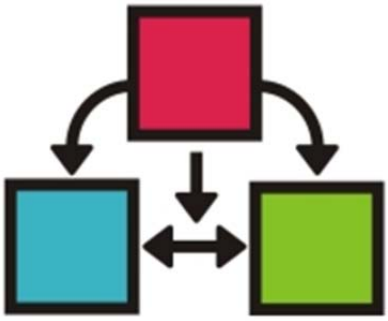
ENHANCING FORMAL MODELLING TOOL SUPPORT WITH **INCREASED AUTOMATION**

Electrical and Computer Engineering
Technical Report ECE-TR-4



AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

DATA SHEET

Title: Enhancing Formal Modelling Tool Support with Increased Automation
Subtitle: Electrical and Computer Engineering
Series title and no.: Technical report ECE-TR-4

Author: Kenneth Lausdahl
Department of Engineering – Electrical and Computer Engineering, Aarhus University

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

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

Year of publication: 2012 Pages: 39
Editing completed: October 2011

Abstract: Progress report for the qualification exam report for PhD Student Kenneth Lausdahl. Initial work on enhancing tool support for the formal method VDM and the concept of unifying a abstract syntax tree with the ability for isolated extensions is described. The tool support includes a connection to UML and a test automation principle based on traces written as a kind of regular expressions.

Keywords: VDM, Interpreter, Co-simulation

Supervisor: Peter Gorm Larsen
Financial support: European FP7 project, Design Support and Tooling for Embedded Control Software (DESTECS)

Please cite as: K. Lausdal, Enhancing Formal Modelling Tool Support with Increased Automation, 2012. Department of Engineering, Aarhus University. Denmark. pp. 39 Technical report ECE-TR-4

Front Image: Logos, DESTECS project + Overture Open Source Community

ISSN: 2245-2087

Reproduction permitted provided the source is explicitly acknowledged.

ENHANCING FORMAL MODELLING TOOL SUPPORT WITH INCREASED AUTOMATION

Kenneth Lausdahl — Aarhus University, Department of Engineering

Abstract

Progress report for the qualification exam report for PhD Student Kenneth Lausdahl. Initial work on enhancing tool support for the formal method VDM and the concept of unifying an abstract syntax tree with the ability for isolated extensions is described. The tool support includes a connection to UML and a test automation principle based on traces written as a kind of regular expressions.

Table of Contents

List of Figures

1.3 Aims of this PhD Project

The intended outcome of this PhD project is to improve automation in-relation to the kinds of analysis supported by the VDM dialects, so that it may be worthwhile to apply it in an industrial setting. In addition, this PhD project aims to show that it is possible to achieve an architecture for a common platform that can be a basis for different research groups to develop new features and/or new dialects of VDM.

1.4 Adjustment of Project Focus

Initially, the title of this PhD project was "Semantics Based Tool Analysis of Models Expressed in Multiple Notations". The intent behind this was to generalize co-simulation, carried out between a discrete-event world and a continuous-time world, so that semantics-based analysis could be carried out between models expressed using different paradigms (e.g. functional, logical). Unfortunately, integration of a discrete and a continuous world is so specific and centred around the progress of time that it cannot easily be generalised to cover the more standardised programming language paradigms. Thus, it was decided to change the focus and the aims of the PhD project as indicated above.

1.5 Structure of the Progress Report

Chapter 2 provides the background information necessary to understand the work carried out in this PhD project. This includes basic information about VDM, the history behind the associated tool support, and the necessary background for modelling of physical systems using differential equations simulated using a continuous-time simulator. Afterwards, Chapter 3 explains the history of the different ASTs present in the Overture platform and the work undertaken to unify these in order to achieve an architecture that is ideal for future development by different research teams. Then Chapter 4 describes the different kinds of automation tools that have been developed so far during this PhD work. This is based on different papers that have been published. This includes:

1. A description of the interpreter with its debugging capabilities [9].
2. Combinatorial testing automation based on regular expressions are presented [10].
3. Automatic exploration of alternative system architectures in a VDM-Real Time setting [11].
4. Automatic transformation between VDM++/UML class and sequence diagrams [12].

The chapter concludes with the work carried out on the co-simulation between VDM and 20-sim. Finally, Chapter 5 summarises the work undertaken so far in this PhD project and points at the future work that is planned for the remaining part of the PhD.

2

Background

This chapter will introduce the reader to the background of the project. Firstly, the formal method the Vienna Development Method (VDM) will be described giving an overview of which features the language has in the three dialects VDM-SL, VDM++ and VDM-RT. Secondly, the background behind tool support is described giving an overview of what work has been done for tool support for VDM in the past. Lastly, an introduction to continuous time modelling and co-simulation is presented explaining the basics about continuous time simulation with event detection.

2.1 The Vienna Development Method

The Vienna Development Method (VDM) [13, 14, 15] was originally developed at the IBM laboratories in Vienna in the 1970's and, as such, it is one of the longest established formal methods. The VDM Specification Language is a language with a formally defined syntax, and both static and dynamic semantics [16, 17]. Models in VDM are based on data type definitions built from simple abstract types using booleans, natural numbers, characters and type constructors for product, union, map, (finite) set and sequences. Type membership may be restricted by predicate invariants meaning that run-time type checking is also required from an interpreter perspective. Persistent state is defined by means of typed variables, again restricted by invariants. Operations that may modify the state can be defined implicitly, using standard pre- and post-condition predicates, or explicitly, using imperative statements. Such operations denote relations between inputs, outputs and states before and after execution; Note that such relations allows non-determinism behaviour. Functions are defined in a similar way to operations, but may not refer to state variables. Recursive functions can have a `measure` defined for them to ensure termination [18]. Arguments passed to functions and operations are always passed by value, apart from object references.

Three different dialects exist for VDM: The ISO standard VDM Specification Language (VDM-SL) [19], the object oriented extension VDM++ [20] and a further extension of that called VDM Real Time (VDM-RT) [21, 22]. All three dialects are supported by an open source tool called Overture [6] as well as by a commercial tool called VDMTools [23]. These tools, among other features, include standard parsers and type checkers that produce Abstract Syntax Trees (ASTs).

None of these dialects are generally executable since the languages permit the modeller to use type bindings with infinite domains, or implicitly defined functions and operations, but the dialects all have subsets that can be interpreted [24]. In addition, some commonly used implicit definitions

can be executed in principle [25]. A full description of the executable subset of the language can be found in [26].

VDM++ and VDM-RT allow concurrent *threads* to be defined. Such threads are synchronised using *permission predicates* that are associated with any operations that limits its allowed concurrent execution. Where pre-conditions for an operation describe the condition the caller must ensure before calling it, the permission predicate describes the condition that must be satisfied before the operation can be activated, and until that condition is satisfied the operation call is blocked. The permission predicates can refer to instance variables as well as *history counters* which indicate the number of times an operation has been requested, activated or completed for the current object. In VDM-RT, the concurrency modelling can be enhanced by deploying objects on different CPUs with buses connecting them. Operations called between CPUs can be asynchronous, so that the caller does not wait for the call to complete. In addition, threads can be declared as *periodic*, so that they run autonomously at regular intervals. For periodic threads it is also possible to express jitter, start time offset as well as the minimum arrival time between occurences of the operation used in a periodic thread[1].

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

The semantics of VDM-RT has been extended with the concept of discrete time, so that all computations a thread performs take time, including the transmission of messages over a bus. Time delays can be explicitly specified by special **`duration`** and **`cycles`** statements, allowing the modeller to explicitly state that a statement or block consumes a known amount of time. This can be specified as a number of nanoseconds or a number of CPU cycles of the CPU on which the statement is evaluated. All virtual resources are infinitely fast: calculation can be performed instantaneously consuming no time, though if an explicit duration statement is evaluated on a virtual CPU, the system time will be incremented by the duration.

The formal semantics of the kernel of VDM-RT is provided in [27] as an operational semantics. This uses an interleaving semantics without restricting non-deterministic choices; in particular, there are no requirements for specific scheduling policies. Thus, the semantics allows multiple different interleavings and as such the deterministic execution provided by the VDMJ [28] interpreter described here can be seen as one possible scheduling of a model containing non-determinism. All the other models are effectively ignored by the interpreter (see section 4.1).

2.2 VDM Tool Support

Early tools for VDM, such as Adelard's SpecBox [29] were largely confined to basic static checking and pretty-printing of specifications. Currently, only two tools for VDM are actively maintained, VDMTools [30, 23] and Overture [6].

[1] In the current version of the VDMJ interpreter the allowable jitter is a simple random distribution but it is expected that the user in the future will be able to specify a desired jitter distribution.

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

A newer tool that is under active development is the Overture tool. This is a community-based initiative that aims to develop a common open-source platform integrating a range of tools for constructing and analysing formal models of systems using VDM. The mission is to both provide an industrial-strength tool set for VDM and also to provide an environment that allows researchers and other stakeholders to experiment with modifications and extensions to the tools and language. As this PhD project is very much involved with the Overture initiative more information can be found in Section 3.1.

2.3 Continuous-time Modelling and Co-simulation

Continuous-time modelling provides a means of modelling a physical system. A continuous-time model changes over real time, not discrete time, thus differential equations are the optimal mathematical formalism to describe such continuous changes.The differential equations which embodies a model provides the values of these derivatives at any particular point in time. A computer can be used to move the state of the model forward in time, simulating the model within a time frame. The term used to describe the physical component which enables such simulations is called a numerical solver. The solver simulates a model by the use of an integration method to solve the ordinary differential equations that describe the dynamics of the system. Different kinds of integration methods exist which differ in the way they calculate the step size: Explicit Fixed Step, Explicit Variable Step, Implicit Fixed Step and Implicit Variable Step.

The fixed step methods always take a predefined step in time without taking into account the dynamics of the system. This is also known as sampling in the computer science domain. The accuracy of the method depends on the step size. The smaller the step size the more accurate the solution is, but the penalty is that it will take a longer time to simulate. Variable step methods use the dynamics of the system to determine how large the step size can be so that a certain accuracy is guaranteed. The advantage is that, usually, less steps have to be taken over the full time range. Only at points where the dynamics of the system dictate a certain accuracy, step sizes are decreased. The disadvantage is that it is not possible to know in advance what the next output time will be. Only a maximum step-size can be specified within the integration method, but this maximum is only used if the dynamics of the system allow it.

