as removing comprehensions or reordering definitions.

Step 4 - Generate syntax: Once the IR is in a form suitable for code generation, syntax can be generated using the syntax generation framework of the CGP. This is done by creating the Apache Velocity template files for each of the nodes that is to be translated and updating the template manager accordingly.

Step 5 - Validate the translation: Validation of the translation should be done by means of the test infrastructure by comparing the translation output to a reference. Alternatively, executable translated code may also be compiled and executed to ensure it produces the right result. This test should then be stored to use as regression in the continued development of the CGP extension.

4 Translations and Transformations

The Isabelle embedding we are targeting is very similar to VDM in the sense that most constructs in VDM are present in the embedding. As such, the initial version of the IR is already close to what is needed for generation – most nodes in the IR already map directly to a construct in the target language. Therefore, there is a relatively small number of operations that need to be performed over the tree.

The first set of operations is also the simplest and most common: direct syntax translations. These translations can be applied directly to the initial IR nodes that already map directly to a construct in the embedding. A few of them are shown in Table 1. These translations take advantage of the fact that the Isabelle embedding of VDM defines its own syntax which is quite close to that of VDM. In general, the syntax is the same as that of source VDM, except for the following:

– all constructs are delimited by `"` to identify them as user-defined syntax in Isabelle
– variables names are delimited by `^` to mark them as model variables
– types are prefixed by `@` to mark them as model types
– string literals are delimited by `''`

VDM	**Isabelle embedding**
`x`	`"^x^"`
`int`	`"@int"`
`f(1)`	`"f(1)"`
`"foo"`	`"''foo''"`
`if b then s1 else s2`	`"if ^b^ then ^s1^ else ^s2^"`

Table 1: VDM constructs and their Isabelle embedding counterparts.

To achieve these translations, all that is necessary is to specify the target syntax in the Velocity templates and the CGP handles everything else. Most templates are simple since most translations only need to add minor pieces of Isabelle syntax. A few translations require some extra logic – for example, sequences of type `char` are handled differently from all other sequences – and this is achieved through a handful of auxiliary static methods callable from within the template engine.

The second set of operations consists of tree transformations, of which the first is reordering of definitions. Isabelle does not allow forward referencing in its definitions so any dependency of a definition must be processed before the definition itself. When generating syntax, the CGP processes definitions in the IR in the order in which they appear so it is necessary to reorder the IR nodes according to their dependency relation. For example, consider the VDM functions shown in Listing 1.1. The initial IR generated for this example would have to be re-ordered as shown in Figure 1.

```
f : int -> int
f (x) == if x = 0 then 0 else g(x);

g : int -> int
g (x) == x/x;
```

Listing 1.1: A simple forward dependency example.

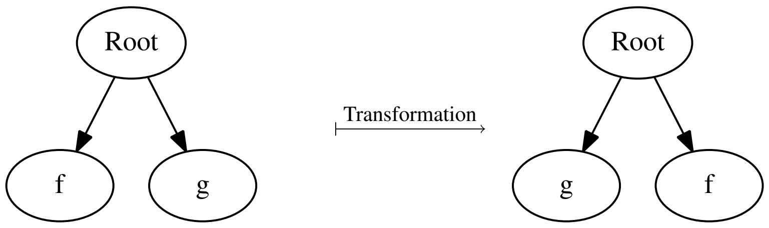


Fig. 1: Dependency sorting transformation.

Dependency sorting is implemented as a CGP transformation that takes an IR module node (the top level element of the IR), constructs a dependency graph of its definitions and then applies a topological sort algorithm [2].

The final operation over the IR is also related to dependency handling, specifically the dependencies between mutually recursive functions. Isabelle can cope with mutually recursive functions but these must be identified as such and grouped together for processing.[2] In order to provide grouping of mutually recursive functions, we construct another transformation that constructs a dependency graph for the function definitions and afterwards applies an algorithm for computing strongly connected components [11]. Thus, the VDM functions in Listing 1.2 would be transformed as shown in Figure 2.

[2] Although Isabelle supports them, the VDM embedding cannot currently cope with mutually recursive functions. However, we have implemented the transformation nonetheless as it was a good way to test the extensibility of the CGP.

```
odd: nat -> bool
odd (x) == if x = 0
    then false
    else even(x-1);

even : nat -> bool
even (x) == if x = 0
    then true
    else odd(x-1);
```

Listing 1.2: A simple example of mutual recursion.

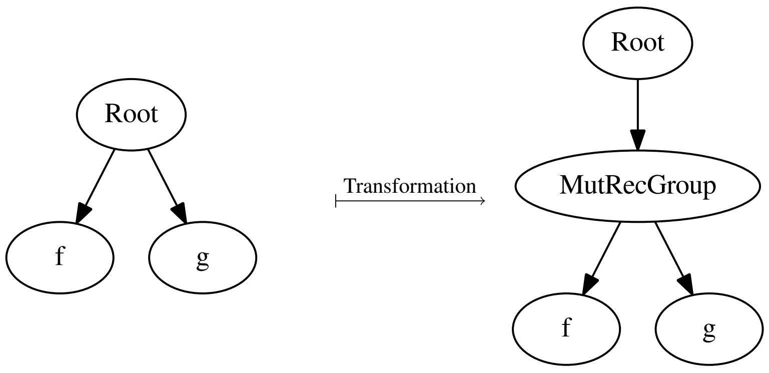


Fig. 2: Mutual recursion grouping transformation.

It is worth noting that the base IR module node does not support mutual recursion groups. As such, we extended the IR to add a new field for it. The mutual recursion transformation takes a base module node as its input and produces an extended module node.

5 Results

5.1 New Isabelle Generation

This section presents the translation from VDM to Isabelle. The translation is demonstrated by means of a complete example, shown in Listing 1.3. Much of the translation is straightforward syntax conversion, however, the example demonstrates the two main issues discussed in section 4: reordering definitions due to dependencies and grouping mutually recursive functions.

Functions `g()` and `f()` shown in lines 3-7 of the VDM model are translated to functions `f()` and `g()` in the Isabelle embedding shown in lines 5-13. Note that the two functions have changed to that `f()` comes before `g()` in the Isabelle source. This is because `f()` is a dependency of `g()` and so must be processed first.

Functions `odd()` and `even()` shown in lines 9-17 of the VDM model are also translated to functions in the Isabelle embedding, shown in lines 15-31. However, the functions in the embedding are delimited by the **`begin_mutrec`** and **`end_mutrec`** keywords which identify them as a block of mutually recursive functions. In Isabelle, such functions must be delimited as they are processed together.

5.2 Code Generation Extensibility Improvements

In addition to constructing the new extension, a series of improvements to the extensibility of the CGP were also carried out. The first set of extensibility improvements had minor impact on the CGP and was related to changing the visibility of various classes and class members. Prior to this work, we were uncertain of which parts of the CGP needed to be exposed to extensions. While it would have been possible to simply expose everything, that would make the CGP too complex to use. By carrying out this work we were able to discover which features to expose and were able to safely keep the rest encapsulated inside the CGP.

As an example of the above, the template manager has a field that defines the folder structure used so store template files. This field was not visible to extensions and that forced an extension to follow the same structure as the base CG with no ability to redefine it. By making the field visible to subclasses, it became possible for each extension to define its own template folder structure.

The second change to increase extensibility had greater impact on the design of the CGP and was related to transformation application. Originally, the CGP was only capable of transforming the internal part of a node. In other words, the root node of the tree could not be changed. This was insufficient for our extension because it was necessary to have a different class at the root of the tree. To address this, the CGP was modified to support transformations that convert between different node types at the root of the tree and thus it became possible to perform transformations between any two arbitrary trees. This new kind of transformation was named *total transformation* and the existing ones were preserved as *partial transformations*. One advantage of the *partial transformation* is that it can rely on the root node of the tree to remain the same and know what kind of node it is. This reduces the amount of conversions that are required to perform the transformation. The *total transformation* is more powerful but will always take as input and produce as output a generic tree node. The CGP was enriched with functionality to help cope with this by converting between generic and specific root nodes via the adapter pattern [5].

6 Evaluation

To assess the effectiveness of using the CGP for Isabelle translation, a simple comparison of volume – measured in Lines of Code (LoC) – was performed between the two

```
functions

g : nat -> nat
g (x) == f(x);

f : nat -> nat
f (x) == x;

odd: nat -> bool
odd (x) == if x = 0
    then false
    else even(x-1);

even : nat -> bool
even (x) == if x = 0
    then true
    else odd(x-1);
```

(a) VDM model.

```
theory A
  imports utp_cml
begin

cmlefun f
  inp x :: "@nat"
  out "@nat"
  is "^x^"

cmlefun g
  inp x :: "@nat"
  out "@nat"
  is "f(^x^)"

begin_mutrec

cmlefun odd
  inp x :: "@nat"
  out "@bool"
  is "if (^x^ = 0)
    then false
    else even((^x^ - 1))"

cmlefun even
  inp x :: "@nat"
  out "@bool"
  is "if (^x^ = 0)
    then true
    else odd((^x^ - 1))"

end_mutrec

end
```

(b) Isabelle translation.

Listing 1.3: VDM model and respective Isabelle translation.

versions. LoC is an imperfect measure of volume and does not particularly capture effort or productivity. However, it can be effectively and accurately measured and does provide a reasonable measure of the size of an implementation, which is sufficient for our comparison.

Table 2 presents a summary of results. In this table, *Manual* refers to the original visitor-based translation and *CGP* refers to the translation we have implemented. The comparison does not consider components from the original translation that are responsible for processing CML-exclusive elements that have no counterpart in VDM. To facilitate comparison, we have broadly grouped the sources of both versions into three groupings:

data Refers to classes implementing the intermediary data representation between source and target syntax
process Refers to classes that are used to help process or analyse the intermediary representation
syntax Refers to classes that provide or define the target syntax for final translation printing

	Manual	CGP	ΔLoC_{abs}	ΔLoC_{rel}
data	981	27	954	97.25%
process	2427	538	1889	77.83%
syntax	1395	86	1309	93.84%
Total	4803	651	4152	86.45%

Table 2: Volume comparison between translation implementations measured in LoC.

Looking at the data in Table 2, it is clear that utilising the CGP allows for an implementation with much less volume – a reduction of 86%. There are gains in every grouping but the largest ones are in the internal representation – 97%. This is because the *Manual* version utilises a handwritten data structure, whereas the *CGP* version reuses the IR and the only code necessary is that for defining the necessary data extensions. Likewise, most of the machinery for processing both the source language and the IR is reused from the CGP. Particularly, the construction of the IR from the source AST is handled entirely by the CGP. The *syntax* grouping is also much smaller in the *CGP* version – a reduction of 93% –since it uses the template engine in the CGP which allows for significantly more concise expression of syntax.

7 Future Work

In the future, there are two main avenues for improving this work: the translation itself and the extensibility of the CGP. Beginning with the translation, the most immediate improvement is to expand the coverage of VDM constructs. This is to some extent tied to the support of the embedding but there is a significant number of supported constructs that are not translated. For most of these it is only a matter of adding the relevant templates, although there is also the matter of making the dependency calculator more generic, which should not present a problem.

On the topic of the embedding, it would be worthwhile to switch to a pure VDM embedding. While the similarities between CML and VDM make the current embedding suitable for an initial translation, it would be beneficial to migrate to a dedicated embedding for VDM that could be maintained and evolved separately as necessary. Furthermore, the current embedding contains multiple definitions supporting the reactive aspects of CML that are unnecessary from a VDM perspective. Finally, a dedicated VDM embedding would allow for syntax that is even closer to that of VDM. Work is already underway on adapting the CML embedding into a pure VDM one.

Returning to the translation, there is a potentially problematic issue in that it is only possible to generate syntax for all definitions of the same kind together in one pass. This is a problem when needing to print definitions of various kinds according to the order of dependencies. The issue is related to the IR being structured as lists of definitions of the same kind. It would need to be altered to support generic definition lists – for example, type and function definitions would be stored in the same list.

In terms of translation, it would also be worthwhile to translate proof obligations along with the model thus allowing them to be discharged in Isabelle. The proof obligations are encoded as ASTs using only the expression subset of VDM. Therefore it should be possible to translate them with the existing machinery and require only some additional syntax to turn them into proof goals for Isabelle.

With regards to the CGP itself, the work presented in this paper has suggested two improvements to be carried out. The first is an architectural refactoring of the CGP. At the moment the CGP is directly tied to the Java code generator and that component must be reused as part of reusing the CGP. While this does not limit the ability to construct new extensions, it does expose a significant amount of Java-related functionality that is not necessary. Therefore, it would be beneficial to refactor the code generator into a *core* component that provides the CGP and a *javacg* component that provides code generation to Java.

Another improvement is related to transformations of the IR. Currently, a new extension must provide all of its transformations and develop them from scratch. It stands to reason that some translations are required for multiple extensions – for example, dependency sorting may be needed in other target languages – so it would be beneficial to reuse existing transformation. However, most transformations make assumptions about the target language and the order in which transformations are applied. This makes it quite challenging to reuse them since none of these assumptions hold in all situations.

8 Conclusion

This paper has presented a VDM to Isabelle translation using the code generation platform of Overture. The initial results show that the translation can be written with a significantly smaller amount of code (86%). Additionally, the use of the platform confers various benefits such as improved maintainability of the intermediary data structure and more easily adjustable syntax (via templates instead of Java strings). Also, any general improvements made to the CGP will be propagated to the translation as well.

The successful development of the Isabelle translation stands as proof of the extensibility of the CGP. Some issues were identified and addressed in order to increase extensibility. Specifically, a more generic transformation mechanism was implemented with support for changing the root node of the tree.

Our initial results show that it is quite worthwhile and beneficial to use the CGP for syntactical translations. The work presented here not only validates the extensibility of the CGP but it also provides a good basis for developing a complete VDM to Isabelle translation.

Acknowledgements

The authors would like to thank Simon Foster, Richard Payne and Sune Wolff for their valuable insight and comments on this paper.

References

1. ASTCreator (2014), `https://github.com/overturetool/astcreator`
2. Cormen, T.H., Stein, C., Rivest, R.L., Leiserson, C.E.: Introduction to Algorithms. McGraw-Hill Higher Education, 2nd edn. (2001)
3. Couto, L.D., Foster, S., Payne, R.: Towards Certification of Constituent Systems through Automated Proof. In: Workshop on Engineering Dependable Systems of Systems (EDSoS) (May 2014)
4. Couto, L.D., Tran-Jørgensen, P.W.V., Lausdahl, J.W.C.K.: Migrating to an Extensible Architecture for Abstract Syntax Trees. In: 12th Working IEEE / IFIP Conference on Software Architecture (May 2015)
5. E.Gamma, R.Helm, R., J.Vlissides: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series, Addison-Wesley Publishing Company (1995)
6. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object–oriented Systems. Springer, New York (2005), `http://overturetool.org/publications/books/vdoos/`
7. Foster, S., Payne, R.J.: Theorem Proving Support - Developers Manual. Tech. rep., COMPASS Deliverable, D33.2b (September 2013)
8. Foster, S., Zeyda, F., Woodcock, J.: Isabelle/UTP: A mechanised theory engineering framework. In: Unifying Theories of Programming, pp. 21–41. Springer (2015)
9. Hoare, T.: Communication Sequential Processes. Prentice-Hall International, Englewood Cliffs, New Jersey 07632 (1985)
10. Hoare, T., Jifeng, H.: Unifying Theorieses of Programming. Prentice Hall (April 1998)
11. JGraphT (2015), `http://jgrapht.org/`
12. Jørgensen, P.W., Couto, L.D., Larsen, M.: A Code Generation Platform for VDM. In: The Overture 2014 workshop (June 2014)
13. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic, vol. 2283. Springer Science & Business Media (2002)
14. Apache Velocity (2015), `http://velocity.apache.org/`
15. Woodcock, J., Cavalcanti, A., Fitzgerald, J., Larsen, P., Miyazawa, A., Perry, S.: Features of CML: a Formal Modelling Language for Systems of Systems. In: Proceedings of the 7th International Conference on System of System Engineering. IEEE (July 2012)

13

Automated translation of VDM to JML annotated Java

This paper has been submitted to a peer-reviewed journal.