2.3.1 The Notation of Events

Events in a continuous-time model can either be a time event or a state event. A time event will be raised if the time crosses a threshold. A state event will be raised by a variable that crosses a threshold, either by a raising or falling crossing as shown in figure 2.1. Because different

integration methods exist it cannot be guaranteed that a step is taken in the integration to the point in time where the event occurs. To handle this a solver must be able to go back if an event occurs and retake the step if it did not match the point in time where the event occurred. The solution is then to decrease the step size to the closest numerical point in time matching the event occurrence.

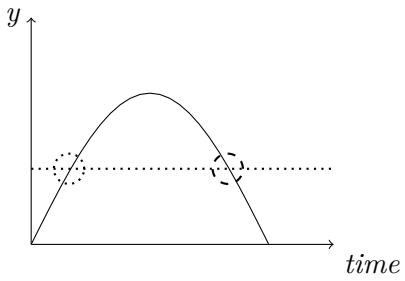


Figure 2.1: State event threshold crossing (raising-edge and falling-edge events).

2.3.2 Co-simulation

A co-simulation (co-operated simulation) is a simulation technique that allows individual models to be simulated by different simulation tools running simultaneously and exchanging state information in a collaborative manner. In other words, co-simulation is model simulation where parts are distributed over more than one simulation engine and possibly belong to more than one domain. One of the most important reasons for choosing to do co-simulation is that it allows different simulators of different domains to work together, so that modellers can use the best tool for the job at hand. Co-simulation can be achieved both with hardware in the loop and as pure software simulations. The latter can further be divided into the following types:

- Multiple continuous time simulation tools co-simulate.
- Multiple discrete event simulation tools co-simulate.
- A mix of both multiple continuous time simulation tool sand discrete event simulation tools co-simulate.

Combining simulations of discrete and continuous time models into a co-simulation forms a combined model which is more expressive than either model in isolation. In this work, the focus will be on co-simulation involving a discrete event simulator and a continuous time simulator.

2.3.3 Related co-simulation projects

This subsection lists a few related projects and shortly compared them to this work. However, a detailed comparison is planned but requires the semantics of our work to be finished, providing a better base for comparison.

2.3.3.1 CosiMate

The CosiMate project [31] is a backbone co-simulation tool which is used to get a flexible co-simulation solution in the sense that the connected tools need only one co-smulation interface to CosiMate, and all supported tools can be coupled. CosiMate allows one to interconnect several simulators and languages such as ModelSim (VHDL), C, C++, SystemC, Matlab/Simulink,

StateMate, Saber, Easy5 and AMESim. The drawback of this backbone approach is a loss in performance: the central Cosimate tool has to sychronize all simulators, which cause waiting by the simulators on each other. Note that this is inherent to a backbone or bus like structure of simulators in a co-simulation setting.

For this work, we expect that this drawback causes a too large performance degradation, and a coupling of *one* continuous-time simulator to *one* discrete-event simulator is sufficient for this work.

2.3.3.2 MODELISAR

The MODELISAR project [32] aims to significantly improve the design of systems and of embedded software in vehicles. The Functional Mockup Interface definition is one result of the MODELISAR project. The intention is that dynamic system models of different software systems can be used together for software/model/hardware-in-the-loop simulation.

The difference with this work is that it allows multiple continuous-time systems to be interconnected and co-simulation is also extended in the area of hardware-in-the-loop simulation where this work is focused on the co-simulation of one discrete-time system and one continuous time system with the distinct possibility of fault injection at the boundaries of both simulated systems. The advantage of connecting *one* continuous-time simulator to *one* discrete-event simulator is a performance gain for simulation. Furthermore, it benefits the understanding of a co-simulation since there is only one interaction between simulators and it simplifies model maintenance, a co-simulation exists only of 2 models instead of n models. When both domains can be sufficiently expressed in their simulators there is no need for multiple simulators.

2.3.3.3 Ptolemy II

A different approach is found in the Ptolemy II tool [33] where instead of linking/interconnecting multiple tools, one tool offers a heterogeneous simulation framework that can be used to model many different domains in one heterogeneous model. This is implemented by using a hierarchical model structure in which a different model of computation can be used (indicated by a so-called director) on each level of the hierarchy. Such an integrated solution is useful, but currently it uses graphical modelling symbols which deviate too much from commonly used symbols. Furthermore, Ptolemy II, being a generic tool, does not always meet the necessary domain-specific requirements.

Indeed, using different tools connected through co-simulation can offer more domain-specific facilities. This is the main reason *not* to use Ptolemy II.

3

Unifying Overture ASTs

This chapter explains how the Overture development and its internal structure have developed over time, and presents the efforts that have been made to unify the different ASTs inside a single common representation. The motivation for this is to achieve sufficient flexibility for researchers all over the world to enable adding new extensions as easy as possible.

3.1 Overture In A Historical Context

The Overture open source initiative was started back in 2003 by the authors of [20]. From the beginning, the mission of the Overture project was defined as:

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

In the first years of Overture work was solely performed by MSc thesis students starting with [34, 35] with a focus on using XML as the internal structure. This was followed by [36] where the AST was essentially isomorphic to the concrete syntax and all classes for the AST was manually coded resulting in numerous bugs. As a reaction to this Marcel Verhoef came up with a way to automate the classes for the underlying AST by producing a tool called `ASTGen`. A key feature of `ASTGen` was the ability to automatically generate both the java code implementing the different nodes in the AST to be used inside Eclipse for the implementation of the Overture tool and in adition a possibility to generate the corresponding AST at a VDM level. The latter was desirable in order to enable development of core components using VDM itself in the same kind of bootstrapping fashion that was successfully applied in the development of VDMTools [37]. The approach where a component was specified using VDM and then subsequently code generated (using the Java code generator from VDMTools) was then used in a series of Master thesis projects used this approach where the static semantics [38], proof support using HOL [39], connection to JML [40], test automation [41] and coupling to UML2.0 [42] was carried out. In parallel with these efforts a stand-alone command-based tool called VDMJ was developed by Nick Battle [28]. An attempt of building a common Overture front-end using Eclipse was then made in [43]. This naturally also meant that basic conversions between the AST generated using `ASTGen` and the AST inside VDMJ was needed.

3.2 Overture ASTs

Before this PhD the internal structure of Overture included two ASTs. The Overture AST is carefully defined such that development can be done both at a VDM level and a Java code level, enabling a Java parser to provide an AST to be used for VDM interpretation and Java execution. The second AST is part of VDMJ. It is hand-coded and distinguishes itself by a very close integration between functionality and the AST itself, making it very difficult to extend. Each node includes parts of e.g. the type checker, the interpreter and the proof obligation generator. In figure 3.1 the Overture features are grouped together with the AST they operate upon. The features in VDMJ is a monolithic unit and cannot be separated from either each other or the AST whereas the features in Overture can exist independently and only depends on the AST as an external entity.

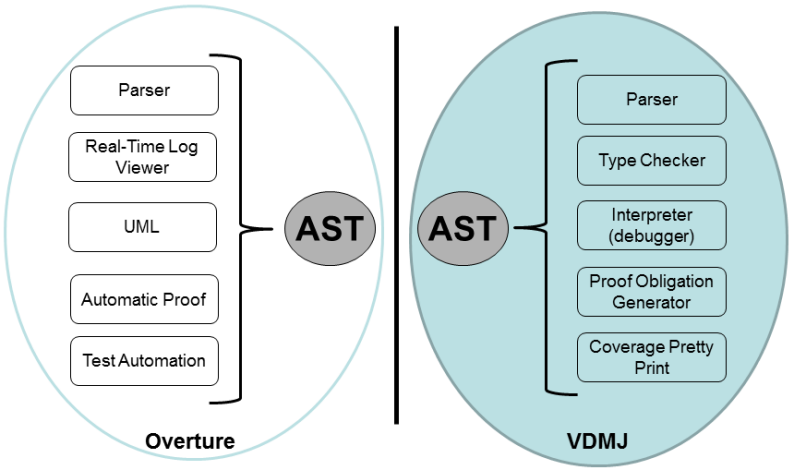


Figure 3.1: Overture ASTs

3.2.1 The Overture AST

The Overture AST is generated based on a VDM-SL type structure by a tool named `ASTGen` as mentioned in section 3.1. It uses VDM types to generate both VDM and Java classes. The most important types used are union types (enabling inheritance) and record types (enabling tree leafs to have fields). In listing 3.1 a small example is illustrating how an apply and unary expression can be sub-classed from an expression and how record types e.g. `UnaryExpression` can be used to specify fields e.g. `operator` and `expression`.

```
Expression = ApplyExpression | UnaryExpression |  ...

ApplyExpression ::
  root : Expression
  args : seq of Expression;

UnaryExpression ::
  operator : UnaryOperator
  expression : Expression;
```

Listing 3.1: Extract from the existing grammar for ASTGen with the apply and unary expression.

The work flow of `ASTGen`is illustrated in figure 3.2 where an AST grammar (like listing 3.1) is given as input to `ASTGen` which in turn will generate source code for VDM and Java.

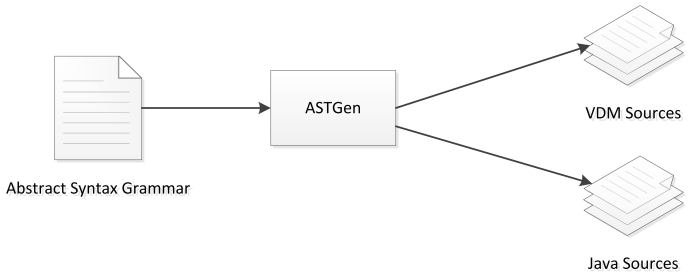


Figure 3.2: Overview the AST generator.

3.2.2 The Hand-coded AST inside VDMJ

The hand-coded AST inside VDMJ follows the Object Oriented (OO) principles and is optimized for speed and debugging. The AST does not only define the VDM abstract syntax but it also defines methods for type checking, evaluation, proof obligation generation and a wide range of utility methods to collect information from nodes. The methods are implemented so that a base class defines a method where as all possible subclasses provide an implementation or override an inherited implementation if required. This allows all expressions to be checked by calling their `typeCheck` method which calls the correct implementation based on the subclassing; however, this is not a flexible way to implement the functionality since all logic is placed inside the AST itself.

3.2.3 Comparison of ASTs

The internal structure of Overture illustrates that both ASTs can be used for tool development. The comparison will be centred around the following key areas: *Multilevel development, Maintainability, Extendibility and Automated Navigation. Multilevel development* covers the ability to use the language itself to specify tools of that language, this is only supported in the tree from `ASTGen` i.e. take your own "medicine". When it comes to *maintainability* the Overture AST has a clear advantage since a grammar file defines the AST at a higher level than the code itself. The fact that the VDMJ AST includes functionality also makes it harder to maintain the AST. However the functionality added inside the tree in VDMJ is easier to maintain since it is grouped with the node it affects. None of the ASTs provide a good solution to *extendibility*, the Overture AST has a grammar file which is easy to change but it does not support fields on super classes, on the other hand the VDMJ AST does support this but does not have grammar file. *Automatic Navigation* is particularly important for integration into an Integrated Development Environment (IDE) when an editor must provide features like: outlining, auto-completion, re-factoring etc. Both ASTs allow only a manual tree decent and have no general way to find a node which covers a specific offset in a file, nor do they provide any automated search to detect if e.g. an expression is inside a function or operation.

3.3 Unifying ASTs

In order to achieve an ideal common AST three goals have been identified:

1. *Development must be supported both at specification and implementation level, allowing the VDM language itself to be used for tool specification.*
2. *The AST must be extensible while extensions must be kept isolated within a plug-in.*
3. *Sufficient navigation machinery must exist for an editor so that features like e.g. completion and re-factoring can be implemented easily.*

The following subsections will explain the essential principles for the new AST and what changes are required to a generator in order to produce an AST that complies with the identified goals.

3.3.1 What Is Essential For A New Unified AST

It is essential that a new AST is easy to maintain and easy to extend, thus it must only contain functionality essential for the tree structure. To achieve both easy maintenance and support for specification and implementation level development an abstract tree definition can be used like in the `ASTGen` tool. However, extendibility is another matter that need attention; This can be handled by allowing one tree to extend another, by adding new nodes and fields or even refining a type of an existing field.

3.3.2 A New AST With Generator

The new AST that has been developed has an improved structure compared to both the existing Overture and VDMJ trees. The main addition made here is the ability to extend an AST while keeping the changes isolated in a plug-in architecture, explained in detail in section 3.3.3. Secondly, the AST is specified using a grammar file inspired by SableCC[1], and can generated to both VDM and Java as supported by `ASTGen`. The main structural change compared to the Overture AST is the ability to add fields to super classes allowing e.g. an expression field to be added to all unary expressions as illustrated in listing 3.2.

```
Abstract Syntax Tree
exp {-> package='org.overture.ast.expressions'}
    =   {apply} [root]:exp [args]:exp* [argtypes]:type* [recursive]:definition
    |   #Unary
    |   ...
    ;

#Unary {-> package='org.overture.ast.expressions'}
    =  {head}
    |   {tail}
    |   ...

Aspect Declaration
exp->#Unary
    = [exp]:exp;
```

Listing 3.2: Extract of the new AST grammar showing the apply and unary expression.

[1] `http://www.sablecc.org/`

3.3.3 Extendibility and Isolation of Additions

In this subsection we describe a new way to specify and generate a tree that extends a base tree and preserves backwards compatibility with its base tree. Extensions are only visible to components that depend on the extended tree otherwise only the base tree structure will be accessible. In figure 3.3 examples of extensions required by Overture is shown. Each box defines a plug-in feature that needs its own extensions like a type field to store the derived type information from a type checker and a proof obligation generator needs a place to attach proof obligations. The figure also illustrates that an extended tree may be further extended; In this case the derived type information is needed by the interpreter.

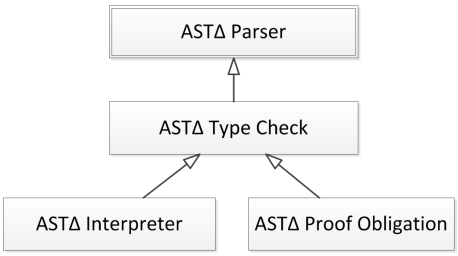


Figure 3.3: Illustration of AST extensions in Overture.

The extension principle is based on sub-classing from a class-hierarchy where each extended node sub-classes the corresponding node in the base hierarchy. This allows a new extended AST to be used by any implementation that supports its base tree e.g from figure 3.3 any feature that uses a typed AST can also use an interpreter AST. To achieve isolation between extensions we require extensions to be declared in a separate file so that a generator (illustrated in figure 3.4) can combine them into a single tree and generate a converter from the base tree to the extended tree. To illustrate the type check extension listing 3.3 shows how expressions are extended with a field for the derived type information.

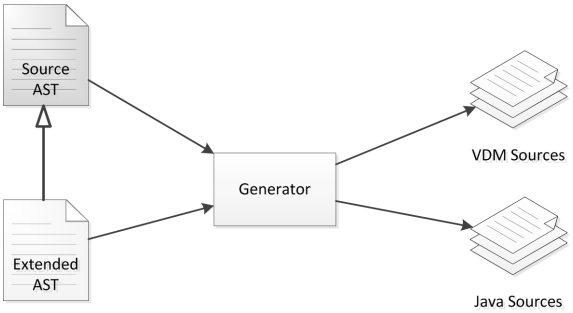


Figure 3.4: Extending an existing AST.

```
Aspect Declaration
exp = [type]:type;
```

Listing 3.3: Example showing how a type can be added to all expressions.

4

Tool Automation

This chapter presents different ways to automatize tool support around the VDM notation. In general, VDM models are not executable but a subset of them can be interpreted and section 4.1 explains how this subset can include looseness as well as interfaces to legacy code. Afterwards, section 4.2 exploits this looseness in a combinatorial testing context using regular expressions. Then section 4.3 takes inspiration from the combinatorial testing principles to automatically generate alternative deployments in VDM-RT models, to replace manual labour when exploring optimal system solutions. This is followed by section 4.4 illustrating how automation between the UML notation and VDM++ can be made. Here sequence diagrams can be used to represent the regular expressions used in section 4.2. Finally, section 4.5 extends the interpreter work from section 4.1 to cope with cyber-physical systems using co-simulation.

4.1 Interpreter

The power of formal methods traditionally lies in being able to write a specification, and to statically analyse and subsequently formally refine that specification to produce a target implementation that is formally verified. However, formal specifications can also be used for direct execution of a system under construction. Executability of specifications in general have caused a debate in the community [44, 45] but although this may "pollute" specifications such features are advantageous to get fast feedback on specifications. Thus, benefits can be gained from exploring design options through simulation even before any formal analysis of the model has been carried out [46]. One way of efficiently finding problems with a formal specification is to evaluate expressions making use of the definitions from the specification [47]. In the event that such expressions do not yield the expected values, it is essential to be able to deterministically reproduce the problem, for example by debugging the model using a deterministic interpreter.

In this work we focus on the ability to execute a simulation when looseness is present in the specification in such a way that the results are deterministic and reproducible. This means that the result of our interpreter will correspond to a valid value from one model of the specification, and in a given input it will always produce that value. All valid models can be collected [48], however, our industrial experience indicates that the ability to investigate multiple models is mainly of academic value. Loose specifications arise because specifications generally need to be stated at a higher level of abstraction than that of the final implementation. Looseness enables the modeller

to express that it does not matter which value a certain expression yields, so long as it fulfils certain requirements [48].

In VDM-RT a specification represents a potentially infinite set of semantic models due to the looseness [49] present in the language. This allows the modeller to gain abstraction and to avoid implementation bias. In the presence of looseness an interpreter will simply have to represent what is possible in one of the semantic models. To perform a specification simulation, tool support must provide an interpreter for the specification language, and a debugging environment that allows the designer to investigate problems. Given a specification with looseness, the tool support must also provide a *deterministic* interpreter and debugger, otherwise problems with the specification would not be reproducible and so could not be investigated easily.

Programming language interpreters and compilers are typically not deterministic when the language includes concurrency, and debuggers exhibit a higher degree of non-determinism, where the interference of the user can easily change the behaviour of the program being debugged. Existing work has examined the problems of the deterministic execution of programming languages with threads [50]. Others have added assertions to check the determinism of multi-threaded applications at the programming language level [51]. Here we demonstrate how it is possible to interpret and debug formal models written in VDM-RT in a deterministic manner.

4.1.1 Interpreting Sequential VDM Models

In order to simulate the evaluation of functions or operations from a VDM model an interpreter first needs to be initialised with the definitions declared in the model. This is achieved with a tree traversal of the AST produced by the parser using the additional derived type information from the type checker. Essentially, the syntactic definitions must be transformed into their semantically equivalent representations. However, since VDM is not designed to be interpreted this transformation can be quite complicated because of the potential dependencies between different definitions. Note however that the interpreter presented here operate with specific values and not symbolic values [52].

The initialization of a specification amounts to the evaluation of the state of the system, either in VDM-SL state definitions or VDM++ and VDM-RT static class definitions. This involves the evaluation of initialization clauses for the various state definitions, guided by their definition dependency ordering. VDM does not define a syntactic order for the evaluation of definitions, but rather the order is inferred from the definitions' dependencies on each other: definitions that do not depend on others are initialized first, in any order; then definitions that depend on those are initialized, and so on. Every time a specification is re-initialized, it returns to the same initial state.

When the initialisation is complete the interpreter is ready to start the evaluation of a test expression making use of the definitions and state from the VDM model. In order to perform such a test evaluation, the interpreter creates a runtime *context* that initially contains the values defined in the state (if any). The evaluation then proceeds by evaluating any arguments by direct recursive traversal evaluation of the argument expressions in the AST and then executing the function or operation body in a new stack frame that is linked to the state context. The evaluation evolves in a natural recursive decent fashion, reflecting the function or operation call structure of the specification on one particular thread.

The interpreter is also able to check all pre- and post-conditions, type and state invariants, recursive measures and it performs general runtime type checking. The additional checks can be switched on or off whenever the user requires additional checking. Extra checking naturally has an impact on the performance of the interpreter but this may be a faster way to discover problems in VDM models. Semantically, bottom values (denoting undefined) will result from different kinds of run-time errors from the interpreter depending upon whether such checks are performed or not,

but the behaviour will always be deterministic. The special check for termination of recursive functions may be worth a special mention, since this is (as far as we know) not performed by any other interpreter. In VDM it is possible to define so-called **measure** functions that must be monotonically decreasing for recursive calls. This can be checked at run-time by the interpreter such that infinite recursion will always be detected.