[P81] Peter W. V. Tran-Jørgensen, Peter Gorm Larsen and Gary T. Leavens. *Automated translation of VDM to JML annotated Java*. Submitted to the International Journal on Software Tools for Technology Transfer (STTT), January, 2016, and invited for a second round of review.

International Journal on Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Automated translation of VDM to JML annotated Java

Peter W. V. Tran-Jørgensen · Peter Gorm Larsen · Gary T. Leavens

Received: date / Accepted: date

Abstract When a system specified using the Vienna Development Method (VDM) is realised using code generation, no guarantees are currently made about the correctness of the generated code. In this paper we improve code generation of VDM models by taking contract-based elements such as invariants and pre- and post conditions into account during the code generation process. The contract-based elements of the Vienna Development Method Specification Language (VDM-SL) are translated into corresponding constructs in the Java Modeling Language (JML) and used to validate the generated code against the properties of the VDM model. VDM-SL and JML are both Design-by-Contract (DbC) languages, with the difference that VDM-SL supports abstract modelling and system specification, while JML is used for detailed specification of Java classes and interfaces. We describe the semantic differences between the contract-based elements of VDM-SL and JML and formulate the translation as a set of rules. We further demonstrate how dynamic JML assertion checks can be used to ensure the consistency of VDM's subtypes when a model is code generated. The translator is fully automated and produces JML annotated Java programs that can be checked for correctness using JML tools.

Keywords Design by contract · Formal methods · VDM · Java · JML · Code generation

P. W. V. Tran-Jørgensen
Aarhus University, Aarhus, Denmark
E-mail: pvj@eng.au.dk

P. G. Larsen
Aarhus University, Aarhus, Denmark
E-mail: pgl@eng.au.dk

G. T. Leavens
University of Central Florida, Orlando, USA
E-mail: leavens@eecs.ucf.edu

1 Introduction

Design-by-Contract (DbC) is an approach for designing software based on concepts such as pre conditions, post conditions and invariants [21]. These concepts are referred to as "*contracts*", in accordance with a conceptual metaphor for the conditions and obligations of a business contract. An example of a formal method that uses DbC elements is the Vienna Development Method (VDM), which was originally developed at IBM in Vienna for the development of their compiler for PL/1 [3,9,7]. One way to realise a VDM specification in a programming language is through refinement [30]. This is a stepwise process by which one can transform a formal model into a program that can be verified to semantically satisfy its contracts [12].

Another way to realise a VDM specification is using code generation. The idea is for the generated code to be a refinement of the specification, but which is not achieved through step-wise refinement, but rather in one step through code generation translation rules. Code generation aims to reduce the resources needed to realise the model as well as to avoid introducing problems in the implementation due to manual translation of model into code. However, current VDM code generators do not make any guarantees about the correctness of the generated code, nor do they provide the necessary means to help check that the code meets the specification. Naturally, this casts doubts on the value of code generation as a way to realise a VDM model, since the goal is to develop software that meets the specification.

In this paper we improve code generation of VDM models by allowing the generated code to be checked against the system properties described by the VDM contracts. This helps ensure that the generated code meets the VDM specification, and is achieved as described in this paper.

Some DbC technologies are tailored to specify detailed designs of programming interfaces for a particular programming language [29]. An example of one such technology is the Java Modeling Language (JML) [4] – a formal specification language that uses DbC elements, written as specialized comments, to specify the behaviour of Java classes and interfaces. JML annotations can be analysed statically or checked dynamically using JML tools. Therefore, JML can be seen as a technology that serves to bridge the gap between an abstract system specification and its Java implementation.

In this paper we attempt to bridge this gap even further by proposing a way to automatically translate a specification written in the Vienna Development Method Specification Language (VDM-SL) to a JML-annotated Java implementation. Current VDM code generators either ignore or provide limited code generation support for the contract-based elements and type constraints of VDM. Ideally we should be able to preserve the contracts and type constraints when the system specification is implemented, since: (1) they serve to document the intention and properties of the system and (2) they can be used to check the system realisation for correctness. Ensuring that the contracts and type constraints, as originally specified in VDM, hold for the system implementation potentially requires a lot of extra checks to be added to the code. Adding these checks to the code manually, is tedious and prone to errors. Instead, these checks could be generated automatically. Representing contracts and type constraints in JML also has the advantage that these checks may be ignored by the Java compiler. This allows the system realisation to be executed without the overhead of checking the contracts and type constraints, if desired.

The two main contributions resulting from our work are (1) a collection of semantics-preserving rules for translating a VDM-SL specification to a JML-annotated Java program and (2) an implementation of these rules as an extension to Overture's [15,23] VDM-to-Java code generator [13]. The full version of the translator will be available in Overture version 2.3.2 onwards. Although the generated code, in principle, can be checked for correctness using any JML tool, we have mostly used the OpenJML [5] runtime assertion checker to validate our work — in particular by generating JML constructs supported by this tool.

The rules propose ways to translate the DbC elements of VDM-SL to JML annotations; these annotations are added to the Java code produced by Overture's Java code generator. The rules cover checking of pre conditions, post conditions and invariants, but the translator also produces JML checks to ensure that no type constraints are violated across the translation.

We present the rules one by one and demonstrate, using a case study model of an Automated Teller Machine (ATM), how the code generator extension translates a VDM-SL specification to JML annotated Java code.

Following this section, we describe DbC with VDM-SL and JML in section 2. We continue by explaining how Overture's code generator translates VDM-SL models to Java in section 3. Then we describe the rules used to translate a VDM-SL specification to a JML annotated Java program in sections 5 to 7. Finally we describe related work in section 8 and present future plans and conclude in section 9.

2 Design by contract with VDM-SL and JML

In this section we describe VDM-SL and JML. We cover different types and all the contract-based elements of VDM-SL, focusing specifically on the VDM-10 release, which we are targeting in our work. The JML constructs described in this section cover those that are used to implement the translation rules.

2.1 VDM-SL

VDM-SL is an ISO standardised sequential modelling language that supports description of data and functionality. The ISO standard has later been informally extended with modules to allow functionality to be imported and exported between modules. A module may define a single state component, which can be constrained by a state invariant. State is modified by assigning a new value to a state designator, which can be either a name, a field reference or a map or sequence reference, as described in the VDM language reference manual [17].

Module state, if specified, implicitly defines a record type, which is tagged with the state name and also defines the type of the state component. The state type can be used like any other record type explicitly defined by the modeller – the difference being that the state invariant [2] constrains the state type and thus every instance of this record type.

Data are defined by means of built-in basic types covering, for instance, numbers, booleans and characters. The basic types can be used to form new structured data types using built-in type constructors that support creation of union types, tuple types and record types. A type may also be declared as optional, which adds **`nil`** as a special value. For collections of values, VDM-SL supports sets, sequences and maps. The built-in data types, type constructors and collections can be used to form named user defined types, which can be constrained by invariants.

Subsequently we refer to these types as *named invariant types*. As an example, Listing 1 shows the definition of the named invariant type `Amount`, which is used to represent an amount of money deposited or withdrawn by an account holder. This type is defined based on natural numbers (excluding zero), i.e. the built-in basic type **`nat1`** in VDM-SL. For this particular example, we say that **`nat1`** is the *domain* *type* of `Amount`. We further constrain `Amount` using an invariant, by requiring a value of this type to be less than 2000.

```
types
Amount = nat1
inv a == a < 2000;
```

Listing 1 Example of a VDM-SL named invariant type.

2.1.1 Functional descriptions

In VDM, functionality can be defined in terms of functions and operations over data types with a traditional call-by-value semantics. Functions are referentially transparent and therefore they are not allowed to access or manipulate state directly, whereas operations are. Therefore, a function cannot call an operation.[1] In addition to accessing module state, operations may also use the **`dcl`** statement to declare local state designators which can be assigned to. Subsequently the term *functional description* will be used to refer to both functions and operations.

Functional descriptions can be implicitly defined in terms of pre- and post conditions, which specify the conditions which must hold before and after invoking the functional description. Alternatively, a functional description can be *explicitly* defined by means of an algorithm.

The pre condition of a function can refer to all the arguments of the function it guards. The same applies to the post condition of a function, which can also refer to the result of the execution using the reserved word **`RESULT`**. In Listing 2, `f` is a function from the input types I_1 through I_N to the result type `R`. Throughout this section we use `e` to represent an expression that ranges over variables, e.g. $e_1(i_1, ..., i_n)$ in Listing 2 —the body of `f`— is an expression whose value is determined by the input passed to `f`.

From the **`pre`** and **`post`** clauses of `f` function definitions are derived for the pre- and post conditions. These function definitions do not appear in the model, but they are used internally by the Overture interpreter to check for contract violations. However, to clarify, the pre- and post condition functions of `f` are shown in Listing 3. In this listing, `+>` specifies that `pre_f` and `post_f` are total functions, unlike partial functions, which use the `->` type constructor.

```
f:I_1*...*I_N -> R
f(i_1,...,i_n) == e_1(i_1,...,i_n)
pre e_2(i_1,...,i_n)
post e_3(i_1,...,i_n,RESULT);
```

Listing 2 User-defined VDM-SL function.

```
pre_f:I_1*...*I_N +> bool
pre_f(i_1,...,i_n) == e_2(i_1,...,i_n)

post_f:I_1*...*I_N*R +> bool
post_f(i_1,...,i_n,RESULT) ==
 e_3(i_1,...,i_n,RESULT)
```

Listing 3 Pre- and post condition functions for the function `f` shown in Listing 2.

Similarly, the pre- and post condition functions of an operation are also derived. A pre condition of an operation can refer to the state, `s`, before executing the operation, whereas the post condition of an operation can read both the before and after states. State access is achieved by passing copies of the state to the pre- and post condition functions. In Listing 4 `op` is a user-defined operation. The corresponding pre- and post condition functions are shown in Listing 5 where the parameters `s~` and `s` of `post_op` refer to the state before and after execution of `op`. We further use `S` to denote the record type used to represent the module's state.

```
op:I_1*...*I_N -> R
op(i_1,...,i_n) == e_1(i_1,...,i_n)
pre e_2(i_1,...,i_n,s)
post e_3(i_1,...,i_n,RESULT,s~,s);
```

Listing 4 User-defined VDM-SL operation.

```
pre_op:I_1*...*I_N*S +> bool
pre_op(i_1,...,i_n,s) == e_2(i_1,...,i_n,s)

post_op:I_1*...*I_N*R*S*S +> bool
post_op(i_1,...,i_n,RESULT,s~,s) ==
 e_3(i_1,...,i_n,RESULT,s~,s)
```

Listing 5 Pre- and post condition functions for the operation `op` shown in Listing 4.

The function descriptions in Listing 3 and Listing 5 assume that the pre- and post conditions are defined (using the **`pre`** and **`post`** clauses) and that the state of the module enclosing the functional description exists. For the cases where pre- and post conditions are not defined they can be thought of as functions that yield **`true`** for every input. Furthermore, when no state component is defined, the pre- and post condition functions simply omit the state parameters. Similarly, when an operation does not return a result (it specifies

[1] With the recent introduction of **`pure`** operations into VDM-10 (not to be confused with **`pure`** methods in JML) it has become possible to invoke operations, albeit **`pure`** ones, from a function. This feature was introduced to address issues with the object-oriented dialect of VDM, called VDM++, but was made available in every VDM-10 dialect (including VDM-SL).

void as the return type) the post condition function omits the **RESULT** parameter.

For each type definition `T` constrained by an invariant (such as that shown in Listing 1), a function is implicitly created to represent the invariant – see Listing 6. The Overture tool uses this function internally to check the consistency of values of type `T`[18].

```
inv_T : T +> bool
inv_T (t) == e(t);
```

Listing 6 Invariant function for type definition `T`.

2.1.2 Atomic Execution

Multiple consecutive statements are sometimes needed to update the state designators to make them consistent with the state invariant. In such situations, multiple assignments can be grouped in an **atomic** statement block as shown in Listing 7.

```
atomic
(
 sd₁ := exp₁;
 ...
 sdₙ := expₙ
)
```

Listing 7 The VDM **atomic** statement.

Given the types $\mathtt{T}_1$,...,$\mathtt{T}_n$ of the respective state designators $\mathtt{sd}_1$,...,$\mathtt{sd}_n$ it is as if the atomic statement is evaluated as shown in Listing 8:

```
let t₁ : T₁ = exp₁,
    ...
    tₙ : Tₙ = expₙ
in
(
 -- Turn off invariants
 sd₁ := t₁;
 ...
 sdₙ := tₙ;
 -- Turn on invariants
 -- Check invariants hold
);
```

Listing 8 The execution semantics of the **atomic** statement.

Executing the **atomic** statement block is semantically equivalent to first evaluating the right-hand sides, i.e. $\mathtt{exp}_1$, ...,$\mathtt{exp}_N$, of all the assignments before turning off invariant checks, and then binding the result to the corresponding state designators. After all the assignments have been executed, it must be ensured that all invariants hold.

There are three properties that follow from the evaluation semantics of the **atomic** statement block that are worth mentioning:

1. When evaluating the right-hand sides of the assignment statements, potential contract violations will get reported.
2. Temporary identifiers, used to store the right-hand side results, are explicitly typed and therefore violations of named invariant types for these variables will get reported. The explicit type annotations thus ensure that the right-hand side of a state designator assignment is checked to be consistent with the type of said state designator.
3. Assignment statements cannot see intermediate values of state designators.

2.2 JML

Although JML [19] is designed to specify arbitary sequential Java programs, in this subsection we only describe the features needed for the translation from VDM-SL.

A method specified with the **pure** modifier in JML is not permitted to have write effects; such methods are allowed to be used in specifications. Pure methods are used to translate VDM-SL functions.

A class invariant in JML should hold whenever the non-helper methods of that class are not being executed; thus invariants must hold in each method's before and after states. However, a method declared with the **helper** annotation in a type `T` does not have its pre- and post conditions augmented with `T`'s invariants. Helper methods (and constructors) must either be pure or private [19], so that the invariant will hold at the beginning and end of all client-visible methods [22]. The before and after states of non-helper methods and constructors are said to be *visible states*; thus invariants must hold in all visible states. JML distinguishes between instance and static invariants. An *instance* invariant can refer to the non-static (i.e., instance) fields of an object. A *static* invariant cannot refer to an object's non-static fields; thus static invariants are often used to specify properties of static fields.

An assertion can reference the invariant for an object explicity using a predicate of the form `\invariant_for(e)`, which is equivalent to the invariant for `e`'s static type [19, section 12.4.22].

In JML pre- and post conditions are written using the keywords **requires** and **ensures**, respectively. In the specification of a post condition, one writes `\old(e)` to refer to the before state value of an expression `e`. For example, an increment method that writes a field `count` could be specified as follows.

```
//@ requires count < Integer.MAX_VALUE;
//@ modifies count;
//@ ensures count == \old(count)+1;
void increment() { count++; }
```

Method post conditions may also use the keyword **result** to refer to the value returned by the method.

Specification expressions in JML can use Java expressions that are pure (have no write effects), and also some logical operators, such as implication ==>, and quantifiers such as **forall** and **exists**.

In addition to method pre- and post conditions, one can also write assertions anywhere a Java statement can appear, using JML's **assert** keyword. Such assertions must hold whenever they are executed.

One way to specify the abstract state of a class is to use JML's **ghost** variables. Ghost variables are specification-only variables and fields of objects that can only be used in JML specifications and in JML **set** statements. A set statement is an assignment statement whose target is a ghost variable.

By default, JML variables and fields may not hold the **null** value. However, should one wish to specify that all fields of a class may hold **null**, then one can annotate the class's declaration with **nullable_by_default**.

3 The VDM-SL-to-Java code generator

The JML translator is implemented as an extension to Overture's VDM-SL-to-Java code generator, which provides code generation support for the executable subset of VDM. Some insights into how the Java code generator currently translates a VDM-SL model to Java is therefore needed in order to understand how the JML translator works.

A module is represented using a **final** Java class with a **private** constructor, since VDM-SL does not support inheritance and a module cannot be instantiated. Due to the latter, both operations and functions are code generated as **static** Java methods.

Module state is represented using a **static** class field in the module class to ensure that only a single state component exists at any given time. The state component is represented using a record value, and as a consequence, an additional record type is generated to represent it.

Each variable in VDM-SL is passed by value, i.e. as a *deep copy*, when it is passed as an argument, appears on the right-hand side of an assignment or is returned as a result. As a consequence, aliasing can never occur in a VDM-SL model. Types are different in Java, where objects are modified via object references or pointers. Therefore different object references can be used to modify the same object. To avoid such aliasing in the generated code, data types are code generated with functionality to support value type behaviour.

Every record definition code generates to a class definition with accessor methods for reading and manipulating the fields. This class implements `equals` and `copy` methods to support comparison based on structural equivalence and deep copying, respectively. In this way the pass by value semantics of VDM-SL can be preserved in the generated code by invoking the `copy` method, which helps to prevent aliasing. Similarly the `equals` method can be invoked to compare code generated records based on structural equivalence rather than comparing addresses of object references. A record object can then be obtained by invoking the constructor of the record class or by invoking the `copy` method of an existing record object.

Java does not support the definition of aliases of existing types, such as the `Amount` named invariant type in Listing 1. Therefore, the Java code generator chooses not to code generate class definitions for these types. Instead, usage of a named invariant type is replaced with its domain type (described in subsection 2.1). Since the named invariant type is an alias of an existing type this is fine, as long as we make sure to check that the type invariant holds.

To assist the translation of VDM to Java, the existing Java code generator uses a runtime library, which among other things, includes Java implementations for some of the different VDM types and operators. The `Tuple` class, for example, is used to represent tuple types and enables construction of tuple values. Sets, sequences and maps are represented using the `VDMSet`, `VDMSeq` and `VDMMap` classes, which themselves are based on Java collections, and so on.

In addition to using the existing runtime library, the JML translator also contributes a small runtime library to aid the generation of JML checks. This runtime library, which we subsequently refer to as `V2J`, is an extension of the existing Java code generator runtime library. As we shall see in subsection 6.6, the `V2J` runtime is mostly used in the generated JML checks to ensure that collections respect the VDM type they originate from.

4 Case study

Throughout the paper we will demonstrate the translation rules using a case study model of an ATM. The model consists of a single module, `ATM` (shown in Listing 9), which uses a state definition to record information about

- The debit cards considered valid by the system (`validCards`).
- The debit card currently inserted into the ATM, if any (`currentCard`).
- If a valid PIN code has been entered (`pinOk`) for the debit card currently inserted into the ATM and,
- all the bank accounts known to the system (`accounts`).

For simplicity, Listing 9 omits invariants and type definitions and only shows the state definition and the signatures for some of the operations. As we proceed, in section 5 and section 6 we elaborate on the specifics of each definition and demonstrate the translation to JML annotated Java.

```
module ATM
definitions
state St of
 validCards : set of Card
 currentCard : [Card]
 pinOk : bool
 accounts : map AccountId to Account
 ...
operations
GetStatus : () ==> bool * seq of char
GetStatus () == ...

OpenAccount : set of Card * AccountId ==> ()
OpenAccount (cards,id) == ...

AddCard : Card ==> ()
AddCard (c) == ...

RemoveCard : Card ==> ()
RemoveCard (c) == ...

InsertCard : Card ==>
  <Accept>|<Busy>|<Reject>
InsertCard (c) == ...

EnterPin : Pin ==> ()
EnterPin (pin) == ...

ReturnCard : () ==> ()
ReturnCard () == ...

Withdraw : AccountId * Amount ==> real
Withdraw (id, amount) == ...

Deposit : AccountId * Amount ==> real
Deposit (id, amount) == ...
 ...
end
```

Listing 9 VDM-SL module representing an ATM.

5 Translating VDM-SL contracts to JML

In this section we present the rules used to translate the DbC elements of VDM-SL to JML annotations that are added to the generated Java code. For each of the elements, we describe, using the case study example, the approach used to translate the element to JML. This is afterwards generalised as a rule, which appears in a grey box.

5.1 Translating modules

Overture's Java code generator may sometimes introduce auxiliary variables that are initialised to **`null`** when it code generates some of the constructs of VDM. To avoid getting errors reported when checking the generated code with a JML tool, we allow **`null`** as a legal value by default for all references in the generated code.

1. Translating modules
Annotate every class output by the Java code generator with the **`nullable_by_default`** modifier to allow all references to use **`null`** as a legal value.

As a consequence we also have to guard against **`null`** values for variables that originate from VDM variables or patterns that do not allow **`nil`**.

5.2 Translating functional descriptions to JML

Recall that a VDM-SL function code generates to a **`static`** Java method. In addition, a VDM-SL function does not have side-effects and therefore the code generated version of the method can be annotated as JML **`pure`**.

2. Translation of functions
Any function – whether it is defined by the user or derived, e.g. from a **`pre`** or **`post`** condition clause – code generates to a **`static`** Java method that gets annotated with the **`pure`** modifier.

Operations, on the other hand, can read and manipulate the state of the enclosing module, or invoke other operations that may have side-effects. Therefore, the method that the operation code generates to cannot be annotated as JML **`pure`**.

When a VDM-SL definition (e.g. a functional description) is code generated to Java, the visibility of the corresponding Java definition can, in principle, be set according to whether the VDM-SL definition is exported (**`public`**) or not (**`private`**). In the presentation of the translation rules following this section, we omit explicit use of access specifiers in the rule formulation as we do not consider it crucial to our work.

5.3 Translating pre conditions to JML

In terms of semantics there is no difference between a pre condition in VDM-SL and JML. There are, however, interesting issues worth mentioning regarding how the JML generator implements the translation. We start by covering pre conditions of operations, and we end this subsection by describing how they differ from these of functions. As an example of how a VDM-SL pre condition is translated, consider the operation in Listing 10. This operation models money withdrawal from a bank account identified by the parameter `id`.

```
Withdraw : AccountId * Amount ==> real
Withdraw (id, amount) ==
let newBalance =
    accounts(id).balance - amount
in
(
 accounts(id).balance := newBalance;
 return newBalance;
)
pre
currentCard in set validCards and pinOk and
currentCard in set accounts(id).cards and
id in set dom accounts
```

Listing 10 VDM-SL operation for bank account withdrawal guarded by a pre condition.

In order to withdraw money from the account, we require that a valid card has been inserted, the PIN code is accepted, and that the bank account exists. In Listing 10 this is specified using a **pre** clause from which `pre_Withdraw` is derived. Although the definition of `pre_Withdraw` is not a visible part of the VDM-SL specification, it is shown in Listing 11 to clarify the relationship between the **pre** clause and the function derived from it.

```
pre_Withdraw: AccountId*Amount*St +> bool
pre_Withdraw (id, amount, St) ==
 St.currentCard in set St.validCards and
 St.pinOk and
 St.currentCard in set St.accounts(id).cards
 and id in set dom St.accounts
```

Listing 11 The signature of the pre condition function derived from the `Withdraw` operation.

The `Withdraw` operation code generates to the Java method shown in Listing 12. In addition, the pre condition function `pre_Withdraw` code generates to another method with the same name (not shown in any listing) that is invoked from the **requires** clause of the `Withdraw` method to check if the pre condition is met. In addition to the input parameters of the `Withdraw` method, the `pre_Withdraw` method also gets passed the state `St`.

```
//@ requires pre_Withdraw(id,amount,St);
public static Number Withdraw(final Number
    id, final Number amount) {...}
```

Listing 12 Code generated version of the `Withdraw` operation.

3. Translating the pre condition of an operation

Let `op` be a method code generated from a VDM-SL user-defined operation and let the signature of `op` be:
static `R op(`I_1 i_1`,...,`I_n i_n`)`
Then `op` has a code generated pre condition method `pre_op` that is **pure** and which in addition to the parameters of `op` also takes the state component `s` as an argument, i.e.
`/*@` **pure** `@*/` **static boolean**
`pre_op(`I_1 i_1`,...,`I_n i_n`,S s)`
To ensure that the pre condition check gets performed, we annotate `op` with the following **requires** annotation:
`//@` **requires** `pre_op(`i_1`,...,`i_n`,s);`

Rule 3 assumes the existence of a state component `s`. However, when the state of the module enclosing `op` is not defined, rule 3 changes to not include the state parameter in the definition of `pre_op`.

The example above considers the case where the pre condition is guarding an operation (i.e. `Withdraw`). As described in section 2, a pre condition is defined differently for a function than it is for an operation. In particular, the pre condition of a function does not get passed the state, so neither does the code generated version of it. We also note that the visibility of the pre condition function must be the same as that of the functional description it guards. Otherwise it cannot be invoked from the corresponding **requires** clause.

4. Translating the pre condition of a function

Let `f` be a method code generated from a VDM-SL user-defined function and let the signature of `f` be:
static `R f(`I_1 i_1`,...,`I_n i_n`)`
Then `f` has a code generated pre condition method `pre_f` that is **pure** and which accepts the same parameters as `f`, i.e.
`/*@` **pure** `@*/` **static boolean**
`pre_f(`I_1 i_1`,...,`I_n i_n`)`
To ensure that the pre condition check gets performed, we annotate `f` with the following **requires** annotation:
`//@` **requires** `pre_f(`i_1`,...,`i_n`);`

5.4 Translating post conditions to JML

Post conditions in VDM-SL and JML are semantically similar, although VDM-SL represents the post condition function as a derived function definition (as it was done for pre conditions). Post conditions (for operations) are, however, represented a bit differently in a VDM model – the reason being that a post condition is allowed to access the state before and after invocation of the operation it guards. Therefore, both the before and after states must be passed to the post condition function.

Returning to the `Withdraw` operation, one could specify a post condition requiring that no more money than that specified as `amount` be withdrawn from the account. This requirement is specified as a post condition in Listing 13.

```
Withdraw : AccountId * Amount ==> real
Withdraw (id, amount) ==
 ...
post
let accountPre = accounts~(id),
    accountPost = accounts(id)
in
 accountPre.balance =
 accountPost.balance + amount and
 accountPost.balance = RESULT;
```

Listing 13 The `Withdraw` operation guarded by a post condition.

The JML generator produces a **`pure`** Java method to represent the post condition function. This Java method is invoked from the **`ensures`** clause to check that the post condition holds. The invocation of the post condition method of the `Withdraw` operation is shown in Listing 14.

```
//@ requires pre_Withdraw(id,amount,St);
//@ ensures post_Withdraw(id,amount,\result
    ,\old(St.copy()),St);
public static Number Withdraw(final Number
    id, final Number amount) {...}
```

Listing 14 Code generated version of the `Withdraw` operation.

Note in particular how the before and after states are passed to the `post_Widthdraw` method. It follows from the semantics of the **`ensures`** clause that the before state of the method is referred to as `\old(St)`. Furthermore, reasoning about before state is done using the JML `\old` expression and for the `Withdraw` operation the before state is constructed as `\old(St.copy())`. Since `St.copy()` is a deep copy of the state (as explained in section 3) the evaluation inside the `\old` expression ensures that the result indeed is a representation of the before state.

Deep copying the state is needed since Java represents every composite data type using a class. So without deep copying the state, only the address of the before state object reference gets copied. In effect, only a single object would exist to represent the pre- and post states. This would never work, since state changes made by the operation would affect what was intended to be a representation of the before state. Therefore, the state is deep copied to get a separate object to represent the before state.

5. Translating the post condition of an operation

Let `op` be a method code generated from a VDM-SL user-defined operation and let the signature of `op` be:
`static` `R op(I1 i1,...,In in)`
Then `op` has a code generated post condition method `post_op` that is **`pure`** and which in addition to the parameters of `op`, also takes the result as well as the before and after states of `op` as arguments, i.e.
`/*@` **`pure`** `@*/` **`static boolean`**
`post_op(I1 i1,...,In in,`
`   R RESULT,S _s,S s)`
To ensure that the post condition check gets performed we annotate `op` with the following **`ensures`** annotation:
`//@` **`ensures`** `post_op(i1,...,in,\`**`result`**`,`
`   \`**`old`**`(s.copy(),s);`

Similar to rule 3, rule 5 also assumes the state of the module enclosing `op` to exist. If the state component is not defined, rule 5 changes to not include the state parameters in the definition of `post_op`. Furthermore, if `op` does not return a result (the return type is void), then the definition of `post_op` does not include the `RESULT` parameter.

The example above considers the post condition of an operation (i.e. `Withdraw`). As described in section 2, the post condition of a function is not allowed to access state. Therefore, the code generated version of the post condition function does not get passed the state.

6. Translating the post condition of a function

Let `f` be a method code generated from a VDM-SL user-defined function and let the signature of `f` be:
`static` `R f(I1 i1,...,In in)`
Then `f` has a code generated post condition method `post_f` that is **`pure`** and which in addition to the parameters of `f` also takes the result of `f` as an argument, i.e.
`/*@` **`pure`** `@*/` **`static boolean`**
`post_f(I1 i1,...,In in,R RESULT)`
To ensure that the post condition check gets performed we annotate `f` with the following **`ensures`** annotation:
`//@` **`ensures`** `post_f(i1,...,in,\`**`result`**`);`

5.5 Translating record invariants to JML

A record can, like any other type definition in VDM-SL, be constrained by an invariant. As an example, Listing 15 shows a record definition modelling a bank account.

```
Account ::
 cards : set of Card
 balance : real
 inv a == a.balance >= -1000;
```

Listing 15 A VDM-SL record definition modelling a bank account.

An `Account` comprises the available balance as well as the debit cards associated with the account. We further constrain an `Account` to not exceed a balance of -1000, which is expressed using an invariant.

As described in section 3, a record definition code generates to a class that emulates the behaviour of a value type using `copy` and `equals` methods.

Since a record invariant is required to hold for every record value, or object instance in the generated code, we represent it using an **`instance invariant`** in JML as shown in Listing 16. In particular, note that the **`instance invariant`** is formulated as an implication such that invariant violations do not get reported when invariant checks are disabled. As we shall see in subsection 5.6 this has to do with the way VDM-SL handles atomic execution.

```
//@ nullable_by_default
final public class Account implements Record
{
 public VDMSet cards;
 public Number balance;
 //@ public instance invariant atm.ATM.
     invChecksOn ==> inv_Account(cards,
     balance);
  ...
 /*@ pure @*/
 public boolean equals(final Object obj){..
     .}
 /*@ pure @*/
 public atm.ATMtypes.Account copy(){...}
 /*@ pure @*/
 public VDMSet get_cards() {...}
 public void set_cards(final VDMSet _cards)
     {...}
 /*@ pure @*/
 public Number get_balance() {...}
 public void set_balance(final Number
     _balance){...}
 /*@ pure @*/
 /*@ helper @*/
 public static Boolean inv_Account(final
     VDMSet _cards, final Number _balance){
     ...}
}
```

Listing 16 The code generated version of the `Account` record definition (omitting the bodies of the methods).

The code generated record `Account` defines an *invariant method* that takes all the record fields of `Account` as input and evaluates the invariant predicate. This method is invoked directly from the JML invariant, as shown in Listing 16. The invariant method is annotated as a **`helper`** to avoid the invariant check triggering another invariant check, which eventually would cause a stack-overflow.

7. Translating a record invariant

Let `R` be a code generated record definition with fields $f_1, \ldots, f_n$ of types $F_1, \ldots, F_n$, respectively, and let `R` be constrained by an invariant. Then `R` has an invariant method `inv_R` which is annotated as a **`helper`** to allow it to be invoked from the invariant clause of `R`. The invariant method can also be annotated as **`pure`** since it originates from a function definition. The annotated signature of `inv_R` thus becomes:

```
/*@ pure @*/
/*@ helper @*/
boolean inv_R(F1 f1,...,Fn fn)
```

Let further `invChecksOn` be a variable that is true if invariant checking is enabled and false otherwise. To represent the record invariant of `R` we annotate `R` with the **`invariant`** annotation:

```
/*@ public instance invariant
invChecksOn ==> inv_R(f1,...,fn); @*/
```

As we shall later see in subsection 5.6, atomic execution sometimes requires extra assertions to be inserted into the generated code in order to guarantee that the record invariant semantics of VDM-SL are preserved.