Some VDM expressions contain looseness, for example a choice construct called a let-be expression looks like: **let** a **in set** {1,2} **in** a. This expression denotes either 1 or 2 but it is deliberately left as an implementation choice for the implementer from a refinement perspective. In order to be able to reproduce executions the interpreter must thus choose one of the possible semantic models in order to produce a deterministic interpretation. In the same way iterations over a set of elements must be performed in the same order every time to ensure a deterministic result. As a result, the evaluation of any sequential VDM model by the interpreter will always produce the same result value, even if looseness means that the result cannot be predicted (easily) ahead of time.

4.1.2 Interpreting Concurrent Real-Time models

All VDM-SL specifications and non-threaded VDM++ specifications result in a simple single threaded evaluation, as described above. Their execution will always produce the same result because VDMJ treats all loose operations as under-determined rather than non-deterministic [17].

VDM-RT simulates distributed systems and thus the initialisation process explained for sequential VDM models above also needs to deal with the deployment of object instances to CPUs, for example. The user indicates the intended deployment in the special system class and so the interpreter needs to extract the necessary information from the AST of that class. In addition, the interpreter needs to make use of the deployment information to determine whether interprocess communication over a BUS is necessary. It is also worthwhile noting that if an object instance creates new object instances (using the new constructor) during the interpretation, those new instances must be deployed on the same CPU by the interpreter.

Note that the interpreter here abstracts away from the challenges of being able to determine the global state in a distributed system [53]. Since the interpreter will always have consistent information about all the CPU's at any point of time, the traditional issues with unsynchronised clocks and dependability in distributed systems [54] are abstracted away.

VDM++ and VDM-RT specifications can have multiple threads of execution, and their evaluation can easily become non-deterministic since the thread scheduling policy is deliberately left undefined in the VDM semantics. In order to eliminate this looseness, VDMJ uses a scheduler which coordinates the activity of all threads in the system and allows them to proceed, according to a policy, in a deterministic order. This guarantees repeatable evaluations even for highly threaded VDM specifications.

VDMJ scheduling is controlled on the basis of multiple *Resources* by a *Resource Scheduler*. A Resource is a separate limited resource in the system, such as a CPU or a bus (see Figure 4.1). These are separate in the sense that multiple CPUs or busses may exist, and limited in the sense that one CPU can only run one thread at a time, and one bus can only be transmitting one message at a time. Therefore there is a queue of activity that should be scheduled for each Resource — threads to run on a CPU, or messages to be sent via a bus. The Resource Scheduler is responsible for scheduling execution on the Resources in the system.

An interpreter (of any VDM dialect) has a single Resource Scheduler. A VDM-SL or VDM++ simulation will have only one CPU Resource (called a virtual CPU) and no bus Resources; a VDM-RT system will have as many CPUs and busses as are defined in the special **system** class.

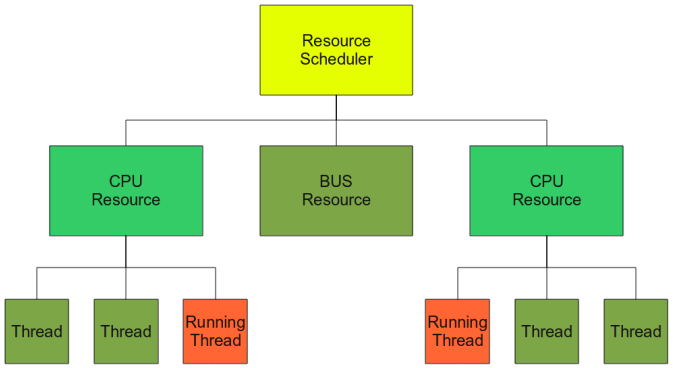


Figure 4.1: Overview of the VDM-RT resource scheduler.

Every Resource has a scheduling policy[1] potentially different for each instance of the Resource. A policy implements methods to identify the thread that is best to run next and for how long it is to be allowed to run (its timeslice).

With VDM-RT, in the event that the active thread is trying to move system time, the Resource will identify this fact. The Resource Scheduler is responsible for waiting until *all* Resources are in this state, and then finding the minimum time step that would satisfy at least one of the waiting threads. System time is moved forward at this point, and those threads that have their time step satisfied are permitted to continue computing, while those that need to wait longer remain suspended. This reflects the semantics of VDM-RT based on a time step and compute step cycle as defined in [27, 22].

For example, if there are two threads running (on two CPUs), the Resource Scheduler will offer each of them timeslices in turn. When one thread wishes to move system time by (say) 10 units, its CPU Resource will indicate this to the Resource Scheduler, which will stop including it in the scheduling process, allowing the other CPU's thread to run. Eventually, the second thread will also want to move system time (typically at the end of a statement), say by 15 units, and its CPU Resource will also indicate this to the Resource Scheduler. At this point, all active threads want to move time, one by 10 and the other by 15. The minimum time step that will satisfy at least one thread is a step of 10 units. So the Resource Scheduler moves system time by 10, which releases the first thread; the second thread remains trying to move time, but for the remaining 5 units that it needs. By default, all statements take a duration of 2 cycles, calculated with reference to the speed of the CPU on which the statement is executed. Statements (or blocks of statements) can have their default times overridden by **`duration`** and **`cycles`** statements. The core of the scheduler is illustrated by:

[1] As described in Section 2.1, this can currently be either a "*First Come First Served*" or a "*Fixed Priority*" scheduling policy, but more could be added in the future and parameterisation of these can be imagined.

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

Listing 4.1: Resource scheduling loop.

The initial loop in listing 4.1 establishes whether any resources can progress. The `CanProgress` operation does a compute step for the Resource, if possible. The `progressing` flag will be true if any Resource was able to progress. The `getTimestep` operation either returns the timestep requested by the Resource, or nil, indicating that it is not currently waiting for time to advance. If no Resource can progress and no Resource is waiting for a timestep, the system is *deadlocked*. Otherwise, if no Resource can progress and at least one is waiting for a timestep, then system time can advance by the *smallest* requested amount. In this case every Resource is adjusted by the minimum step which results in at least one Resource being able to progress. This scheduling process continues until the original expression supplied by the user completes its evaluation.

4.1.3 Support for External Java Code and GUI Front-ends

Experience shows that it is not economically feasible to formally specify all parts of a system in an industrial application [55]. Either a number of trusted components may exist or is so simple that nothing will be gained through formally specifying it. Trusted components often exist as legacy components which may already have been proven correct. Components which do not benefit for being formally specified includes simple components with very little functionality and graphical user interfaces (GUIs). In [55] work have been done on a new semantics for the VDMTools which allows a combined specification to be executed, consisting of both specification and externally specified C++ code from Dynamic Link Libraries (DLLs). Our work is based on [55] with a similar dynamic semantics but without changes to the VDM syntax. It is developed in Java and uses a VDM module or class name to lookup any external Java class which contains an implementation of a functions or operations which has a body defined as **`is not yet specified`**. The key improvements in our work is as follows:

1. No changes have been made to the VDM syntax.
2. Dynamic semantics is only changed for the **`is not yet specified`** expression and statement.
3. The compilation process is significantly simplified.

4. All Java platforms are supported.

5. External code is loaded dynamically through a simple lookup.

6. All standard VDM checks are supported: pre-/post-conditions, invariants, exceptions including proper internal errors for error handling of the external code.

This approach enables a specification to be combined with externally specified components while the interpreter itself stays in control during the execution, while this is good for accessing legacy code it is not sufficient for a controlling GUI. A GUI can advantageously be used to present specifications to domain experts which may not have a formal background. Previous work has been carried out in this area for VDMTools, providing a CORBA interface. CORBA is a standard that enables software components written in multiple computer languages and running on multiple computers to work together (i.e. it support multiple platforms); Because it supports multiple languages and platforms it is complex to integrate compared to a plain Java integration. The principles behind this work is to reduce the integration time so that this kind of integration becomes more feasible for industry. To achieve fast and easy integration an architecture were designed allowing this by only sub-classing a single class provided by the interpreter. This then enables the GUI to get a delegate that can control the interpreter, in the same way a terminal execution is carried out. The GUI integration will not change the behaviour of the specification but allow a user to test the specification by applying domain specific scenarios in a way that do not require any formal modelling knowledge.

4.2 Combinatorial Testing for VDM

Tool support is essential in order to be able to perform automatic testing. The tool presented here enables the application of combinatorial testing principles for the formal method VDM [15]. Special attention has been given to the design of a user interface within the Eclipse platform that is intuitive and efficient for developers using VDM.

The tool described has been developed as part of the Overture tool initiative [6]. The work is based on earlier work conducted with the TOBIAS tool by the research group led by Yves Ledru [56, 57, 58, 59] and a MSc project creating an initial VDM specification for combinatorial in VDM [41]. Combinatorial testing has drawn attention from many other researchers, and it is also related to recent work conducted on the generation of test cases from model checkers [60, 61, 62, 63].

A language extension has been made to all existing VDM dialects (VDM-SL, VDM++ and VDM-RT) enabling a section of combinatorial trace definitions to appear inside any VDM specification. The Overture tool makes use of that extension by expanding each trace definition into a collection of test cases then executing them and presenting the results. This enables small regular expressions to bring the system into a particular state and from there test functions and operations with a range of generated inputs which for optimization can be filtered through different principles.

4.2.1 VDM and Traces

In order to automate the testing process VDM has been extended with a notation to define the traces that one would like to have tested exhaustively. Such traces exploits the fact that VDM expressions can be written so that they deliberately contain looseness, enabling multiple different

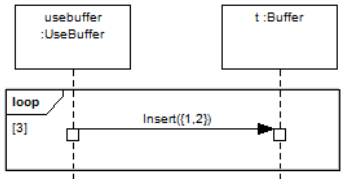


Figure 4.2: Sequence diagram representing the `TTest` trace

interpretations to be exploited during test expansion [64, 17]. Traces are used to express combinations of sequences of operations to be tested, and are conceptually similar to UML Sequence Diagrams. Overture enables traces to be exported as UML Sequence Diagrams as shown in Figure 4.2 [12].

The body of a trace definition can be thought of as a regular expression for identifying sequences of operation calls on different objects. The combinations tested are the complete set of operation sequences that would match the regular expression.

It is possible to introduce variable bindings (with looseness that is expanded to all possible combinations), alternatives (using the `|` operator) and repetitions (using different kinds of repeat patterns). The tool automatically generates all possible test cases from a trace. So for example if we have an operation called `Insert` then a trace definition might be as follows:

```
TTest: (let x in set {1, 2}
        in t.Insert(x)){3}
```

where `x` is a variable binding to a set of values and `t` must refer to an instance of a class where the `Insert` operation is defined. This particular trace definition will expand to a call of `Insert` operations with 1 or 2 as an argument, repeated three times because of the repeat pattern of `{3}`. So the expansion (i.e. all the possible sequences which match the pattern) would be:

```
TC1: t.Insert(1);t.Insert(1);t.Insert(1)
TC2: t.Insert(1);t.Insert(1);t.Insert(2)
TC3: t.Insert(1);t.Insert(2);t.Insert(1)
TC4: t.Insert(1);t.Insert(2);t.Insert(2)
TC5: t.Insert(2);t.Insert(1);t.Insert(1)
TC6: t.Insert(2);t.Insert(1);t.Insert(2)
TC7: t.Insert(2);t.Insert(2);t.Insert(1)
TC8: t.Insert(2);t.Insert(2);t.Insert(2)
```

The order of these cases is not significant. They are entirely independent.

For every test case there are three possible verdicts. If all operation calls yield a result satisfying their post-conditions (if defined) the verdict is that the test case has passed. If one of the calls violates a pre-condition of that operation then the verdict is inconclusive in the sense that the test case has, most likely, tried the operation outside the domain for which is was defined. Finally, if any other run-time error is detected, the verdict is that the test case failed (i.e. an error has definitely been detected). In most cases this will indicate missing pre-conditions but failures can also be caused by erroneous explicit definitions (i.e. simple “bugs”).

4.2.2 Reduction Techniques for Filtering the Test Cases

Because of the regular structure of a trace definition the expansion produces many tests which differ only slightly. If a test execution fails after (say) 10 operation calls then all tests which start with the same 10 operation calls will fail in the same way, for the same reason. This gives us an opportunity to improve the test execution performance by skipping such tests, and marking them as "filtered" by the original failed test with the same stem.

It is easy to produce traces that expand to thousands of tests and, even with test filtering, it may take a very long time to execute the full trace expansion. In these cases, it is desirable to reduce the expanded test set to a manageable size prior to execution, given that the subset of tests that remain are representative of the whole.

Related work [59] has described the analysis of test "shapes" as a way to make the reduced test set as representative as possible. A test shape is a sequence of named operation calls, regardless of their argument values. By guaranteeing to retain at least one example of every test shape the reduced set of tests can claim to be more representative than, say, a random selection of tests, which may eliminate important shapes. Our implementation in VDMJ performs both random and shaped test reduction on this basis and this can be selected upon invocation of the combinatorial testing.

However, because we inject explicit variable assignments into the test sequences, we have a further basis on which to distinguish test shapes. So in addition to simple shaped reduction, our implementation also permits variables to be taken into consideration in the shape analysis. This can be simply by their name and position in the test sequence, or it can consider both their name and the value being assigned to them. The effect is to produce finer grained shapes, which therefore limits the degree of reduction possible.

For example, consider the following trace (making use of loose pattern matching of sets of values):

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

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

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

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

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

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

Note that as the specification of shapes becomes more detailed, it is not possible to achieve the requested 1% reduction in the number of tests. This is a natural consequence of the reduction

process retaining at least one test of each shape.

4.2.3 Graphical User Interface

Having a good user interface is essential for combinatorial testing, because even simple traces can expand to thousands of tests, which will be difficult to overview. This will require a well-structured interface with proper filtering mechanisms, enabling filtering based on the test verdicts. We have developed a GUI which is able to show a tree structure of all test groupings with icons showing their verdict. It is possible to select any test and inspect the sequence of calls made and if the test failed, see what sequence of calls led to the failure, but more importantly it is possible to send the test to the interpreter and re-run the test within the debugger. This allows a more thorough inspection of the state of the model at the point where problem occurred.

4.3 Automated Exploration of Alternative System Architectures with VDM-RT

Choosing the optimal deployment of a distributed embedded application onto alternative hardware configurations is difficult and time consuming industrially and is not simple in non-industrial cases, either. When developing a new product, a company must chose a hardware architecture that ensures both that the system behaves correctly according to its functional and timing specifications but also keeps its production cost at a minimum. The investigation to find this tradeoff between cost and performance can be very expensive if carried out at implementation time. A company can save money and development time if they are able to quickly explore the design alternatives before the start of the implementation. In this section we describe a method and associated tool support to assist in finding the best system design solution.

As distributed real-time embedded systems become more and more prevalent around us, new techniques must to be used to ensure lower costs of development while keeping the service quality high. The quality of these kind of systems is normally not only measured by functional correctness but also by timing behaviour correctness. Because timing correctness is so essential when developing such systems, the implementation cost can increase considerably if it is discovered at later stages of the development that the selected hardware architecture cannot fulfil the timing parameters. Typical design questions considered by an architect are [27]:

1. Does the proposed architecture meet the performance requirements of all applications?
2. How robust is the chosen architecture with respect to changes in the application or architecture parameters?
3. Is it possible to replace components by cheaper, less powerful equivalents to save cost while maintaining the required performance targets?

Models of software/hardware can be used to assist the system architect in answering these questions. They have previously been used to explore and validate different deployment architectures even before the implementation cycle starts [65, 66]. By doing so, it is possible to gain knowledge, at an early stage, of the product that is being developed, even before any deployment decisions have been made. Usually, companies that wish to develop a distributed embedded system have certain target Printed Circuit Boards (PCBs) in mind, which support a small limited range of CPUs. These PCBs establish the architecture of the hardware and by using models and simulation techniques, one can gain insight into which PCB should be selected in order to fulfil the

project requirements. Furthermore, one can identify which kinds of CPUs and buses are needed to respect the speed/capacity requirements of the application. Having a method and tool support to test out all the interesting PCB/CPU/bus combinations and identify which ones satisfy the system timing invariants, would give the system architect an advantage when making such initial design choices.

A prototype has been made enabling architectures and deployments to be generated based on the components of the system. This enables a system architect to generate a large number of possible configurations for the system. A challenge is then to rank configurations in such a way that the architect can answer the design questions. The VDM language itself offer a few ways to decide if a specification is acceptable: run-time error, dynamic type check and checks for pre, post and invariants. However none of which are depended on timing thus some work [67] has been put into creating a system timing invariants checker allowing one to specify invariants like minimum time between two operation calls and check this at run-time. Based on this it is now possible to check the effect of a slower CPU or BUS.

4.4 Automated Translation between VDM++ and UML

Various techniques for software modelling are used in industry often with advanced tool support allowing integration with target languages. The Unified Modelling Language (UML) is a widely used modelling technique during the software design process, it has rich tool support for code generation and re-factoring. To improve industrial use of VDM a set of transformation rules have been defined that enable automated transformation between VDM and UML. This opens up for the possibility of rapid prototyping of VDM specifications from the UML notation.

The core of this work is centred around transformation rules that translate between both VDM++ and UML Class diagrams and between VDM traces and UML Sequence diagrams; the latter enables traces to be visualized. The VDM++/UML transformation were specified in VDM itself. The abstract syntax tree representation of both VDM and UML was defined so that both a VDM++ and Java AST could be generated, illustrated as circles in figure 4.3. This enabled the specification to be code generated to Java and directly integrate with the existing Overture AST and parser.

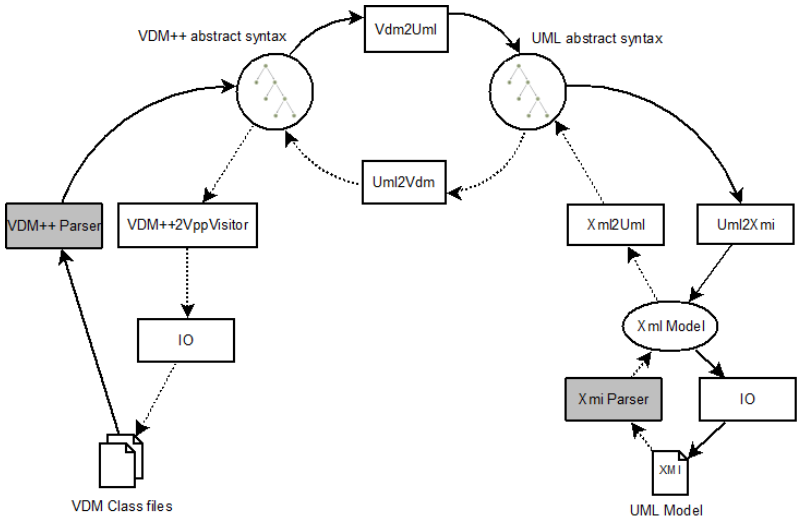


Figure 4.3: Overview of components involved in the VDM++ and UML transformation process.

The transformation between VDM++ and UML is performed at an abstract level, and specified using VDM++ itself and bootstrap to run as a standalone program[37]. An Abstract Syntax Tree (AST) is generated for both VDM++ and UML. A transformation is then specified in VDM++ to accomplish the transformation between the two abstract representations. Fig. 4.3 gives an overview of the architecture of the transformation. The gray boxes denote pure Java implementations whereas all other boxes are specified in VDM++ and subsequently automatically converted to Java using VDMTool's support for automatic generation of Java code. Starting with a VDM model, it is first parsed to populate the VDM++ AST which is then transformed to a UML AST equivalent. The UML AST must then be de-parsed to the XML Metadata Interchange (XMI) format in order to be integrated with UML modeling tools. The dotted arrows in Fig. 4.3 shows transformation from UML to VDM++ whereas the solid arrows show the transformation from VDM++ to UML.

The XML Metadata Interchange (XMI) is a standard for exchanging meta data information via Extensible Markup Language (XML). It can be used for any meta model that can be expressed in the Object Management Group (OMG) Meta-Object Facility (MOF). XMI is standardized by the OMG [68]. XMI is widely used to exchange UML models by UML modeling tools.

There are several incompatibilities between different tool vendors' implementations XMI for UML. At the diagram interchange level the standard is almost nonexistent and there are multiple incompatibilities between abstract models. Unfortunately, this means that the goal of XMI, i.e. to enable the free interchange of UML models, is rarely possible. Moreover the new XMI 2.1 standard is even less widespread which limits the interchange of models even further.