All the methods inside a record class – except for the constructor and the "setter" methods – do not modify the state of the record class and therefore they are marked as **`pure`**. Updates to a record object in the generated code are made using the "setter" methods of the generated record class, or by using the record modification expression [17]. Use of "setter" methods instead of direct field access to manipulate the state of a record (which is how field access is achieved in VDM-SL) forces the record object into a *visible state* (as described in subsection 2.2) after it has been updated, thus triggering the invariant check in accordance with the VDM-SL semantics. For example, in VDM-SL we could set the balance of an account as shown in Listing 17.

```
acc.balance := newBalance;
```

Listing 17 Updating the balance of an `Account` in VDM-SL.

This assignment generates to the Java code shown in Listing 18. Note that for this particular case there is no need to generate any additional JML assertions since the state of `acc` becomes visible after the call to `set_balance`. This causes the instance invariant check of the `Account` record class to trigger.

```
acc.set_balance(newBalance);
```

Listing 18 Updating the balance of an `Account` in the generated code.

5.6 Atomic Execution

There are situations where multiple assignment statements in VDM-SL need to be evaluated atomically in order to avoid unintentional violation of a state invariant. In our example, this is the case when the ATM returns the card to the owner, which is done as the last step of a transaction. Returning the debit card also requires us to invalidate the PIN code currently entered. These two things have to be done atomically to avoid violating the state invariant of the `ATM` module. Therefore the body of the `ReturnCard` operation is executed inside an **`atomic`** statement block as shown in Listing 19.

```
ReturnCard : () ==> ()
ReturnCard () ==
atomic
(
 currentCard := nil;
 pinOk := false;
)
pre currentCard <> nil
post currentCard = nil and not pinOk;
```

Listing 19 VDM-SL operation modelling removal of the debit card from the ATM.

JML does not include a syntactic construct similar to that of the **`atomic`** statement. Instead atomic execution must be achieved using different means – for example by manipulating state directly using field access or **`helper`** methods.

To be consistent with regards to the way record state is updated, and to reflect the way that VDM-SL handles atomic execution, we believe a better approach is to use a flag that indicates if invariant checks are enabled or not. Since this flag should not affect the generated code, we make it a **`ghost`** variable such that it is only visible at the specification level. Since this **`ghost`** variable must be accessible everywhere in the translation, we make it a static field of the class, as shown in Listing 20. The **`ghost`** variable must be added to one of the generated Java classes since Java does not really have global variables. Note that this flag does not affect pre- and post conditions since these checks must always be evaluated.

```
/*@ public ghost static boolean invChecksOn
    = true; @*/
```

Listing 20 Ghost variable used to control invariant checking.

The declaration of `invChecksOn` allows us to formulate invariants such that violations only get reported if invariant checking is enabled. An example of this is shown in Listing 21 for the record state class of the `ATM` module.

```
//@ public instance invariant atm.ATM.
    invChecksOn ==> inv_St(validCards,
    currentCard,pinOk,accounts);
```

Listing 21 The invariant of the record state class.

The `invChecksOn` flag provides us with the means to emulate the behaviour of atomic execution in a Java environment as shown in Listing 22. More specifically we use the JML **`set`** statement to disable and enable invariant checking before and after executing the body of the `ReturnCard` method, respectively.

```
//@ requires pre_ReturnCard(St);
//@ ensures post_ReturnCard(\old(St.copy()),
    St);
public static void ReturnCard() {
 atm.ATMtypes.Card atomicTmp_1 = null;

 //@ assert ((atomicTmp_1 == null) || Utils.
     is_(atomicTmp_1,atm.ATMtypes.Card.class
     ));
 Boolean atomicTmp_2 = false;
 //@ assert Utils.is_bool(atomicTmp_2);
 { /* Start of atomic statement */
   //@ set invChecksOn = false;

   //@ assert St != null;
   St.set_currentCard(Utils.copy(atomicTmp_1
       ));

   //@ assert St != null;
   St.set_pinOk(atomicTmp_2);

   //@ set invChecksOn = true;

   //@ assert \invariant_for(St);
 } /* End of atomic statement */
}
```

Listing 22 The code generated version of the `ReturnCard` operation.

8. Enabling and disabling invariant checking

Declare in module `M` a globally accessible JML **`ghost`** variable `invChecksOn` to control invariant checking:

```
/*@ public ghost static
boolean invChecksOn = true; @*/
```

Before executing the code generated atomic statement, invariant checking is disabled using the following JML **`set`** statement:

```
//@ set M.invChecksOn = false;
```

After the code generated atomic block has finished executing invariant checking is re-enabled using:

```
//@ set M.invChecksOn = true;
```

When all the statements have been executed it must be ensured that no invariants have been violated. For the example in Listing 22, the only thing that needs to be checked is that the state component of the `ATM` class, i.e. `St` does not violate its invariant. Finally, we ensure that the invariant of `St` holds by asserting `\invariant_for(St)`.

The `\invariant_for` construct is not currently supported by OpenJML, and therefore we also allow the invariant check to be generated in a way that this tool supports.

Throughout this paper we use the `\invariant_for` construct to explicitly check the invariant of a record whenever needed. We choose to demonstrate our work using this approach as we believe it makes it easier to understand what we are tying to achieve.

The JML generator keeps track of state designators of records that potentially have been updated as part of executing the code generated atomic statement block. Immediately after invariant checking is re-enabled, i.e. the code generated atomic statement block has finished execution, it is checked that no record violates its invariant.

9. Resuming invariant checking

Let $d_1,...,d_n$ be state designators of records which have been updated, or affected by an update, during execution of a code generated atomic statement block. Further assume that $d_1,...,d_n$ have been updated in the given order, i.e. d_i was updated before d_{i+1} and that d_i may be of one of m_i record types $D_{i1},...,D_{im_i}$. Immediately after executing the code generated atomic statement block, it is checked that $d_1,...,d_n$ do not violate any invariants using the following sequence of `assert` statements:

```
//@ assert d1 instance of D11 ==>
  \invariant_for((D11) d1);
...
//@ assert d1 instance of D1m1 ==>
  \invariant_for((D1m1) d1);
...
//@ assert dn instance of Dn1 ==>
  \invariant_for((Dn1) dn);
...
//@ assert dn instance of Dnmn ==>
  \invariant_for((Dnmn) dn);
```

A state designator can be "masked" as a union type and in such situations it cannot always be statically determined what the runtime type of a state designator will be. To demonstrate this, consider the record types `R1` and `R2` and a state designator declared as `dcl r : R1 | R2 := ...`. Further assume that `R1` and `R2` code generate to classes $\texttt{R1}_c$ and $\texttt{R2}_c$. After updating `r` atomically in the generated code, it is ensured that `\invariant_for((R1c) r)` holds if `r` is of type $\texttt{R1}_c$, and similarly that the equivalent condition is true if `r` is of type $\texttt{R2}_c$. Since rule 9 has to take all possible types into account, the invariant checks are formulated as implications.

Although the VDM type system allows state designators to be "masked" as union types, most of the time it is possible to statically determine the runtime type of a state designator. For example, in Listing 22 no `instanceof` check is needed since the static type of the state component is `St`. This is an example where the JML generator simplifies the check proposed by rule 9.

There are more aspects to rule 9 worth discussing – especially when state designators are based on arbitrarily complex data structures such as nested records. This will be addressed in subsection 7.1.

5.7 Translating module state to JML

As described in subsection 2.1, a module state invariant constrains the record type used to represent the state component of the enclosing module. Therefore, a module state invariant can essentially be seen as a record invariant that can be translated into JML annotated Java without introducing additional translation rules. This subsection instead explains how a VDM-SL state definition is translated into a form that allows the rules related to record invariants to be applied (see subsection 5.5).

In our example each account can be accessed from an ATM using one of the debit cards associated with it. In addition to the bank accounts, the state of the ATM also keeps track of: the debit cards that the systems considers valid, the debit card which is currently inserted into the ATM, and whether the PIN code entered by the user is valid. The state as specified in VDM-SL is shown in Listing 23. Note that this listing also includes the state initialisation and the invariant.

```
state St of
 validCards : set of Card
 currentCard : [Card]
 pinOk : bool
 accounts : map AccountId to Account
 init St == St = mk_St({},nil,false,{|->})
 inv mk_St(v,c,p,a) ==
  (p or c <> nil => c in set v)
  and
  forall id1, id2 in set dom a &
   id1 <> id2 =>
   a(id1).cards inter a(id2).cards = {}
end
...
types
Card ::
 id : nat
 pin : Pin;
```

Listing 23 State of the `ATM` module and the `Card` type definition.

The invariant requires that, at all times the following two conditions must be met: a debit card must at most be associated with a single account and secondly, for a PIN code to be considered valid, the debit card currently inserted into the ATM must itself be a valid debit card. Based on the state definition, a record class gets generated which represents the state type as shown in Listing 24.

```
final public class St implements Record {
 public VDMSet validCards;
 public atm.ATMtypes.Card currentCard;
 public Boolean pinOk;
 public VDMMap accounts;

 //@ public instance invariant atm.ATM.
     invChecksOn ==> inv_St(validCards,
     currentCard,pinOk,accounts);
 /* Record methods omitted */
}
```

Listing 24 The record class used to represent the state type.

In addition, an instance of the record class gets created to represent the state component as shown in Listing 25. The state component is annotated with the **`spec_public`** modifier so that it can be referred to from the **`requires`** and **`ensures`** clauses of **`public`** methods. Also note that the module is not constrained by an invariant. This is handled entirely by the record invariant shown in Listing 24.

```
final public class ATM {
 /* Fields omitted */

 /*@ spec_public @*/
 private static atm.ATMtypes.St St = new atm
     .ATMtypes.St(SetUtil.set(),
            null, false, MapUtil.map());
 /* Module methods omitted */
}
```

Listing 25 The state component in the `ATM` module.

10. Translating the state component
Annotate state components of module classes with the **`spec_public`** modifier to ensure that the state components can be referred to from the **`requires`** and **`ensures`** clauses of **`public`** methods.

6 Checking VDM types using JML

In this section we describe how the translator uses JML to check the consistency of VDM types when they are code generated.

Throughout this section we construct a function called `Is(v,T)` which takes as input a Java value `v` and a VDM type `T` and produces a JML expression that can be used to check whether `v` represents a value of type `T`. We use `Is(v,T)` to check whether a Java value remains consistent with the VDM type that it originates from. The check produced by `Is(v,T)` can be added to the generated Java code to ensure that no type violations occur.

We cover the different classes of VDM types, one by one, and explain, using our case study example, how JML is used to check a Java value against the VDM type that it originates from. Based on this we gradually extend the coverage of `Is(v,T)` to include checking of more types, and continue like this until we have covered all the VDM types supported by the Java code generator. Finally, we summarise and provide the complete definition of `Is(v,T)` (see Figure 1).

6.1 Where to generate dynamic type checks

Most of the types available in VDM are also present in Java in some form or other. The VDM and Java type systems do, however, have some differences that require us to generate extra checks to ensure that a Java value remains consistent with the VDM type that it originates from.

In addition to producing the JML expression needed to check the consistency of a type, i.e. `Is(v,T)`, we also need to consider where to add the check to the generated code. The description below summarises the VDM-SL constructs that must be considered when adding these checks to the generated Java code. We use the term *parameter* to refer to an identifier that cannot change its value. A parameter can be defined using a **`let`** construct, which is different from a state designator or variable that can be locally defined using a **`dcl`** statement or globally using a state definition (see section 2). The constructs to be considered are:

- **`return`** statement: If a function or operation returns a value, as specified using the result type in the signature of the enclosing function or operation, then it must be checked whether the value returned violates the result type.
- Parameters of functions and operations: The arguments passed to a functional description, when invoked, are mapped to the formal parameters of said functional description. Formal parameters are typed variables, thus subject to type constraints. Upon entering a function or operation, the value received by each formal parameter must be checked against its type.
- State designators: After updating a local or global state designator, the new value assigned must respect the type of the state designator.
- Variable or parameter declaration: After initialising a variable or parameter it must be checked against its declared type.
- Value definition: An explicitly typed value definition must specify a value consistent with its type.

All of the constructs in the list above – with the exception of the value definition – can be checked using a JML **`assert`** statement. The reason for this is that the code generated versions of these constructs appear inside methods in the generated code. Since a VDM value definition code generates to a **`public static final`** field (a constant) it is checked using a **`static`** invariant.

6.2 Translating basic types

In our example we may wish to check that the amount being withdrawn from an account is valid – for example by requiring that it is a natural number larger than zero, as shown in Listing 26.

```
let amount : nat1 = expense - profit
in
  Withdraw(accId, amount);
```

Listing 26 Use of explicit type annotation to ensure that a valid amount is being withdrawn.

In the generated Java code, shown in Listing 27, this is checked by analysing the value of the `amount` variable using the `Utils.is_nat1` method available from the Java code generator runtime library. This method is invoked from a JML annotation in order to check that `amount` is different from **`null`** and that it represents an integer larger than zero.

```
Number amount =
  expense.longValue() - profit.longValue();
//@ assert Utils.is_nat1(amount);
return Withdraw(accId, amount);
```

Listing 27 Use of JML to check that a valid amount is being withdrawn.

11. Checking of the `nat1` type

Let `v` be a value or object reference in the generated code that originates from a variable or pattern of type **`nat1`** and further define `Is(v,nat1) = Utils.is_nat1(v)`. To ensure that `v` represents a value of type **`nat1`**, generate a JML check to ensure that `Is(v, nat1)` holds.

The approach used to check other basic types follow the principles demonstrated using Listing 26 and Listing 27 – the main difference being that each basic type uses a dedicated method from the Java code generator runtime library. Therefore, we omit the details of how other basic types of VDM are checked using JML, and instead provide the complete set of rules in Figure 1.

We note that a record type or a quote type can be checked in a way similar to that of a basic type. The reason for this is that the Java generator produces a Java class for each of the record definitions and quote types in the VDM model. Therefore, all there is to checking whether an object reference represents a given record or quote class is to check whether the object reference is an instance of said class. The rules for checking record and quote types are included in Figure 1.

6.3 Translating optional types

To demonstrate how the JML generator handles optional types consider the `GetCurrentCardId` operation in Listing 28. This operation returns the identification of the debit card currently inserted into the machine, if any. Otherwise the operation returns **`nil`** to indicate the absence of a debit card. To allow **`null`** as a return value, the optional type operator is used to specify the return type of the operation as `[nat]`.

```
GetCurrentCardId : () ==> [nat]
GetCurrentCardId () ==
 if currentCard <> nil then
   return currentCard.id
 else
   return nil;
```

Listing 28 Operation for getting the id of the debit card currently inserted into the ATM.

Considering solely the signature of the code generated version of this operation, shown in Listing 29, there is no way to tell that the return type represents a `[nat]`.

```
public static Number GetCurrentCardId(){...}
```

Listing 29 Signature of the code generated version of the `GetCurrentCardId` operation.

The reason for this is that the Java code generator uses the `Number` class (which is part of the Java standard library) to represent all numeric VDM types. That the return type of the operation is `[nat]` only becomes apparent when we start using the corresponding method.

To demonstrate this, the Java fragment in Listing 30 uses the result of invoking the `GetCurrentCardId` method to initialise a variable named `id`. The initialisation of `id` is immediately followed by a check that ensures that it represents either **`null`** or a natural number. The approach of allowing **`null`** values like this is the same for all optional types.

```
Number id = GetCurrentCardId();
//@ assert id == null || Utils.is_nat(id);
```

Listing 30 Use of the `GetCurrentCardId` method in the generated code.

12. Checking of optional types

Let `v` be a value or object reference in the generated code that originates from a variable or pattern of the VDM type `[T]` and further define
`Is(v,[T]) = (v == null || Is(v,T))`
To ensure that `v` represents a value of type `[T]`, generate a JML check to ensure that `Is(v, [T])` holds.

6.4 Translating tuple types

In our case study example we use a tuple type to represent the status of the ATM: the first field is a **boolean** flag that indicates if the ATM is currently awaiting a debit card to be inserted, and the second field is a human-readable description of the current state of the ATM, e.g. “transaction in progress”. The signature of the operation that retrieves the status of the ATM is shown in Listing 31. Note in particular that the status returned is represented using the tuple type **bool * seq of char**.

```
GetStatus : () ==> bool * seq of char
GetStatus () == ...
```

Listing 31 The signature of the `GetStatus` operation.

In the generated Java code, every tuple value is represented as an instance of the `Tuple` class available from the Java code generator runtime library. Since the `Tuple` class represents tuple values in general, each instance of this class must be checked against the specific tuple type that it originates from.

After the status of the ATM has been retrieved using the `GetStatus` method in the generated code, the status is checked as shown in Listing 32. First it is checked that `status` is a tuple of size two. Afterwards it is checked that the first field is a **boolean** and that the second field is a Java `String` (which represents the **seq of char** type).

```
Tuple status = GetStatus();
//@ assert (V2J.isTup(status,2) && Utils.
    is_bool(V2J.field(status,0)) && Utils.
    is_(V2J.field(status,1),String.class));
```

Listing 32 Checking the ATM status in the generated code.

13. Checking of tuple types

Let `v` be a value or object reference in the generated code that originates from a variable or pattern of the VDM tuple type `T`$_1$`*...*T`$_n$ and further define
`Is(v,T`$_1$`*...*T`$_n$`) = V2J.isTup(v,n) &&`
`Is(v,T`$_1$`) &&...&& Is(v,T`$_n$`)`
To ensure that `v` represents a value of type `T`$_1$`*...*T`$_n$, generate a JML check to ensure that `Is(v,T`$_1$`*...*T`$_n$`)` holds.

6.5 Translating union types

Attempting to insert a debit card into the ATM results in the debit card being accepted, if no card is currently inserted and it is considered a valid card by the system. Otherwise the card is rejected. Based on the outcome of this the `NotifyUser` operation, shown in Listing 33, displays a message to inform the card holder about the current status of the session. This operation uses a union type, formed by the three quote types `<Accept>`, `<Busy>` and `<Reject>`, to represent one of three outcomes of the card holder attempting to insert a debit card into the ATM.

```
NotifyUser : <Accept>|<Busy>|<Reject> ==> ()
NotifyUser (outcome) ==
if outcome = <Accept> then
  Display("Card accepted")
elseif outcome = <Busy> then
  ...
```

Listing 33 Operation used to notify a ATM user.

The code generated version of the `NotifyUser` operation is shown in Listing 34. Since the `outcome` parameter originates from the union type formed by the three quote types, it must be checked that `outcome` equals one of the three possible values. This check is performed immediately after entering the `NotifyUser` method, as shown in Listing 34.

```
public static void NotifyUser(final Object
    outcome) {
 //@ assert (Utils.is_(outcome,atm.quotes.
     AcceptQuote.class) || Utils.is_(outcome
     ,atm.quotes.BusyQuote.class) || Utils.
     is_(outcome,atm.quotes.RejectQuote.
     class));
 if (Utils.equals(outcome, atm.quotes.
     AcceptQuote.getInstance())) {
   Display("Card␣accepted");
 } else if (Utils.equals(outcome, atm.quotes
     .BusyQuote.getInstance())){
   ...
}
```

Listing 34 Code generated version of the `NotifyUser` operation.

14. Checking of union types

Let `v` be a value or object reference in the generated code that originates from a variable or pattern of the VDM union type `T`$_1$`|...|T`$_n$ and further define
`Is(v,T`$_1$`|...|T`$_n$`) = Is(v,T`$_1$`) ||...||`
`Is(v,T`$_n$`)`
To ensure that `v` represents a value of type `T`$_1$`|...|T`$_n$, generate a JML check to ensure that
`Is(v,T`$_1$`|...|T`$_n$`)` holds.

6.6 Translating collections

In the generated code the `VDMSet`, `VDMSeq` and `VDMMap` collection classes are used as raw types. Therefore the code generator does not take advantage of Java generics to make compile-time guarantees about the types of the objects a collection store. This approach has the advantage of making it easier to store Java objects and values of different types in the same collection without having to introduce additional types. Although this allows the type system of VDM to be represented in Java (which generally uses a stronger type system than VDM) it has the disadvantage that no compile-time guarantees can be made about the types of the objects that a collection stores.

In the ATM example we use the `TotalBalance` function, shown Listing 35, to calculate the total balance available from a set of accounts.

```
TotalBalance : set of Account -> real
TotalBalance (acs) ==
 if acs = {} then
   0
 else
  let a in set acs
  in
    a.balance + TotalBalance(acs \ {a});
```

Listing 35 Operation that calculates the total balance available from a set of accounts.

When the `TotalBalance` function is code generated to JML annotated Java, the code generator adds JML assertions to ensure that the set of accounts is consistent with the collection type used in VDM. Since an `Account` record is represented using a Java class with the same name, we have to check that every element in the set is an instance of said Java class. As shown in Listing 36, this is checked using a quantified expression. This expressions uses a bound variable `i` to iterate over all the accounts and check that each element is an instance of the `Account` record class. Although sets are unordered collections, the quantified expression takes advantage of `VDMset` being implemented as an ordered collection. The formulation of the range expression in the quantified expression further ensures that the assertion can be checked using a tool such as the OpenJML runtime assertion checker, i.e. the assertion is executable.

```
/*@ pure @*/
public static Number TotalBalance(final
    VDMSet acs) {
 //@ assert (V2J.isSet(acs) && (\forall int
     i; 0 <= i && i < V2J.size(acs); Utils.
     is_(V2J.get(acs,i),atm.ATMtypes.Account
     .class)));
 if (Utils.empty(acs)) {
   Number ret_1 = 0L;

   //@ assert Utils.is_real(ret_1);
   return ret_1;
 } else { ... /*Compute sum recursively */}
}
```

Listing 36 Code generated version of the `TotalBalance` operation.

15. Checking of sets
Let `v` be a value or object reference in the generated code that originates from a variable or pattern of the VDM set type **`set of`** `T` and further define `Is(v,`**`set of`**` T) = V2J.isSet(v) && (\`**`forall int`**` i; 0 <= i && i < V2J.size(v); Is(V2J.get(v,i),T))` To ensure that `v` represents a value of type **`set of`** `T`, generate a JML check to ensure that `Is(v,`**`set of`**` T)` holds.

The VDM sequence types **`seq`** and **`seq1`** are checked in a way similar to sets. The difference between checking the **`seq`** and **`seq1`** collection types is that the **`seq1`** type requires at least one element to be present in the sequence. Checking a map, which like a set is an unordered collection, takes advantage of `VDMMap` imposing an order on the domain and range values. The main difference between checking a map and a set is that both the domain and range values of a map have to be checked. Checking the injective map type **`inmap`** is similar to checking a standard map, except that the injectivity property must hold. We refrain from providing examples of how to check each of the collection types in VDM since they are very similar to what has already been shown. Instead we summarise the rules for checking all of the collection types in Figure 1.

6.7 Translating named invariant types to JML

Since the Java code generator does not generate additional class definitions for named invariant types, the invariant imposed on such a type cannot be expressed as a JML invariant. This is only possible for a record since it translates to a class definition.

Instead, we identify places in the generated code where a named invariant type may be violated, as described in subsection 6.1, and check that the invariant holds. Also, it is worth noting that a named invariant type, unlike a record type, does not have an explicit type constructor. Therefore, an expression can only violate a named invariant type if the expression is explicitly declared to be of that type.

The ATM in our example is not capable of dispensing cents and also imposes a limit on the amount of money that can be withdrawn. Therefore, the amount of money can be represented as a named invariant type. An attempt to withdraw an amount of money that exceeds 2000 will yield a runtime error. The named invariant type used to represent the amount withdrawn from an account is shown together with the `Withdraw` operation in Listing 37.

```
types
Amount = nat1
```

```
inv a == a < 2000;

operations
Withdraw : AccountId * Amount ==> real
Withdraw (id, amount) == ...
```

Listing 37 The amount to withdraw modelled using a named invariant type.

On entering the code generated version of the `Withdraw` operation, shown in Listing 38, we assert that `amount` meets the named invariant type `Amount`.

```
...
public static Number Withdraw(final Number
    id, final Number amount){
...
//@ assert (Utils.is_nat1(amount) &&
    inv_ATM_Amount(amount));
 ...
}
```

Listing 38 Checking a named invariant type of a operation parameter in JML.

The assertion does two things: First it performs a dynamic type check to ensure that `amount` is a valid domain type of `Amount` and secondly, it checks that the invariant predicate holds. For the example in Listing 38 this means checking that `amount` is of type **`nat1`** and smaller than 2000. Note that meeting the invariant condition does not imply compatibility with the domain type of the named invariant type and vice versa. For example, -1 is smaller than 2000 but it is not of type **`nat1`**. Likewise, 2001 is of type **`nat1`** but it exceeds 2000 so neither -1 nor 2001 are of type `Amount`.

The code generated invariant method `Amount` is shown in Listing 39. Since the named invariant type check, shown in Listing 38, is evaluated from left to right using McCarthy evaluation semantics, the invariant method is only invoked if the value subject to checking is compatible with the domain type of the named invariant type. Therefore, it is safe to narrow (or cast) the type the argument passed to the invariant method before performing the invariant check.

```
/*@ pure @*/
/*@ helper @*/
public static Boolean inv_ATM_Amount(final
    Object check_a) {
 Number a = ((Number) check_a);
 return a.longValue() < 2000L;
}
```

Listing 39 The code generated named invariant type method for `Amount`.

16. Checking of named invariant types

Let `v` be a value or object reference in the generated code that originates from a variable or pattern of the VDM named invariant type `T` based on the domain type `D` and constrained by invariant predicate `e(p)`, i.e. `T` is defined as

```
types
T = D
inv p == e(p)
```

Then `T` has an invariant method, responsible for running the code generated version of the `e(p)` check, with a signature defined as:

```
public static boolean inv_T(Object o)
```

Further define

```
Is(v,T) = Is(v,D) && inv_T(v)
```

To ensure that `v` represents a value of type `T`, generate a JML check to ensure that `Is(v,T)` holds.

Note that the invariant method `inv_T` in rule 16 defines the input parameter `o` to be of type `Object`, thus allowing `inv_T` to accept inputs of any type. Therefore, `inv_T` must narrow the type of the input parameter `o` before performing the invariant check (see the example in Listing 39). This approach has the advantage that it allows simpler JML checks since the argument type does not need to be narrowed before the invariant method is invoked. Had the input parameter of the invariant method been defined using the smallest possible type, then the argument type would need to be narrowed for situations where the argument is masked as a union type. Although this would complicate the JML checks, it would have the advantage of allowing type narrowing to be removed from the invariant methods.

7 Other aspects of VDM-SL affecting the JML generation

There are other aspects of VDM-SL that further complicate the generation of VDM-SL models to JML annotated Java. In this section we use examples to demonstrate these issues and explain how they may be overcome.

7.1 Complex state designators

State designators may be composite data structures such as records with fields which themselves are records. Such a data type forms *complex state designators* that when modified require careful handling during the translation process. To demonstrate this, consider the three VDM-SL record definitions `R1`, `R2` and `R3` in Listing 40. Note in particular how the invariants of `R1` and `R2` depend on the field of `R3`. This transitive dependency complicates checking of invariants in the generated code. To demonstrate this, the operation in

`Is(v,T) =`

`Utils.is_bool(v)`	**if** `T` = **bool**
`Utils.is_nat(v)`	**if** `T` = **nat**
`Utils.is_nat1(v)`	**if** `T` = **nat1**
`Utils.is_int(v)`	**if** `T` = **int**
`Utils.is_rat(v)`	**if** `T` = **rat**
`Utils.is_real(v)`	**if** `T` = **real**
`Utils.is_char(v)`	**if** `T` = **char**
`Utils.is_token(v)`	**if** `T` = **token**
`Utils.is_(v,String.`**class**`)`	**if** `T` = **seq of char**
`Utils.is_(v,`S_{CG}`.`**class**`)`	**if** `T` is a record or quote type `S` that generates to a Java class with the fully qualified name S_{CG}
`(v == `**null**` || Is(v,S))`	**if** `T = [S]`
`V2J.isTup(v,n) && Is(v,`T_1`) &&...&& Is(v,`T_n`)`	**if** `T` = T_1`*...*`T_n
`Is(v,`T_1`) ||...|| Is(v,`T_n`)`	**if** `T` = T_1`|...|`T_n
`V2J.isSet(v) && (\`**forall int**` i; 0 <= i && i < V2J.size(v); Is(V2J.get(v,i),S))`	**if** `T` = **set of** `S`
`V2J.isSeq(v) && (\`**forall int**` i; 0 <= i && i < V2J.size(v); Is(V2J.get(v,i),S))`	**if** `T` = **seq of** `S`
`V2J.isSeq1(v) && (\`**forall int**` i; 0 <= i && i < V2J.size(v); Is(V2J.get(v,i),S))`	**if** `T` = **seq1 of** `S`
`V2J.isMap(v) && (\`**forall int**` i; 0 <= i && i < V2J.size(v); Is(V2J.getDom(v,i),D) && Is(V2J.getRng(v,i),R))`	**if** `T` = **map** `D` **to** `R`
`V2J.isInjMap(v) && (\`**forall int**` i; 0 <= i && i < V2J.size(v); Is(V2J.getDom(v,i),D) && Is(V2J.getRng(v,i),R))`	**if** `T` = **inmap** `D` **to** `R`
`Is(v,D) && inv_T(v)`	**if** `T` is named invariant type with domain type `D` and invariant method `inv_T`

Fig. 1 Complete definition of `Is(v,T)`.

Listing 40 instantiates `R1` as `r1` and modifies it to violate the `R1` invariant, which causes a runtime-error to be reported.

```
types

R1 :: r2 : R2
inv r1 == r1.r2.r3.x <> -1;

R2 :: r3 : R3
inv r2 == r2.r3.x <> -2;

R3 :: x : int
inv r3 == r3.x <> -3;

operations

op: () ==> nat
op () ==
(
 dcl r1 : R1 := mk_R1(mk_R2(mk_R3(5)));
 r1.r2.r3.x := -1;
 return 0;
)
```

Listing 40 Record nesting in VDM-SL.

The operation `op` in Listing 40 generates to the method in Listing 41. In the generated code `r1`, `stateDes_1` and `stateDes_2` represent the three state designators `r1`, `r2` and `r3` from Listing 40 respectively. Note that in Listing 41 we have removed fully qualified names of record classes as well as other JML checks, which are not related to the point we want to make.

```
/* Type definitions omitted */ ...
public static Number op() {
 R1 r1 = new R1(new R2(new R3(5L)));

 R2 stateDes_1 = r1.get_r2();
 R3 stateDes_2 = stateDes_1.get_r3();
 stateDes_2.set_x(-1L);

 //@ assert \invariant_for(stateDes_1);
 //@ assert \invariant_for(r1);

 Number ret_1 = 0L;
 return ret_1;
}
```

Listing 41 Code generated version of the operation from Listing 40.

Immediately after completing the state update, i.e. invoking `stateDes_2.set_x(-1L)`, the following things happen:

1. The state of `stateDes_2` becomes *visible* thus triggering the invariant check of `stateDes_2`.
2. The invariant check of `stateDes_1` is run by asserting `\invariant_for(stateDes_1)` and finally,
3. the invariant check of `r1` is run by asserting `\invariant_for(r1)` which causes the invariant of `r1` to be violated and a runtime-error to be reported.

Strictly speaking the objects pointed to by `stateDes_1` and `r` are also in visible states after executing the update to `stateDes_2` and therefore the invariants of those objects should also hold. In particular a state is *visible* for an object `o` *"when no constructor, destructor, non-static method invocation with* `o` *as receiver, or static method invocation for a method in* `o`*'s class or some superclass of* `o`*'s class is in progress[19]"*. So in theory the invariant checks should not have to be run explicitly (step 2 and step 3). The reason that the JML translator generates these checks anyway has to do with the strategies JML tools use to check invariants.

Tools such as JML runtime checkers may assume no problems with ownership aliasing to avoid having to keep track of what objects and types are in visible states. Although this reduces the overhead of checking invariants, it also means that some invariant violations might go unnoticed. Alternatively, tools can check every applicable invariant for classes and objects in visible states but this adds a significant overhead to the program execution.

Since aliasing can never occur in VDM-SL, it becomes simpler to keep track of what objects are in a visible state in the generated code and thus generate JML checks that explicitly trigger the invariants checks. This has the advantage that invariant violations do not go unnoticed even though a JML tool adopts a more practical approach to checking invariants.

For the example in Listing 41, the important thing is to ensure that the violation of the invariant of `R1` gets reported after executing the state update. This is done by asserting the entire chain of state designators. The JML generator is able to generate these checks since it keeps track of state designators of records that may have been affected by updates to other state designators.