4.4.1 Transformations for UML Class Diagrams

The static structure representation offered by a UML CD is largely conceptually compatible with VDM++ models. This includes concepts such as classes, inheritance, associations and multiplicities which all have a one-to-one relationship between UML and VDM++ making it possible to move both ways. However, only the static structure of a VDM model can be efficiently transformed to UML. VDM++ has a well-defined semantics for determining properties about a model, e.g. using pre- and post-conditions. Such elements are awkward to display in a visual UML model, easy to express in text. It would be possible to transform such bodies to OCL in UML and give the user access to the relevant definitions via a UML tool, but with the disadvantage of having to do a lot of navigation to access the definitions. This could be easily be done for a subset of VDM++ but OCL has a number of limitations that would increase the complexity of the mapping and make the bi-directional transformation harder to understand for the user. This can easily be reconsidered at a later stage if the analysis tools of such OCL expressions are improved.

4.4.2 Transformations for UML Sequence Diagrams

The primary transformation direction chosen in this work is to transform UML SDs to VDM++ trace definitions, since it is considered to have the greatest value. By transforming an SD into a trace definition, a trace can be seen both textually or visually. In this section the focus will be to present a subset of the rules to enable this transformation. The rules will be specified in such a way that a round-trip between VDM++ and UML is possible [69].

4.4.2.1 VDM++ Trace Definitions

A new VDM++ definition block has been introduced for trace definitions. The listing below shows an ordinary VDM++ class `Stack` followed by another class, `UseStack`, which has a **`traces`**

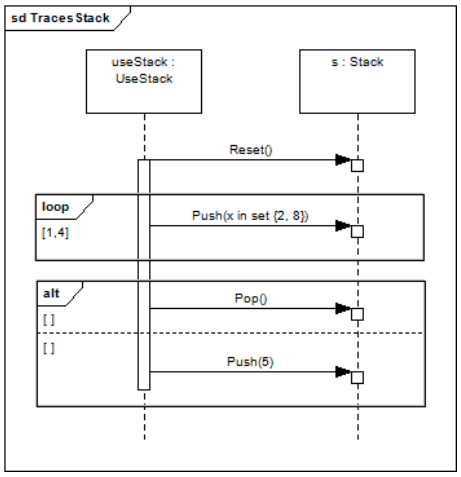


Figure 4.4: A SD showing a simplified trace definition.

block containing trace definitions. Each trace definition is given a name which is separated from its body by a colon. The body of a trace definition is similar to a regular expression for identifying sequences of function/operation calls on different instances of classes. It is possible to introduce bindings (also with looseness that is expanded to all possible combinations), alternatives (using the | operator) and repeat patterns (using different kinds of repeat patterns). Tool support exists for automatic test case generation using combinatorial testing principles.

4.4.2.2 UML 2 Sequence Diagrams

UML 2 SDs are intended to present a dynamic interaction between objects of a system. SDs show a collection of scenarios illustrating how the flow of control between different instances of classes evolves. The vertical line below each instance displayed at the top of an SD indicates a lifeline. The horizontal arrows stemming from the lifeline indicate interactions with other lifelines. An arrow going from such a lifeline indicates a call of an operation named above the line with its parameters. An operation call on the same instance is showed as an arrow originating from and arriving at the same lifeline.

A significant improvement made to SDs in UML 2 is the ability to express procedural logic and the ability to nest fragments to an unlimited degree. Most important is the ability to express repetitions and alternatives using the **`loop`** and **`alt`** constructs, respectively. Both of these constructs may also be nested.

4.4.2.3 Sequence Diagrams and Trace Definitions

A UML SD visually represents a sequence of executions. As an example, fig. 4.4 shows the message calls `Reset`, `Push` and `Pop` which can be directly related to the application of these operations:

```
class Stack
... The usual kinds of basic Stack definitions
end Stack
class UseStack
instance variables
  stack : Stack := new Stack();
traces
 TS: stack.Reset() ;
     let x in set {2,8} in stack.Push(x){1,4};
     (stack.Push(9) | stack.Pop())
end UseStack
```

Traces like this will expand to 16 test cases (each being a sequence of operation calls):

```
TC1: stack.Reset();stack.Push(2);stack.Push(9)
TC2: stack.Reset();stack.Push(8);stack.Push(8);stack.Pop()
...  14 more
```

4.5 Co-simulation and Semantics

The initial semantics work have been to rework the semantics definitions from [27] in the style of Plotkins SOS [70, 71, 72]. In this process a number of cases needing further clarification has been identified i.e. deployment of threads, usage of periodic statements, the solver and event handler functions. The semantics splits up the execution into two areas: computation and time steps where the computation steps can be taken on all nodes if possible. The time steps are taken in sync for all nodes. The main difference from the standard VDM-SL semantics is that variables have transactions, that hide changes to variables from the rest of the system while the thread executes (waiting for enough time steps to complete the current calculation).

The following Skip rule illustrates that a skip instruction can be performed if the conditions above the line is true where $instr$ will be modified after. The rule describes that if there exists a thread i where the thread can execute in the configuration C ($exec(C, i)$) and if the next instruction for that thread is a SKIP then the skip is removed from the instruction sequence of that thread.

$$\text{Skip}\ \frac{\begin{array}{l} i \in Thread \\ exec?(C, i) \\ \textbf{hd}\ C.instr(i) \in \text{SKIP} \\ instr' = C.instr \dagger \{i \mapsto \textbf{tl}\ C.instr(i)\} \end{array}}{C \xrightarrow{s} \mu(C, instr \mapsto instr')}$$

The Time Step rule illustrates when a time step can be performed and what side effect it will have. The rule describes that a time step can only be taken when all threads either have a duration as its next instruction to execute (e.g. at the head of the instruction sequence (IS)) or has finished, $t > 0$ describe that this is the time step rule for non zero durations thus a time step will be required by the external underspecified continuous time solver. The solver will progress forward in time and may read any output variables from the VDM model and in turn write any input variables and generation events. Lastly the rule describes that all durations at the head of the IS of the current threads will be decremented and any transactional variables related to the thread will be committed. However, it is clear that more work is needed to simplify the semantics definition presented here. Through the semantics work it has become clear that the semantics presented

in [27] do not entirely match the implementation of co-simulation in the DESTECS tool. The reason for this is combination of underspecified semantics definitions and implicit assumptions in the semantics. Future work is required to clarify this and make the necessary adjustments to semantics and the co-simulation tool.

$$
\begin{array}{l}
targetThreads = \{i \mid i \in \mathbf{dom}\ C.instr \bullet \quad \mathbf{hd}\ C.instr(i) \in Duration \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge exec(C,i)\ \Rightarrow\ \mathbf{hd}\ C.instr(i) = \{\,\}\} \\
t \in Time \\
\forall j \cdot \in targetThreads \bullet \quad (\mathbf{hd}\ C.instr(i) = mk\_Duration(d)\ \Rightarrow \\
\qquad\qquad\qquad\qquad (t \leq d \wedge \nexists\, t' \cdot \in Time \bullet t' > t \wedge t' \leq d)) \\
\forall m \cdot \in \mathbf{rng}\ C.linkset \bullet \quad (m = mk\_Message(\text{-}, l, u) \\
\qquad\qquad\qquad\qquad \wedge (C.now + t) \leq u \\
\qquad\qquad\qquad\qquad \wedge \nexists\, t' \cdot \in Time \bullet t' > t \wedge (C.now + t') \leq u) \\
t > 0 \\
(t_s, now', events, ioval' = \underline{solver}(t, C.ioval, D) \\
events \neq \{\,\} \\
\quad instr' = C.instr\dagger \quad \{i \mapsto \{mk\_Duration(d\text{-}t_s)\} \curvearrowright \mathbf{tl}\ C.instr(i) \\
\qquad\qquad\qquad\qquad \mid i \in targetThreads \wedge \mathbf{hd}\ C.instr(i) = mk\_Duration(d)\} \\
\quad ioval' = ioval' \dagger \{x \mapsto C.modif(x,i) \quad \mid (x,i) \in \mathbf{dom}\ C.modif \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge C.modif(x,i) \neq \bot \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge \mathbf{hd}\ C.instr(i) = mk\_Duration(0) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge exec(C,i)\ \Rightarrow\ \mathbf{hd}\ C.instr(i) = \{\,\}\} \\
\quad modif' = C.modif \dagger \{(x,i) \mapsto \bot \quad \mid (x,i) \in \mathbf{dom}\ C.modif \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge C.modif(x,i) \neq \bot \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge \mathbf{hd}\ C.instr(i) = mk\_Duration(0) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge exec(C,i)\ \Rightarrow\ \mathbf{hd}\ C.instr(i) = \{\,\}\} \\
eventThreadPairs = \{(j,e) \mid e \in events \wedge j \in Thread \wedge j \notin \mathbf{dom}\ instr'\} \\
\forall (\cdot\, j, e) \in eventThreadPairs \bullet \forall (\cdot\, j', e') \in eventThreadPairs \bullet j = j' \Leftrightarrow e = e' \\
eventMap = \{j \mapsto ((is_e, n_e) = evhdlr(e)) \mid (j,e) \in eventThreadPairs\} \\
status' = C.status \dagger \{j \mapsto active \mid (j,\text{-}) \in eventThreadPairs\} \\
instr'' = instr' \dagger \{j \mapsto is \mid j \in \mathbf{dom}\ eventMap \bullet (is,\text{-}) = eventMap(j)\} \\
thrNode' = C.thrNode \dagger \{j \mapsto n \mid j \in \mathbf{dom}\ eventMap \bullet (\text{-}, n) = eventMap(j)\} \\
status'' = status' \dagger \{j \mapsto waiting \mid j \in \mathbf{dom}\ status' \bullet j \notin \mathbf{dom}\ eventMap\}
\end{array}
$$

Time Step

$$C \xrightarrow{s} \mu(C, ioval \mapsto ioval'', status \mapsto status'', modif \mapsto modif', instr \mapsto instr'', thrNode \mapsto thrNode')$$

5

Status of the PhD Work and Concluding Remarks

The main goal of the PhD project is to show that it is possible to build an architecture for a common platform that can be a basis for different research groups in development of new features and/or new dialects of VDM. In addition, this PhD project aims to provide automation of the kinds of analysis that can be performed using different dialects of VDM so that it may be worthwhile to apply it in an industrial setting. This chapter gives a summary and status of the work carried out so far, as well as an outline of the work planned for the remainder of the project for the different research directions.

5.1 Summary of Work

So far the work in this PhD thesis has been structured in a number of areas that commonly can be seen as ways to increase automation around the Overture/VDM tool support. Most of work performed in these areas has been published already but in some of the areas the work is still ongoing and as a consequence publications have not yet been made. The different areas, their status and my contribution are outlined in the subsections below.

5.1.1 Unifying ASTs inside Overture

This work is still ongoing and thus nothing has been published yet. However, it is believed that the extension capabilities in the new AST automation can be beneficial for tool builders in other areas, so it is possible that a publication will come out of this before the PhD is completed (but this is too early to tell).

The inspiration for this work came from `ASTGen` produced prior to this PhD by Marcel Verhoef. My main contribution has been the concepts behind the extensibility feature inspired by the plug-in structure used in Overture. I have outlined the principles of a new AST and developed a generator for it, taking the existing VDMJ usage into account in order not to fully have to redevelop all its functionality. Finally, I have implemented most of the new AST into Overture to verify that the principles improved the internal structure of Overture.

5.1.2 Interpretation/Debugging of VDM Specifications

This work has been published in [9] and this work is complete. The main development of this work has been carried out by Nick Battle, my primary contribution has been to come up with the principles behind the front-end GUI for the debugger and the underlying protocol integration into the interpreter as well as the handling of highly threaded VDM-RT models. I have implemented most of the debugger front-end and the underlying protocol integration with the interpreter.

5.1.3 Test Automation using Regular Expressions

This work has been published in [10] and this work is complete. The base idea came from the TOBIAS [73] project. The first work done in this area for VDM was done in [41], this work did not take general expressions into account so Peter Gorm Larsen and I rebuilt the kernel specification in VDM. To optimize the execution speed of the specification, Nick Battle incorporated the feature into the VDMJ interpreter. My main contribution here was to come up with the optimal handling of a front-end GUI and its optimal coupling to the interpreter in particular how to handle failed tests with debugging facilities.

5.1.4 Exploration of Alternative System Architectures

This work is not yet complete but a workshop paper has been published for it [11]. It is envisaged that it will be possible to improve this work and combine it with testing of timed predicates formulated of the traces resulting from executions [74].

The work has been inspired from the work done on combinatorial testing. My contribution is the conceptual idea for this kind of automation and its close relation to [74] which enables different architectures to be checked against timed predicated. This work has been carried out together with Augusto Ribeiro and is now implemented in Overture.

5.1.5 Automatic Transformation between UML and VDM

This work has been published in [12] and this work is conceptually complete but it still needs to be transformed to make use of the new AST. It is anticipated that redoing this can be conducted as a new Masters thesis project that I can take part in supervising.

The original idea for this work came from the UML mapping capability in VDMTools, however this work used the newer UML 2 version. My contribution here was mainly carried out as my Masters thesis [42] where I did the majority of the research and implementation. The tool is now part of the Overture distribution.

5.1.6 Co-Simulation between VDM and 20-sim

This work is only partly finished and it is not yet published. The actual tool coupling between the VDM interpreter and the 20-sim simulator has been released and is used for the DESTECS case studies. However, the description of the semantic foundations here are not yet made and from a tooling perspective it would be good to increase the speed of the co-simulation without adjusting the semantics and this research has not yet been carried out.

This work is primarily based on the idea of co-simulation from [27] and its semantics definitions. The actual tool integration has been done in conjunction with Augusto Ribeiro and Frank Groen from CLP[1]. My main contribution here has been to develop the principles underlying the

[1] Controllap Products, the company behind 20-sim.

co-simulation engine which coordinates the two simulators. The principles has carefully been developed such that it still matches a control engineering discipline.

It is envisaged to submit a paper on the semantics of the VDM and 20-sim co-simulation before the completion of this PhD. It is expected that this publication will be the one that creates most impact from a citation perspective.

5.2 Concluding Remarks

This progress report describes the main activities of the first half of my PhD studies. Three peer-reviewed papers have been published during the first year and a half and in addition four lightly-reviewed papers have been published. A lot of this work has been centered around different kinds of tool supported automations. Most of this has been automation that is visible to the users of the tool (Overture/DESTECS) but it also includes a new form for automation of the internal AST inside the Overture tool in order to achieve a unified AST that all research groups around the world can exploit if they would like to. It is too early to predict if the automation will improve the industrial usage of the formal method VDM.

In addition, I have outlined possible future research directions. Here it is the plan to complete a number of the research directions that are currently ongoing and focus on the underlying semantics of the co-simulation between VDM and 20-sim. In total 4 additional papers are envisaged as listed in Appendix A.

Appendices

My Publications

This appendix provides a list of the different publications that I have made so far in my career.

A.1 PhD Peer-Reviewed Publications

- Connecting UML and VDM++ with Open Tool Support [12]
- Combinatorial Testing for VDM++ [10]
- A Deterministic Interpreter Simulating a Distributed Real Time System using VDM [9]

A.2 PhD Publications with Light Reviews

- The Overture Initiative – Integrating Tools for VDM [6]
- Overview of VDM-RT Constructs and Semantic Issues [75]
- Automated Exploration of Alternative System Architectures with VDM-RT [11]
- Run-Time Validation of Timing Constraints for VDM-RT Models [67]

A.3 PhD Technical Report

- Overture VDM-10 Tool Support: User Guide [76]

A.4 Non PhD Publications

- Optimizing Energy Usage in Private Households [77]
- Facilitating Home Automation Through Wireless Protocol Interoperability [78]

A.5 Planned Future Publications

- Techniques for Combining VDM with Java with Claus Ballegaard Nielsen and Peter Gorm Larsen
- Semantics of the DESTECS Co-Simulation with Augusto Ribeiro, Peter Gorm Larsen, Marcel Verhoef, Ken Pierce and Angelica Mader
- DESTECS Automated Co-model Analysis with Augusto Ribeiro, Peter Gorm Larsen and Frank Groen
- Extendability of ASTs between Plug-ins with Augusto Ribeiro and Marcel Verhoef

Courses Completed

During the first half of my Ph.D. studies I have completed the following courses:

Compiler: A 10 ECTS point course held at the Computer Science department of Aarhus University. A compiler for a sub-set of Java was developed including parser, type checker, static analysis, code generation and code optimization. This course gave a very in-depth understanding of the structure of object-oriented languages.

Modelchecking: A short 2 ECTS point course arranged by the Computer Science department of Aalborg University. The course gave a short introduction to modelchecking with a practical approach where several exercises had to be completed using the SPIN model checker.

Programing Languages A 5 ECTS point course arranged by the Computer Science department of Aalborg University.

I plan to complete the following courses during the second part of my Ph.D:

Academic English for Danish speaking PhD students: 3 ECTS

Summer School in Portuga l- ICCES ERASMUS IP course: 5 ECTS

Types in Object Oriented Languages: 5 ECTS

Bibliography

[1] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal Methods: Practice and Experience," *ACM Computing Surveys*, vol. 41, pp. 1–36, October 2009.

[2] L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An overview of JML Tools and Applications," *Intl. Journal of Software Tools for Technology Transfer*, vol. 7, pp. 212–232, 2005.

[3] A. Martin, "Why effective proof tool support for Z is hard," Tech. Rep. 97-34, Software Verification Research Centre, 1997.

[4] T. Henzinger and J. Sifakis, "The Discipline of Embedded Systems Design," *IEEE Computer*, vol. 40, pp. 32–40, October 2007.

[5] J. F. Broenink, P. G. Larsen, M. Verhoef, C. Kleijn, D. Jovanovic, K. Pierce, and W. F., "Design support and tooling for dependable embedded control software," in *Proceedings of Serene 2010 International Workshop on Software Engineering for Resilient Systems*, pp. 77–82, ACM, April 2010.

[6] P. G. Larsen, N. Battle, M. Ferreira, J. Fitzgerald, K. Lausdahl, and M. Verhoef, "The Overture Initiative – Integrating Tools for VDM," *ACM Software Engineering Notes*, vol. 35, January 2010.

[7] J. F. Broenink, "Modelling, Simulation and Analysis with 20-Sim," *Journal A Special Issue CACSD*, vol. 38, no. 3, pp. 22–25, 1997.

[8] J. F. Broenink, *Computer-aided physical-systems modeling and simulation: a bond-graph approach*. PhD thesis, Faculty of Electrical Engineering, University of Twente, Enschede, Netherlands, 1990.

[9] K. Lausdahl, P. G. Larsen, and N. Battle, "A Deterministic Interpreter Simulating a Distributed Real Time System using VDM," in *ICFEM 2011*, October 2011.

[10] P. G. Larsen, K. Lausdahl, and N. Battle, "Combinatorial Testing for VDM," in *8th IEEE International Conference on Software Engineering and Formal Methods, SEFM 2010*, September 2010.

[11] K. Lausdahl and A. Ribeiro, "Automated Exploration of Alternative System Architectures with VDM-RT," in *9th Overture Workshop, June 2011, Limerick, Ireland*, 2011.

[12] K. Lausdahl, H. K. A. Lintrup, and P. G. Larsen, "Connecting UML and VDM++ with Open Tool Support," in *Formal Methods 09*, Springer-Verlag, November 2009. LNCS-5850.

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

[14] C. B. Jones, *Systematic Software Development Using VDM*. Englewood Cliffs, New Jersey: Prentice-Hall International, second ed., 1990. ISBN 0-13-880733-7.

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

[16] N. Plat and P. G. Larsen, "An Overview of the ISO/VDM-SL Standard," *Sigplan Notices*, vol. 27, pp. 76–82, August 1992.

[17] P. G. Larsen and W. Pawłowski, "The Formal Semantics of ISO VDM-SL," *Computer Standards and Interfaces*, vol. 17, pp. 585–602, September 1995.

[18] A. Ribeiro and P. G. Larsen, "Proof Obligation Generation and Discharging for Recursive Definitions in VDM," in *The 12th International Conference on Formal Engineering Methods (ICFEM 2010)* (J. Song and Huibiao, eds.), Springer-Verlag, November 2010.

[19] J. Fitzgerald and P. G. Larsen, *Modelling Systems – Practical Tools and Techniques in Software Development*. The Edinburgh Building, Cambridge CB2 2RU, UK: Cambridge University Press, Second ed., 2009. ISBN 0-521-62348-0.