17. Checking transitive dependencies

Let d_n be a state designator of a record in the generated code, which has been updated non-atomically, and let $d_k, \ldots, d_1$, for $k = n-1$, be state designators that were affected by the update to d_n. Further assume that d_i may be of one of m_i record types $D_{i1}, \ldots, D_{im_i}$. Immediately after executing the update to d_n the state of d_n becomes visible. To ensure that the invariant check gets triggered for all affected state designators, execute the following sequence of assertions:

//@ **assert** d_k **instance of** D_{k1} ==>
  \invariant_for((D_{k1}) d_k);
...
//@ **assert** d_k **instance of** D_{km_k} ==>
  \invariant_for((D_{km_k}) d_k);
....
//@ **assert** d_1 **instance of** D_{11} ==>
  \invariant_for((D_{11}) d_1);
...
//@ **assert** d_1 **instance of** D_{1m_1} ==>
  \invariant_for((D_{1m_1}) d_1);

Note that the code in Listing 41 omits the **`instance of`** checks, proposed by rule 17, since the types of the affected state designators can be determined statically.

Regarding rule 9, similar issues with transitive dependencies may occur in the generated code when dealing with atomic execution. Recall that before executing the code generated atomic statement block, invariant checking is disabled. Once the atomic execution has completed, invariant checking is re-enabled, and therefore rule 9 must also take into account all the state designators that were affected by the atomic execution.

7.2 Recursive types

It is possible to formulate recursive types for which the generated JML checks can only perform limited type checking. To demonstrate this, consider the recursive VDM type definition in Listing 42. For this example, `S` represents an infinite number of types including **`nat1`** as well as all possible dimensions of sequences that store elements of type **`nat1`**, i.e. **`seq of nat1`**, **`seq of seq of nat1`** and so on.

```
types
S = nat1 | seq of S;
```

Listing 42 Example of recursive type definition in VDM.

The issue with this kind of type definition is that `Is(v,S)` in theory becomes an expression of infinite length. The JML code generator stops generating type checks whenever it encounters type cycles. For the particular example in Listing 42 this means that a Java value or object reference `v` is only considered to respect `S` if `Utils.is_nat1(v)` holds.

This approach to checking types could be changed to also take the depth of the recursion `n` into account, i.e. use `Is(v,T,n)` to generate the type checks. The current approach used by the JML generator thus corresponds to generating checks using `Is(v,T,1)`. `Is(v,S,2)` then generates checks for types **`nat1`** and **`seq of nat`**, whereas `Is(v,S,3)` additionally generates a check for the type **`seq of seq of nat1`**.

8 Related work

In [28] Vilhena considers the possibilities for automatically converting between VDM++ and JML and the approach is demonstrated using a proof-of-concept implementation. That work considers a bi-directional mapping, whereas we only consider a one-way translation from executable VDM-SL to JML annotated Java. The bi-directional mapping proposed by Vilhena produces only the JML specification files and not the Java sources, which is an essential part of our work. The implementation of the bi-directional mapping was originally targeting the Overture tool, but it never reached maturity to be included in the release of the tool.

Rules for translating from a subset of VDM-SL to JML are proposed by Jin et al. in [11]. Their approach also considers implicit functional descriptions but it provides limited support for translation of record definitions and named invariant types. In the early phases of the software development process the authors propose to formulate requirements in natural language or using the Unified Modeling Language (UML) [26] and then formalise them in VDM-SL to eliminate ambiguity. Subsequently the authors manually apply their rules to the VDM-SL specification to produce an initial version of the software implementation. Their work does, however, not take generation of the bodies of functions and operations into account. Therefore, the authors only produce the method signatures for the Java methods when translating the functional descriptions of the VDM-SL model.

The translation rules proposed by Jin et al. have been implemented as an Eclipse plugin by Zhou et al. in [32]. The plugin takes a VDM-SL specification as input, which is type-checked using VDMTools [27], and outputs JML annotated Java classes that must be completed manually by the developer.

Code generation from other formal notations or modelling languages to JML annotated Java is also possible. As an example, Rivera et al. present the EventB2Java tool [25] – a code generator, which is capable of translating both abstract and refinement Event-B [1] models into JML annotated Java. EventB2Java has the advantage over other Event-B code generators that it does not require user intervention as part of the code generation process, which is similar to our approach.

In [20] Lensink et al. present a prototype code generator which translates a subset of the Prototype Verification System (PVS) [24] to an intermediate representation in Why [6] suitable for program verification. Subsequently the Why representation is translated to JML annotated Java. In their work the authors focus on translating executable PVS constructs, which is similar to what we do for VDM-SL. A key feature of their code generator is that it, in addition to specification code, also translates proven properties, which is outside the scope of our work.

Hubbers et. al propose AutoJML [10] – a tool for translating UML state diagrams into JML annotated Java Card code [31]. A state diagram describes a Java Card applet from which AutoJML produces Java skeleton code annotated with JML. In the generated code the different states are represented as constant values, and an additional Java field is used to represent the current state of the applet. A JML **`invariant`** is used to specify the valid state values for this field, and a JML **`constraint`** is used to describe the valid state transitions. This is comparable to the way we enable and disable invariant checking, which we do by toggling the `invChecksOn` **`ghost`** variable using **`set`** statements.

In [14] Klebanov proposes an approach similar to that of Hubbers et al. Instead of using UML state diagrams, Klebanov uses automata-based programming to describe the behaviour of a smart card application, which is generated to JML annotated Java Card code. Klebanov argues that use of automata-based programming over UML state diagrams is a better way to describe application-specific behaviour. A similar argument can be made for VDM-SL, which is suitable for capturing the dynamic aspects of a system.

9 Conclusion and future plans

In this paper we have demonstrated how VDM-SL models can be translated to JML annotated Java programs that can be checked for correctness using JML tools. The JML translator uses JML to represent the DbC elements of VDM-SL, and generates checks that help ensure the consistency of VDM-SL types across the translation.

The principles for pre- and post conditions in VDM-SL and JML are similar although there are subtle semantic differences between the two notations. These differences are mostly caused by the fact that JML is built on top of Java, where object types use reference semantics. VDM-SL, on the other hand, solely uses value types. Therefore, it is necessary to employ deep cloning principles when representing value types in JML annotated Java code.

Checking state and record invariants in the generated code is complicated due to two reasons: First, atomic execution in VDM requires a way to control when invariant checking must be done. We achieve this by using a **ghost** variable to indicate when invariant checking is enabled, and update it before entering and leaving the **atomic** statement. Secondly, we have demonstrated that transitive dependencies between records sometimes require extra JML checks to be generated to ensure that the invariant checks get triggered when they should.

The differences between the type systems of VDM-SL and Java further necessitate extra checks to be produced. These checks are needed to ensure that the generated code does not violate any of the constraints imposed by the types in the VDM-SL model. Overture performs these dynamic type checks internally, whereas they must be made explicit in Java.

In the future we plan to use this work in the context of test automation. In VDM it is possible to specify a trace definition in a way similar to that of a regular expression. This trace can then be expanded into a large collection of tests that can be executed against the model. This is a useful way to detect deficiencies in the model, such as missing pre conditions, post conditions and invariants [16].

We plan to code generate the trace expansion such that the generated tests can be executed against the code generated version of the model. The work presented in this paper can then be used to detect contract or type violations and give verdicts to the code generated trace tests. We believe that this will be particularly advantageous for execution of large collections of tests. We expect this approach to significantly increase execution speed for test cases and also allow more tests to be executed. In addition we plan to look into JML generation for other VDM dialects such as VDM++ [8]. However, since VDM++ is object-oriented and supports concurrency, we envisage that this will give rise to a completely new set of challenges not addressed by the work in this paper.

We hope that our work will serve as inspiration for other researchers who seek to bridge the gap between other formal model-oriented specification notations and implementation technologies that support the DbC approach or differ in terms of the types they support. We believe that the rules proposed in this paper can be useful for others who want to translate between specification languages such as ASM, B and Z and implementation technologies such as Spec#, Sparc-Ada and Eiffel.

Acknowledgements The authors would like to thank Victor Bandur and Nick Battle for their valuable feedback on earlier versions of this paper.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
2. Andrews, D., Bruun, H., Damm, F., Dawes, J., Hansen, B., Larsen, P., Parkin, G., Plat, N., Totenel, H.: A Formal Definition of VDM-SL. Tech. Rep. 1998/9, Leicester University (1998)
3. Bjørner, D., Jones, C. (eds.): The Vienna Development Method: The Meta-Language, *Lecture Notes in Computer Science*, vol. 61. Springer-Verlag (1978)
4. Burdy, L., Cheon, Y., Cok, D., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML Tools and Applications. Intl. Journal of Software Tools for Technology Transfer **7**, 212–232 (2005)
5. Cok, D.: OpenJML: JML for Java 7 by Extending OpenJDK. In: M. Bobaru, K. Havelund, G. Holzmann, R. Joshi (eds.) NASA Formal Methods, *Lecture Notes in Computer Science*, vol. 6617, pp. 472–479. Springer Berlin Heidelberg (2011). DOI 10.1007/978-3-642-20398-5_35. URL http://dx.doi.org/10.1007/978-3-642-20398-5_35
6. Filliâtre, J.C.: Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud (2003). URL http://www.lri.fr/~filliatr/ftp/publis/why-tool.ps.gz
7. Fitzgerald, J., Larsen, P.G.: Modelling Systems – Practical Tools and Techniques in Software Development, Second edn. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK (2009). DOI 10.1017/CBO9780511626975. ISBN 0-521-62348-0
8. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object–oriented Systems. Springer, New York (2005). DOI 10.1007/b138800. URL http://overturetool.org/publications/books/vdoos/
9. Fitzgerald, J.S., Larsen, P.G., Verhoef, M.: Vienna Development Method. Wiley Encyclopedia of Computer Science and Engineering (2008). Edited by Benjamin Wah, John Wiley & Sons, Inc.
10. Hubbers, E., Oostdijk, M.: Generating JML specifications from UML state diagrams. In: In Forum on Specification and Design Languages FDL'03, pp. 263–273 (2003)
11. Jin, D., Yang, Z.: Strategies of Modeling from VDM-SL to JML. Advanced Language Processing and Web Information Technology, International Conference on **0**, 320–323 (2008). DOI http://doi.ieeecomputersociety.org/10.1109/ALPIT.2008.25
12. Jones, C.B.: Software Development A Rigorous Approach. Prentice-Hall International, Englewood Cliffs, New Jersey (1980)
13. Jørgensen, P.W.V., Couto, L.D., Larsen, M.: A Code Generation Platform for VDM. In: The Overture 2014 workshop (2014)
14. Klebanov, A.: Automata-Based Programming Technology Extension for Generation of JML Annotated Java Card Code. pp. 41–44. Proc. of the Spring/Summer Young Researchers' Colloquium on Software Engineering (2008)
15. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The Overture Initiative – Integrating Tools for VDM. SIGSOFT Softw. Eng. Notes **35**(1), 1–6 (2010). DOI 10.1145/1668862.1668864. URL http://doi.acm.org/10.1145/1668862.1668864
16. Larsen, P.G., Lausdahl, K., Battle, N.: Combinatorial Testing for VDM. In: Proceedings of the 2010 8th IEEE International Conference on Software Engineering and Formal Methods, SEFM '10, pp. 278–285. IEEE Computer Society, Washington, DC, USA (2010). DOI 10.1109/SEFM.2010.32. URL http://dx.doi.org/10.1109/SEFM.2010.32. ISBN 978-0-7695-4153-2
17. Larsen, P.G., Lausdahl, K., Battle, N.: The VDM-10 Language Manual. Tech. Rep. TR-2010-06, The Overture Open Source Initiative (2010)
18. Lausdahl, K., Larsen, P.G., Battle, N.: A Deterministic Interpreter Simulating A Distributed real time system using VDM. In: S. Qin,

Z. Qiu (eds.) Proceedings of the 13th international conference on Formal methods and software engineering, *Lecture Notes in Computer Science*, vol. 6991, pp. 179–194. Springer-Verlag, Berlin, Heidelberg (2011). DOI 10.1007/978-3-642-24559-6_14. ISBN 978-3-642-24558-9

19. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Kiniry, J.: JML Reference Manual, revision 2344 edn. (2013)
20. Lensink, L., Smetsers, S., van Eekelen, M.: Generating Verifiable Java Code from Verified PVS Specifications. In: A. Goodloe, S. Person (eds.) NASA Formal Methods, *Lecture Notes in Computer Science*, vol. 7226, pp. 310–325. Springer Berlin Heidelberg (2012). DOI 10.1007/978-3-642-28891-3_30. URL `http://dx.doi.org/10.1007/978-3-642-28891-3_30`
21. Meyer, B.: Object-oriented Software Construction. Prentice-Hall International (1988)
22. Müller, P., Poetzsch-Heffter, A., Leavens, G.T.: Modular Invariants for Layered Object Structures. Sci. Comput. Program. **62**(3), 253–286 (2006). URL `http://dx.doi.org/10.1016/j.scico.2006.03.001`
23. The Overture tool website. `http://overturetool.org/` (2015)
24. Owre, S., Rushby, J.M., Shankar, N.: PVS: A Prototype Verification System. In: D. Kapur (ed.) 11th International Conference on Automated Deduction (CADE), *Lecture Notes in Artificial Intelligence*, vol. 607, pp. 748–752. Springer-Verlag, Saratoga, NY (1992)
25. Rivera, V., Cataño, N., Wahls, T., Rueda, C.: Code generation for Event-B. International Journal on Software Tools for Technology Transfer pp. 1–22 (2015)
26. Rumbaugh, J., Jacobson, I., Booch, G.: Unified Modeling Language Reference Manual, The (2nd Edition). Pearson Higher Education (2004)
27. SCSK: The VDMTools website. `http://www.vdmtools.jp/en/` (2015)
28. Vilhena, C.: Connecting between VDM++ and JML. Master's thesis, Minho University with exchange to Engineering College of Arhus (2008)
29. Wing, J.M.: Writing Larch interface language specifications. ACM Trans. Progr. Lang. Syst. **9**(1), 1–24 (1987). URL `http://doi.acm.org/10.1145/9758.10500`
30. Woodcock, J., Davies, J.: Using Z – Specification, Refinement, and Proof. Prentice Hall International Series in Computer Science (1996)
31. Zhen, Z.: Java Card Technology for Smart Cards. Prentice-Hall, Boston (2000)
32. Zhou, J., Jin, D.: Research on modeling from VDM-SL to JML for systematic software development. In: Control and Decision Conference (CCDC), 2010 Chinese, pp. 2312–2317. IEEE, Xuzhou (2010)

14

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

This paper has been submitted to a peer-reviewed conference.

[P80] Peter W. V. Tran-Jørgensen, Peter Gorm Larsen and Nick Battle. *Using JML-based Code Generation to Enhance Test Automation for VDM Models*. Submitted to the 21st International Symposium on Formal Methods (FM), November, 2016.

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

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

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

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

1 Introduction

Inspired by TOBIAS [20, 21], the Vienna Development Method (VDM) [8, 12] has been enhanced with combinatorial testing – a test automation feature that uses a pattern-based notation to describe and execute test sets. In VDM, combinatorial testing is used to validate data and behaviour against invariants and pre- and post conditions [16, 17]. The tests are generated from a pattern, referred to as a *trace*, which describes a finite subset of all possible executions of a VDM model. Combinatorial testing for VDM is supported for an executable subset of VDM by the Overture [18] and the VDMJ [1] interpreters. However, execution of large combinatorial test sets can be slow and demanding in terms of memory resources. To improve on this situation, this paper presents a technique that allows traces to be executed as compiled code.

Using Overture's [15, 5, 23] VDM-to-Java code generator [13, 26] a VDM-SL model can be translated to a Java program, where the contract-based VDM elements are described using Java Modeling Language (JML) constraints [19]. In this paper we extend this code generator to support traces.[3] When a code generated trace is executed, using a JML tool, the trace tests are expanded and run against the code generated version of the VDM specification. The verdict of each test is determined by checking the generated code against the JML constraints. There are two benefits to this approach. First, it has

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

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

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

Combinatorial testing has been researched for several years [22] with most of the work centred around tools that support combinatorial testing for different programming languages [27]. At the level of specification languages it is worth comparing our work to [7]. In their work, Dick et al. use a finite-state automaton approach to support test sequencing of implicit-style VDM models. However, since their technique uses symbolic values, one needs to provide the means to select concrete ones.

Similar to VDM, TOBIAS supports test case generation from a pattern that describes a test set. The test cases generated by TOBIAS, can be output as sequences of VDM operation calls or as JUnit [14] test cases. In particular, the latter can be used to test a Java implementation against a JML specification. Although our work currently only supports Java and JML, it also enables code generation of the VDM specification in addition to the tests. Our technique may, however, be adopted to use other Design-by-Contract (DbC) implementation technologies (section 6).

The test automation technique, presented in this paper, is also comparable to what can be achieved using SAT solvers [2] since both techniques conduct an analysis for a finite collection of cases. However, while SAT solvers are limited by the number of values for each domain, our approach is guided by traces for the finite number of combinations to be analysed. Similar to our approach the Spin model checker translates a Promela model into an optimised C-program that performs exhaustive exploration of the state space [10]. In order to avoid the state explosion problem that is inherent to model checking, one needs to limit the size of the state space subject to exploration. In VDM the state space is limited by the desired paths undertaken, which is expressed using a trace.

The structure of the paper is as follows: After this introduction, combinatorial testing for VDM is described in section 2. Next, our approach to code generating traces is presented in section 3. Afterwards, the performance of code generated traces is studied using an industrial case study in section 4. This is followed by a discussion of the performance results in section 5. Finally, this paper concludes and outlines future work in section 6.

2 Background

2.1 VDM

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

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

As an informative annex to the VDM-SL ISO standard [11], a module-based extension has been added to the language. This extension enables structuring of a VDM-SL model into a collection of modules with capabilities to import and export definitions to/from other modules. A module may define a single state component which can be constrained by an invariant.

2.2 Combinatorial testing of VDM models

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

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

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

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

Listing 1.1. Example of trace in VDM.

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

2.3 Code generation for VDM-SL

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

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

Overture's Java code generator can translate pre- and post conditions and invariants of VDM-SL specifications to JML annotations that are added to the generated Java code [26]. JML is a DbC specification language, used for specification of Java classes and interfaces. To demonstrate how the code generator uses JML, consider a code generated method `f`, with input parameters $\mathtt{i}_1$`,...,`$\mathtt{i}_n$, derived from a user-defined VDM-SL function. Then `f` has a code generated **`static`** pre condition method `pre_f` with the same parameters as `f`. To indicate that `pre_f` has no write-effects it is marked with the JML **`pure`** modifier. In addition, `f` is annotated with the **`requires`** `pre_f(`$\mathtt{i}_1$`,...,`$\mathtt{i}_n$`)` annotation to make `pre_f` a pre condition of `f`. The generated Java program can be validated against the JML annotations in order to ensure that the generated code meets the properties described by the contracts of the VDM specification.

The Java code generator also produces JML checks to ensure the consistency of VDM types when they are translated to Java. This is achieved using a function `Is(v,T)` which checks that a Java value or object reference `v` is consistent with the VDM type `T` that it originates from. For example, consider a Java value or object reference `v` which represents a VDM identifier that is either a natural number or a boolean value, i.e. it is of type **`bool|nat`**. In the generated Java code `Is(v,`**`bool`**`|`**`nat`**`)` is a JML predicate that is used to test whether `v` is either **`true`**, **`false`** or some positive integer. For this simple example the JML annotation checks that `Utils.is_nat(v) || Utils.is_bool(v)` is true. More complex types, such as user-defined types constrained by invariants or collection-based types, generate more complicated JML checks. The full definition of `Is(v,T)` and a description of how this function is used by the Java code generator, is described in detail in [26]. Some of the checks generated by `Is(v,T)` uses functionality of the Java code generator's runtime library, including a small extension of it called `V2J`. For example, `V2J` has functionality to check if a Java object represents an injective mapping or a non-empty sequence, which is used to check type constraints in the generated code.

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

3 Code Generating Traces

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

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

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

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

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

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

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

Expansion and execution of the trace is handled entirely by the runtime library, and performed using the `ExecTests` method in the `TraceNode` class. The execution of the trace tests is shown using a UML sequence diagram in fig. 2. In this figure `ast` represents the trace AST, `module` is the code generated version of the module enclosing the trace, `testAcc` is used to record the test results, and finally `store` is used to manage system states between the different test runs. As described in section 2, the system is reset between each test, i.e. the tests are executed independently. The code generator uses the `store` to achieve this when executing the code generated version of the trace. The test accumulator (`testAcc` in fig. 2) receives information about each test that has been executed via the `registerTest` method. This method call has been omitted from fig. 2 to keep the figure simple. A test accumulator is implemented as a strategy [9], i.e. one test accumulator may print the test results directly to the console, another test accumulator may write the results to the file system and so on.

As a first step of the test expansion and execution, the module class enclosing the code generated trace, is registered in the `store`. In case there are other module classes they are also registered in the `store`, since they might also have their state changed during test execution. Subsequently the tests are derived by invoking the `getTests` method on the root of the `ast`, which returns a `TestSequence` that contains the

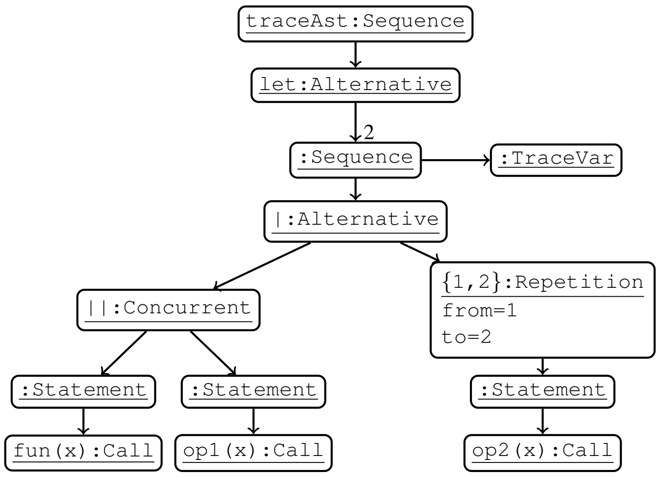


Fig. 1. Runtime representation of a code generated trace, shown using a UML object diagram.

generated tests. Next, each test is executed and the `store` is reset to restore the system state. This process continues until there are no more tests to be executed.

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

```
Call callStm_3 = new Call() {
   public Boolean isTypeCorrect() {
     try {
       //@ assert Utils.is_nat(x);
     } catch (AssertionError e) {
       return false;
     }
     return true;
```

```
  }
  public Boolean meetsPreCond() {
    return pre_op2(x);
  }
  public Object execute() {
    return op2(x);
  }
  public String toString() {
    return "op2(" + Utils.toString(x) + ")";
  }
};
```

Listing 1.4. Implementation of a `Call` statement.

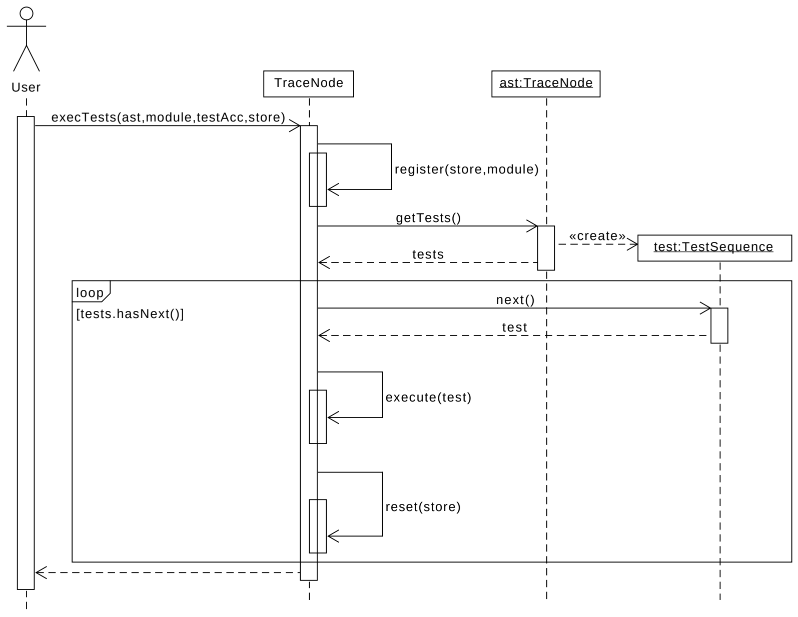


Fig. 2. Execution of a code generated trace, shown using a UML sequence diagram.

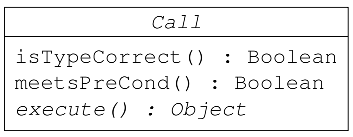


Fig. 3. The interface of the `Call` statement node.

For a `Call` statement to be type correct the input arguments $\texttt{a}_1, \ldots, \texttt{a}_n$ must match the types of the formal parameters, $\texttt{T}_1, \ldots, \texttt{T}_n$, of the function or operation that the `Call` statement originates from. Using the function `Is(v,T)`, described in section 2, we require that $\texttt{Is(a}_1\texttt{,T}_1\texttt{)}$ `&& ... &&` $\texttt{Is(a}_n\texttt{,T}_n\texttt{)}$ holds in order for the `Call` statement to be considered type correct. This check is code generated to a JML assertion, which can be configured to produce an `AssertionError` that signals that the `Call` statement is not type correct for the given arguments. As indicated by the generated JML check in listing 1.4, it is assumed that the formal parameter of `op2` is of type **`nat`**.

If a test is considered type correct the runtime library will check that the pre condition of the `Call` statement is met, which is checked using the `meetsPreCond` method. The `meetsPreCond` method returns **`true`** by default, which corresponds to the situation where no pre condition is defined. If either the `isTypeCorrect` or `meetsPreCond` method yield false the test is `INCONCLUSIVE`. Otherwise the runtime library proceeds by calling the `execute` method, which evaluates the `Call` statement and returns the result to the runtime library. This method is abstract and implemented by the code generator when the trace is code generated (see listing 1.4).

4 Case study

4.1 FAD codes

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

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

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

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

Listing 1.5. Excerpts from the FAD code VDM model

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

One could write a trace which applied every possible injective map to the entire set of FAD codes to test that every case produced a permutation. However, with one million FAD codes per map, that would be intractable. This illustrates the point that although trace expansion is useful, combinatorial explosions can easily limit the size of the traces that are possible to execute. It also shows that performance gains achieved by translation of the VDM source into another language may significantly increase the scope of combinatorial testing that is possible.

4.2 Performance results

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

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

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

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

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

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

5 Discussion

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

Overture did not manage to run the one million tests generated for six digit FAD codes because the tool ran out of memory. The code generated trace, on the other hand, completed these tests in over $2,900$ s, or 49.22 minutes. What is surprising about the results obtained using these two tools is that Overture expands and executes the tests significantly faster than the code generated trace, for FAD code sizes smaller than six. Therefore, the only performance gain of the code generated traces is the reduction in memory needed to expand and execute the trace. This is surprising since traces that are

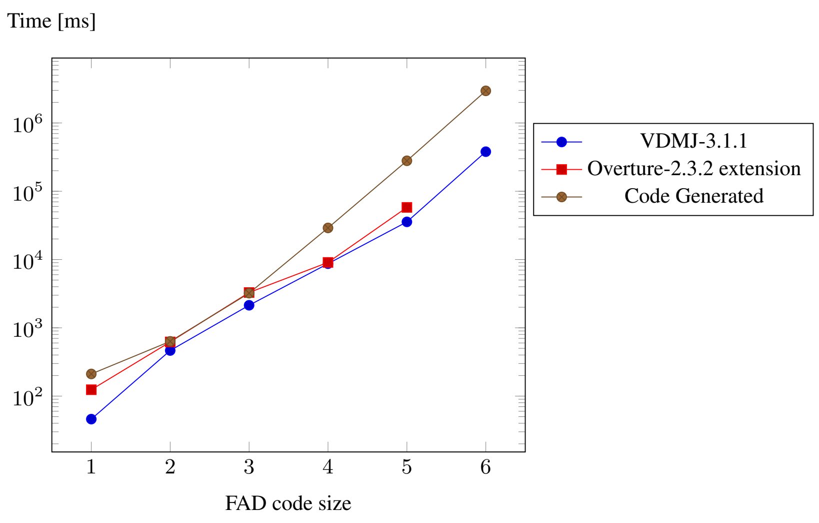


Fig. 4. The execution times in table 1 visualised using a logarithmic data plot.

run as compiled code are, by intuition, expected to run faster. Comparing the execution time of a code generated trace to that obtained using Overture provides a good indication of the performance that can be gained since these tools use the same algorithm to expand the trace.

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

To study the performance overhead posed by OpenJML, we first tried to compile and execute the code generated trace as a normal Java program – without using OpenJML. This is similar to running the code generated trace without checking the generated JML annotations. When this is done, even with the poor expansion algorithm, the one million tests are expanded and executed in 33.94 seconds. If the expansion algorithm is changed to that used by VDMJ, this will most likely be significantly lower. The reason for this is that VDMJ seems to scale better than Overture for larger test sets, as indicated by the execution times in table 1.

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

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

6 Conclusion and future plans

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

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

Looking forward, we plan to investigate other technologies that can support our work and help us achieve better performance results. One way to ensure that the contracts and type constraints, as specified in VDM, hold across the translation, is by adding extra Java checks to the generated code – without using a particular DbC technology. Although this allows us to control exactly when these checks are triggered, the separation between specification and code becomes less clean. One DbC technology that is worth investigating, as an alternative to OpenJML, is the .NET-based technology, Microsoft Code Contracts [25, chapter 15]. Compared to JML, which uses a dedicated syntax for program specification, Code Contracts provides its features via libraries to support all languages within the .NET framework. JML and Code Contracts share many of the same concepts although their semantics sometimes differ. Changing our work to use another DbC technology therefore requires new rules for representing VDM constraints in the generated code. However, due to the similarities between Java and C#, we expect the approach used to code generate traces to be readily reusable. In a C++ context another DbC technology that is worth investigating is the Contract++ library [4], which has been accepted into Boost [24]. On a longer term, contracts may also be a native feature of C++17 [6].