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

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

[22] J. Hooman and M. Verhoef, "Formal semantics of a VDM extension for distributed embedded systems," in *Concurrency, Compositionality, and Correctness, Essays in Honor of Willem-Paul de Roever* (D. Dams, U. Hannemann, and M. Steffen, eds.), vol. 5930 of *Lecture Notes in Computer Science*, pp. 142–161, Springer-Verlag, 2010.

[23] J. Fitzgerald, P. G. Larsen, and S. Sahara, "VDMTools: Advances in Support for Formal Modeling in VDM," *ACM Sigplan Notices*, vol. 43, pp. 3–11, February 2008.

[24] P. G. Larsen and P. B. Lassen, "An Executable Subset of Meta-IV with Loose Specification," in *VDM '91: Formal Software Development Methods*, VDM Europe, Springer-Verlag, March 1991.

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

[26] P. G. Larsen, K. Lausdahl, and N. Battle, "The VDM-10 Language Manual," Tech. Rep. TR-2010-06, The Overture Open Source Initiative, April 2010.

[27] M. Verhoef, *Modeling and Validating Distributed Embedded Real-Time Control Systems*. PhD thesis, Radboud University Nijmegen, 2008.

[28] N. Battle, "VDMJ User Guide," tech. rep., Fujitsu Services Ltd., UK, 2009.

[29] P. F. Robin Bloomfield and B. Monahan, "SpecBox: A toolkit for BSI-VDM," *SafetyNet*, no. 5, pp. 4–7, 1989.

[30] R. Elmstrøm, P. G. Larsen, and P. B. Lassen, "The IFAD VDM-SL Toolbox: A Practical Approach to Formal Specifications," *ACM Sigplan Notices*, vol. 29, pp. 77–80, September 1994.

[31] Chiastek, "Cosimate." http://www.chiastek.com, Oct. 2010.

[32] Modelisar, "Modelisar." http://modelisar.org, Oct. 2010.

[33] Berkley, "Ptolemy II." http://ptolemy.berkeley.edu/ptolemyII, Oct. 2010.

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

[35] P. van der Spek, N. Plat, and C. Pronk, "Syntax error repair for a java-based parser generator," *SIGPLAN Not.*, vol. 40, no. 4, pp. 47–50, 2005.

[36] J. P. Nielsen and J. K. Hansen, "Development of an overture/vdm++ tool set for eclipse," Master's thesis, Technical University of Denmark, Informatics and Mathematical Modelling, August 2005. IMM-THESIS-2005-58.

[37] P. G. Larsen, "Ten Years of Historical Development: "Bootstrapping" VDMTools," *Journal of Universal Computer Science*, vol. 7, no. 8, pp. 692–709, 2001.

[38] T. J. H. Christensen, "Extending the vdm++ formal specification language with type inference and generic classes," Master's thesis, Aarhus University, Computer Science Department, April 2007.

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

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

[41] A. S. Santos, "VDM++ Test Automation Support," Master's thesis, Minho University with exchange to Engineering College of Arhus, July 2008.

[42] K. Lausdahl and H. K. Lintrup, "Coupling Overture to MDA and UML," Master's thesis, Aarhus University/Engineering College of Aarhus, December 2008.

[43] D. H. Møller and C. R. P. Thillermann, "Using Eclipse for Exploring an Integration Architecture for VDM," Master's thesis, Aarhus University/Engineering College of Aarhus, June 2009.

[44] I. Hayes and C. Jones, "Specifications are not (Necessarily) Executable," *Software Engineering Journal*, pp. 330–338, November 1989.

[45] N. E. Fuchs, "Specifications are (preferably) executable," *Software Engineering Journal*, pp. 323–334, September 1992.

[46] J. Rushby, "Formal Methods: Instruments of Justification or Tools for Discovery?," in *Nordic Seminar on Dependable Computing System 1994*, (Department of Computer Science), The Technical University of Denmark, August 1994.

[47] J. Rushby, "Disappearing formal methods," in *High Assurance Systems Engineering, 2000, Fifth IEEE International Symposium on. HASE 2000*, IEEE, November 2000.

[48] P. G. Larsen, "Evaluation of Underdetermined Explicit Expressions," in *FME'94: Industrial Benefit of Formal Methods* (M. B. M. Naftalin, T. Denvir, ed.), pp. 233–250, Springer-Verlag, October 1994.

[49] H. Søndergaard and P. Sestoft, "Non-determinism in Functional Languages," *The Computer Journal*, vol. 35, pp. 514–523, October 1992.

[50] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman, "CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution," in *ASPLOS'10*, ACM, March 2010.

[51] J. Burnin and K. Sen, "Asserting and Checking Determinism for Multithreaded Programs," in *17th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, ACM, 2009.

[52] R. Kneuper, *Symbolic Execution as a Tool for Validation of Specifications*. PhD thesis, Department of Computer Science, Univeristy of Manchester, March 1989. Technical Report Series UMCS-89-7-1.

[53] Ö. Babaoglu and K. Marzullo, "Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms," Tech. Rep. UBLCS-93-1, University of Bologna, Piazza di Porta S. Donato, 5, 40127 Bologna (Italy), January 1993.

[54] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, pp. 558–565, July 1978.

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

[56] P. B. Oliver Maury, Yves Ledru and L. du Bousquet, "Using TOBIAS for the automatic generation of VDM test cases," in *VDM Workshop 3* (J. F. J. Bicarregui and P. Larsen, eds.), (Copenhagen, Denmark), Part of the FME 2002 conference, July 2002.

[57] Y. Ledru, L. du Bousquet, O. Maury, and P. Bontron, "Filtering TOBIAS Combinatorial Test Suites," in *FASE 2004* (M. Wermelinger and T. Margaria-Steffen, eds.), (Springer-Verlag Berlin Heidelberg), pp. 281–294, LNCS 2984, 2004.

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

[59] Y. Ledru, F. Dadeau, L. du Bousquet, S. Ville, and E. Rose, "Mastering Combinatorial Explosion with the Tobias-2 Test Generator," in *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, (New York, NY, USA), pp. 535–536, ACM, 2007.

[60] P. Ammann, P. E. Black, and W. Ding, "Model Checkers in Software Testing," tech. rep., NIST-IR 6777, National Institute of Standards and Technology, 2002.

[61] V. Okun and P. E. Black, "Issues in Software Testing with Model Checkers," in *Proc. 2003 International Conference on Dependable Systems and Networks (DSN-2003)* (E. Clarke, M. Fujita, and D. Gluch, eds.), (San Francisco, California), IEEE Computer Society, June 2003.

[62] D. Richard Kuhn and V. Okum, "Pseudo-Exhaustive Testing for Software," in *SEW '06: Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop*, (Washington, DC, USA), pp. 153–158, IEEE Computer Society, 2006.

[63] G. Fraser and F. Wotawa, "Improving Model-Checkers for Software Testing," in *Seventh International Conference on Quality Software (QSIC 2007)*, IEEE, 2007.

[64] P. G. Larsen, "Towards Proof Rules for Looseness in Explicit Definitions from VDM-SL," in *Proceedings of the "International Workshop on Semantics of Specification Languages (SoSL)"*, (25–27 October 1993, Utrecht), Springer-Verlag 1994.

[65] M. Verhoef and P. G. Larsen, "Interpreting Distributed System Architectures Using VDM++ – A Case Study," in *5th Annual Conference on Systems Engineering Research* (B. Sauser and G. Muller, eds.), March 2007. Available at http://www.stevens.edu/engineering/cser/.

[66] M. Verhoef, "On the use of VDM++ for Specifying Real-Time Systems," in *Towards Next Generation Tools for VDM: Contributions to the First International Overture Workshop, Newcastle, July 2005* (J. S. Fitzgerald, P. G. Larsen, and N. Plat, eds.), (School of Computing Science, Newcastle University, Technical Report CS-TR-969), pp. 26–43, June 2006.

[67] A. Ribeiro, K. Lausdahl, and P. G. Larsen, "Run-Time Validation of Timing Constraints for VDM-RT Models," in *9th Overture Workshop, June 2011, Limerick, Ireland*, 2011.

[68] OMG, *Unified Modeling Language UML*, http://www.omg.org/spec/UML/, 2008. OMG Formally Released Versions of UML and ISO Released Versions of UML.

[69] P. G. Larsen, K. Lausdahl, and N. Battle, "Combinatorial Testing for VDM++," in *Submitted for publication*, December 2009.

[70] G. D. Plotkin, "A structural approach to operational semantics," Tech. Rep. DAIMI FN-19, Aarhus University, 1981.

[71] G. D. Plotkin, "The origins of structural operational semantics," *Journal of Logic and Algebraic Programming*, vol. 60–61, pp. 3–15, July–December 2004.

[72] G. D. Plotkin, "A structural approach to operational semantics," *Journal of Logic and Algebraic Programming*, vol. 60–61, pp. 17–139, July–December 2004.

[73] Y. Ledru, "The tobias test generator and its adaptation to some ase challenges," in *Workshop on the State of the Art in Automated Software Engineering*, (University of California, Irvine), June 2002.

[74] A. Ribeiro, K. Lausdahl, and P. G. Larsen, "Run-Time Validation of Timing Constraints for VDM-RT Models," in *9th Overture Workshop, June 2011, Limerick, Ireland*, 2011.

[75] K. Lausdahl, M. Verhoef, P. G. Larsen, and S. Wolff, "Overview of VDM-RT Constructs and Semantic Issues," in *Newcastle Lecture notes, 8th Overture Workshop, 13 September 2010*, September 2010.

[76] P. G. Larsen, K. Lausdahl, A. Ribeiro, S. Wolff, and N. Battle, "Overture VDM-10 Tool Support: User Guide," Tech. Rep. TR-2010-02, The Overture Initiative, www.overturetool.org, May 2010.

[77] J. Rohde, S. Wolff, T. S. T. P. G. Larsen, K. Lausdahl, A. Ribeiro, and P. E. Rovsing, *Towards Green ICT*, ch. 13: Optimizing Energy Usage in Private Households, pp. 185–209. River Publishers, 2010.

[78] S. Wolff, P. G. Larsen, K. Lausdahl, A. Ribeiro, and T. S. Toftegaard, "Facilitating Home Automation Through Wireless Protocol Interoperability," in *WPMC'09: The 12th International Symposium on Wireless Personal Multimedia Communications*, September 2009.

Kenneth Lausdal:
Enhancing Formal Modelling Tool Support with Increased Automation, 2012

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

Tel.: +45 4189 3000