References

1. Battle, N.: VDMJ website. `https://github.com/nickbattle/vdmj` (2016)
2. Brummayer, R., Lonsing, F., Biere, A.: Automated Testing and Debugging of SAT and QBF Solvers. In: Strichman, O., Szeider, S. (eds.) Theory and Applications of Satisfiability Testing – SAT 2010. pp. 44–57. Springer, Edinburgh, UK (Jul 2010)
3. Cok, D.: OpenJML: JML for Java 7 by Extending OpenJDK. In: Bobaru, M., Havelund, K., Holzmann, G., Joshi, R. (eds.) NASA Formal Methods, Lecture Notes in Computer Science, vol. 6617, pp. 472–479. Springer Berlin Heidelberg (2011), `http://dx.doi.org/10.1007/978-3-642-20398-5_35`
4. Contract++ website. `https://sourceforge.net/projects/contractpp/` (2016)
5. Couto, L.D., Larsen, P.G., Hasanagić, M., Kanakis, G., Lausdahl, K., Tran-Jørgensen, P.W.V.: Towards Enabling Overture as a Platform for Formal Notation IDEs. In: Proceedings of the 2nd Workshop on Formal-IDE (F-IDE) (Jun 2015)
6. Thoughts on C++17. `http://www.infoq.com/news/2015/04/stroustrup-cpp17-interview` (2016)
7. Dick, J., Faivre, A.: Automating the Generation and Sequencing of Test Cases from Model-Based Specifications. In: Woodcock, J.C.P., Larsen, P.G. (eds.) FME'93: Industrial-Strength Formal Methods. pp. 268–284. Formal Methods Europe, Springer-Verlag (Apr 1993)
8. Fitzgerald, J., Larsen, P.G.: Modelling Systems – Practical Tools and Techniques in Software Development. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, Second edn. (2009), ISBN 0-521-62348-0
9. Gamma, E., Helm, R., Johnson, R., Vlissides, R.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series, Addison-Wesley Publishing Company (1995)
10. Holzmann, G.J.: The model checker SPIN. IEEE Transactions on Software Engineering 23(5) (May 1997)
11. ISO: Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language (Dec 1996)
12. Jones, C.B.: Scientific Decisions which Characterize VDM. In: Wing, J.M., Woodcock, J.C.P., Davies, J. (eds.) FM'99 - Formal Methods. pp. 28–47. Springer-Verlag (1999)
13. Jørgensen, P.W.V., Larsen, M., Couto, L.D.: A Code Generation Platform for VDM. In: Battle, N., Fitzgerald, J. (eds.) Proceedings of the 12th Overture Workshop. School of Computing Science, Newcastle University, UK, Technical Report CS-TR-1446 (Jan 2015)
14. The JUnit website. `http://www.junit.org` (2016)
15. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The Overture Initiative – Integrating Tools for VDM. SIGSOFT Softw. Eng. Notes 35(1), 1–6 (Jan 2010), `http://doi.acm.org/10.1145/1668862.1668864`
16. Larsen, P.G., Lausdahl, K., Battle, N.: Combinatorial Testing for VDM. In: Proceedings of the 2010 8th IEEE International Conference on Software Engineering and Formal Methods. pp. 278–285. SEFM '10, IEEE Computer Society, Washington, DC, USA (Sep 2010), `http://dx.doi.org/10.1109/SEFM.2010.32`, ISBN 978-0-7695-4153-2
17. Larsen, P.G., Lausdahl, K., Battle, N., Fitzgerald, J., Wolff, S., Sahara, S., Verhoef, M., Tran-Jørgensen, P.W.V., Oda, T.: VDM-10 Language Manual. Tech. Rep. TR-001, The Overture Initiative, www.overturetool.org (Apr 2013)

18. Lausdahl, K., Larsen, P.G., Battle, N.: A Deterministic Interpreter Simulating A Distributed real time system using VDM. In: Qin, S., Qiu, Z. (eds.) Proceedings of the 13th international conference on Formal methods and software engineering. Lecture Notes in Computer Science, vol. 6991, pp. 179–194. Springer-Verlag, Berlin, Heidelberg (Oct 2011), `http://dl.acm.org/citation.cfm?id=2075089.2075107`, ISBN 978-3-642-24558-9
19. Leavens, G.T., Cheon, Y.: Design by Contract with JML (2005), `ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf`, draft, available from jmlspecs.org.
20. Ledru, Y., du Bousquet, L., Maury, O., Bontron, P.: Filtering TOBIAS Combinatorial Test Suites. In: Wermelinger, M., Margaria-Steffen, T. (eds.) FASE 2004. pp. 281–294. LNCS 2984, Springer-Verlag Berlin Heidelberg (2004)
21. Ledru, Y., Dadeau, F., du Bousquet, L., Ville, S., Rose, E.: Mastering Combinatorial Explosion with the Tobias-2 Test Generator. In: ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. pp. 535–536. ACM, New York, NY, USA (2007)
22. Nie, C., Leung, H.: A Survey of Combinatorial Testing. ACM Comput. Surv. 43(2) (Feb 2011), `http://doi.acm.org/10.1145/1883612.1883618`
23. The Overture tool website. `http://www.overturetool.org/` (2016)
24. Schling, B.: The Boost C++ Libraries. XML Press (2011)
25. Skeet, J.: C# in Depth, Second Edition. Manning Publications (2010)
26. Tran-Jørgensen, P.W.V., Larsen, P.G., Leavens, G.T.: Automated translation of VDM to JML annotated Java (Jan 2016, submitted to the International Journal on Software Tools for Technology Transfer (STTT))
27. Yu, L., Lei, Y., Kacker, R.N., Kuhn, D.R.: ACTS: A combinatorial test generation tool. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation. pp. 370–375 (Mar 2013)

Bibliography

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

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

[3] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer*, 12(6):447–466, 2010.

[4] The OSGi Alliance. About the OSGi Service Platform. Technical Report Revision 4.1, June 2007.

[5] D. J. Andrews, H. Bruun, F. Damm, J. Dawes, B. S. Hansen, P. G. Larsen, G. Parkin, N. Plat, and H. Totenel. A Formal Definition of VDM-SL. Technical Report 1998/9, Leicester University, June 1998.

[6] Michele Banci, Alessandro Fantechi, Stefania Gnesi, and Giovanni Lombardi. *Design for Dependable Systems: 13th International SDL Forum Paris, France, September 18-21, 2007 Proceedings*, chapter Model Driven Development and Code Generation: An Automotive Case Study, pages 19–34. Springer Berlin Heidelberg, 2007.

[7] Nick Battle. VDMJ website. `https://github.com/nickbattle/vdmj`, 2016.

[8] Nick Battle, Anne Haxthausen, Sako Hiroshi, Peter W. V. Jørgensen, Nico Plat, Shin Sahara, and Marcel Verhoef. The Overture Approach to VDM Language Evolution. In *Proceedings of the 11th Overture workshop*, August 2013.

[9] Juan C. Bicarregui, John S. Fitzgerald, Peter A. 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] DD Bochtis and SG Vougioukas. Minimising the Non-working Distance Travelled by Machines Operating in a Headland Field Pattern. *Biosystems Engineering*, 101(1):1–12, 2008.

[12] Richard Bonichon, David Déharbe, Thierry Lecomte, and Valério Medeiros. *Formal Methods: Foundations and Applications: 17th Brazilian Symposium, SBMF 2014, Maceió, AL, Brazil, September 29–October 1, 2014. Proceedings*. Springer International Publishing, Cham, 2015.

[13] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML Tools and Applications. *Intl. Journal on Software Tools for Technology Transfer*, 7:212–232, 2005.

[14] Clearsy. Atelier B website. `http://www.atelierb.eu/en/`, 2016.

[15] David R. Cok. OpenJML: JML for Java 7 by Extending OpenJDK. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 472–479. Springer Berlin Heidelberg, 2011.

[16] Joey W. Coleman, Anders Kaels Malmos, Peter Gorm Larsen, Jan Peleska, Ralph Hains, Zoe Andrews, Richard Payne, Simon Foster, Alvaro Miyazawa, Cristiano Bertolini, and André Didier. COMPASS Tool Vision for a System of Systems Collaborative Development Environment. In *Proceedings of the 7th International Conference on System of System Engineering, IEEE SoSE 2012*, pages 451–456, July 2012.

[17] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[P18] Luís Diogo Couto, Peter Gorm Larsen, Miran Hasanagić, Georgios Kanakis, Kenneth Lausdahl, and Peter W. V. Tran-Jørgensen. Towards Enabling Overture as a Platform for Formal Notation IDEs. In *Proceedings of the 2nd Workshop on Formal-IDE (F-IDE)*, June 2015.

[19] Luís Diogo Couto and Richard Payne. The COMPASS Proof Obligation Generator: A test case of Overture Extensibility. In *Proceedings of the 11th Overture Workshop*, 2013.

[P20] Luís Diogo Couto and Peter W. V. Tran-Jørgensen. Extending the Overture code generator towards Isabelle syntax. In *Proceedings of the 13th Overture Workshop*, June 2015.

[P21] Luís Diogo Couto and Peter W. V. Tran-Jørgensen. Integrating Real System Components in Model-Based Development. July 2016. Draft paper planned to be submitted for publication (venue to be found).

[P22] Luís Diogo Couto, Peter W. V. Tran-Jørgensen, Joey W. Coleman, and Kenneth Lausdahl. Migrating to an Extensible Architecture for Abstract Syntax Trees. In *Proceedings of the 12th Working IEEE / IFIP Conference on Software Architecture*, May 2015.

[23] Luís Diogo Couto, Peter W. V. Tran-Jørgensen, and Gareth T. C. Edwards. Combining Harvesting Operations Optimisation using Strategy-based Simulation. July 2016. Accepted to appear at the 6th International Conference on Simulation and Modeling Methodologies, Technologies and Applications (Simultech) 2016.

[P24] Luís Diogo Couto, Peter W. V. Tran-Jørgensen, and Kenneth Lausdahl. Principles for Reuse in Formal Language Tools. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC)*, April 2016.

[25] Delegate Tutorial. `https://github.com/ldcouto/delegate-tutorial`, 2016.

[26] The FMI development group. Functional Mock-up Interface for Model Exchange and Co-Simulation. `https://www.fmi-standard.org/downloads`, July 2014.

[27] Edsger W Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[28] Steffen Pham Diswal. Transcompilation of VDM-SL to C#. Master's thesis, Aarhus University, Department of Computer Science, June 2016.

[29] Paul M Duvall, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk.* Pearson Education, 2007.

[30] Modelon, Dymola website. `http://www.modelon.com/products/dymola/`, 2016.

[31] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2004.

[32] J. S. Fitzgerald, P. G. Larsen, and M. Verhoef. Vienna Development Method. *Wiley Encyclopedia of Computer Science and Engineering*, 2008.

[33] John Fitzgerald, Carl Gamble, Peter Gorm Larsen, Kenneth Pierce, and Jim Woodcock. Cyber-Physical Systems design: Formal Foundations, Methods and Integrated Tool Chains. In *FormaliSE: FME Workshop on Formal Methods in Software Engineering*, Florence, Italy, May 2015. ICSE 2015.

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

[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 Marcel Verhoef, editors. *Collaborative Design for Embedded Systems – Co-modelling and Co-simulation*. Springer, 2014.

[37] Simon Foster and Richard J. Payne. Theorem Proving Support - Developers Manual. Technical report, COMPASS Deliverable, D33.2b, September 2013.

[38] Apache Software Foundation. The Apache Velocity website. `http://velocity.apache.org`, 2016.

[39] E. Gamma, R. Helm, R. Johnson, and R. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, 1995.

[40] William Grosso. *Java RMI*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 2001.

[41] Miran Hasanagić, Peter Gorm Larsen, and Peter W. V. Tran-Jørgensen. Generating Java RMI for the distributed aspects of VDM-RT models. In *Proceedings of the 13th Overture Workshop*, June 2015.

[42] Miran Hasanagić, Peter W. V. Tran-Jørgensen, Kenneth Lausdahl, and Peter Gorm Larsen. Formalising and Validating the Interface Description in the FMI standard. May 2016. Submitted to the 21st International Symposium on Formal Methods (FM 2016).

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

[44] Peter Holst and Nikolas Bram. Code Generation from VDM++ to TypeScript. Master's thesis, Aarhus University, Department of Engineering, June 2016.

[45] Joan K. Hughes. *PL/1 Structured Programming*. Wiley, 1986.

[46] José Antonio Esparza Isasa, Peter W. V. Jørgensen, and Claus Ballegaard. Modelling Energy Consumption in Embedded Systems with VDM-RT. In *Proceedings of the 4th International ABZ conference*, July 2014.

[47] José Antonio Esparza Isasa, Peter W. V. Jørgensen, and Peter Gorm Larsen. Hardware In the Loop for VDM-Real Time Modelling of Embedded Systems. In *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development (Modelsward)*, January 2014.

[48] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley. The Java Language Specification, Java SE 7 Edition. `https://docs.oracle.com/javase/specs/jls/se7/jls7.pdf`, February 2013.

[49] Michael Jastram and Prof Michael Butler. *Rodin User's Handbook: Covers Rodin V.2.8*. CreateSpace Independent Publishing Platform, USA, 2014.

[50] Jenkins integration server website. `https://jenkins.io`, 2016.

[51] Martin F Jensen, Dionysis Bochtis, and Claus G Sørensen. Coverage planning for capacitated field operations, part II: Optimisation. *Biosystems Engineering*, 139:149–164, 2015.

[52] JGraphT. A free Java Graph Library. `http://jgrapht.org`, 2016.

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

[54] 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.

[P55] Peter W. V. Jørgensen, Luís D. Couto, and Morten Larsen. A Code Generation Platform for VDM. In *Proceedings of the 12th Overture workshop*, June 2014.

[56] Peter W. V. Jørgensen and Peter Gorm Larsen. Towards an Overture Code Generator. In *Proceedings of the 11th Overture workshop*, August 2013.

[57] Peter W. V. Jørgensen, Kenneth Lausdahl, and Peter Gorm Larsen. An Architectural Evolution of the Overture Tool. In *Proceedings of the 11th Overture workshop*, August 2013.

[58] The JUnit website. `http://www.junit.org`, 2016.

[59] Georgios Kanakis, Peter Gorm Larsen, and Peter W. V. Tran-Jørgensen. Code Generation of VDM++ Concurrency. In *Proceedings of the 13th Overture Workshop*, June 2015.

[60] J. C. Knight, K. S. Hanks, and S. R. Travis. Tool Support for Production Use of Formal Techniques. In *Proceedings of the 12th International Symposium on Software Reliability Engineering*, pages 242–251, November 2001.

[61] Morten Larsen, Peter W. V. Jørgensen, and Peter Gorm Larsen. Improving Time Estimates in VDM-RT Models. In *Proceedings of the 13th Overture Workshop*, June 2015.

[62] 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.

[63] 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.

[64] Peter Gorm Larsen, Kenneth Lausdahl, Nick Battle, John Fitzgerald, Sune Wolff, Shin Sahara, Marcel Verhoef, Peter W. V. Tran-Jørgensen, and Tomohiro Oda. VDM-10 Language Manual. Technical Report TR-001, The Overture Initiative, April 2013.

[65] Peter Gorm Larsen, Kenneth Lausdahl, Peter W. V. Tran-Jørgensen, Augusto Ribeiro, Sune Wolff, and Nick Battle. Overture VDM-10 Tool Support: User Guide. Technical Report TR-2010-02, The Overture Initiative, May 2010.

[66] 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, *Proceedings of the 13th international conference on 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.

[67] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, and Joseph Kiniry. *JML Reference Manual*, revision 2344 edition, May 2013.

[68] Maihemutijiang Maimaiti. Towards Development of Overture/VDM++ to Java Code Generator. Master's thesis, Aarhus University, Department of Computer Science, May 2011.

[69] MathWorks, MATLAB website. `http://www.mathworks.com`, 2016.

[70] The Apache Maven Project website. `https://maven.apache.org`, 2016.

[71] J. McCarthy. A Basis for a Mathematical Theory of Computation. In *Western Joint Computer Conference*, 1961.

[72] Bertrand Meyer. *Object-oriented Software Construction*. Prentice-Hall International, 1988.

[73] 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.

[74] Christian Nilsson. Heuristics for the Traveling Salesman Problem. Technical report, Linköping University, Sweden, 2003.

[75] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[76] Tomohiro Oda, Keijiro Araki, and Peter Gorm Larsen. VDMPad: a Lightweight IDE for Exploratory VDM-SL Specification. In *Proceedings of FormaliSE 2015*, pages 33–39, Florence, May 2015.

[77] Tomohiro Oda, Keijiro Araki, and Peter Gorm Larsen. Automated VDM-SL to Smalltalk Code Generators for Live Specification Environments. November 2016. Planned for submission to the 14th Overture Workshop.

[78] Oliver Oppitz. Concurrency Extensions for the VDM++ to Java Code Generator of the IFAD VDM++ Toolbox. Master's thesis, TU Graz, Austria, April 1999.

[79] Jon Skeet. *C# in Depth, Second Edition*. Manning Publications, 2010.

[P80] Peter W. V. Tran-Jørgensen, Peter Gorm Larsen, and Nick Battle. Using JML-based Code Generation to Enhance Test Automation for VDM Models. May 2016. Submitted to the 21st International Symposium on Formal Methods (FM 2016).

[P81] Peter W. V. Tran-Jørgensen, Peter Gorm Larsen, and Gary T. Leavens. Automated translation of VDM to JML annotated Java. January 2016. Submitted to the International Journal on Software Tools for Technology Transfer (STTT).

[82] Lars Vogel. *Eclipse Rich Client Platform*. Vogella. Lars Vogel, third edition, May 2015.

[83] Alan Wassyng and Mark Lawford. Software tools for safety-critical software development. *International Journal on Software Tools for Technology Transfer*, 8(4):337–354, 2005.

[84] J. Woodcock, A. Cavalcanti, J. Fitzgerald, P. Larsen, A. Miyazawa, and S. Perry. Features of CML: a Formal Modelling Language for Systems of Systems. In *Proceedings of the 7th International Conference on System of System Engineering*. IEEE, July 2012.

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

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