Implementing the Overture Automatic Proof System for VDM

Miguel Alexandre Ferreira
Software Improvement Group
The Netherlands
m.ferreira@sig.nl

October 13, 2009

Abstract

This paper reports on the implementation of the Overture Automatic Proof System as a tool-chain of components. Each component contributes specific functionalities to the tool-chain, enabling the system to discharge VDM proof obligations with the theorem prover HOL.

In previous research work, Vermolen has produced a formal model of a VDM to HOL translator, together with HOL tactics that allow VDM proof obligations to be automatically discharged in the theorem prover HOL. Through the VDM to Java code generator feature provided by the VDMTools, the model was automatically implemented in Java, allowing for its integration in the Overture Automatic Proof System.

The Automatic Proof System is a Java program that enables the interoperation between a VDM proof obligation generation tool, the VDM to HOL translator, and the theorem prover HOL. The current paper reports on the challenges and achievements of the Automatic Proof System's implementation, as part of the Overture Tool framework.

1 Introduction

The Overture initiative is a platform that enables researchers, students and practitioners to experiment with software modelling languages and tools [7]. Although the initiative has mainly focused on VDM, due to Overture's open source nature everyone is welcome to contribute with tools and extensions to other languages as well. Through Overture, extensions to the different VDM dialects and supporting tools have been proposed, analysed, tested and finally transferred to industrial settings, namely to the VDMTools [5]. The language extensions, and respective tool support, that allows for automated generation of combinatorial tests for VDM models [8] is an example of such knowledge transfer to industry.

Another tool that adds to the capabilities of the VDMTools is the Automatic Proof Support (APS), which is able to discharge proof obligations (POs) arising from functional VDM models, using the theorem prover HOL [6]. Due to VDM's formal semantics it is possible to analyse a VDM model and pinpoint the locations where inconsistencies might occur. Such inconsistencies can arise from type invariant violations, misuse of partial functions and mappings, etc. Furthermore, besides pinpointing possible inconsistencies, it is also possible to generate verification conditions that if proven to be true assure the model's consistency. In a VDM context, these verification conditions are deemed POs.

The APS was designed by Vermolen during his MSc project [9]. The deliverables from Vermolen's project that are relevant to the APS implementation are:

- a VDM++ formal model of a prototype tool that translates VDM to HOL (the Overture VDM-to-HOL Translator);
- a Java implementation of the prototype that was automatically generated using the VDMTools;
- a set of lemmas which he identified as useful in a VDM context;
- and a set of HOL tactics to automate the proofs.

In this paper we focus on the development of the core of the APS , and the integration of several external components. The external components were integrated in the system through Java interfaces, allowing for a loose coupling between them and the APS core. Furthermore, the external components were wrapped in utility classes that handle their intrinsic specificities, and were plugged into the APS by implementing the appropriate interfaces. Figure 1 depicts an abstraction of the APS tool-chain, from which the core, wrapper and utility classes, and their integration are the main contributions from this paper. Note that the external components inside the wrappers were previously available.

In remainder of this introduction some work related with the APS is presented. Section 2 describes Vermolen's architecture for the APS, and discusses the difficulties to implement it. Section 3 covers the actual implementation issues. Future work is presented in Section 4, and the paper terminates with some conclusions in Section 5.

1.1 Related work

Related to this work is of course Vermolen's MSc project [9], in which he laid down the first bricks for the APS. The prototype and HOL tactics from Vermolen's MSc project were used in other research to verify POs arising from file system case studies [3, 2, 4]. These case studies not only provided more insight on what were the capabilities and limitations of the VDM to HOL translator prototype, but also what needed to be automated in order to build a proof system for VDM.

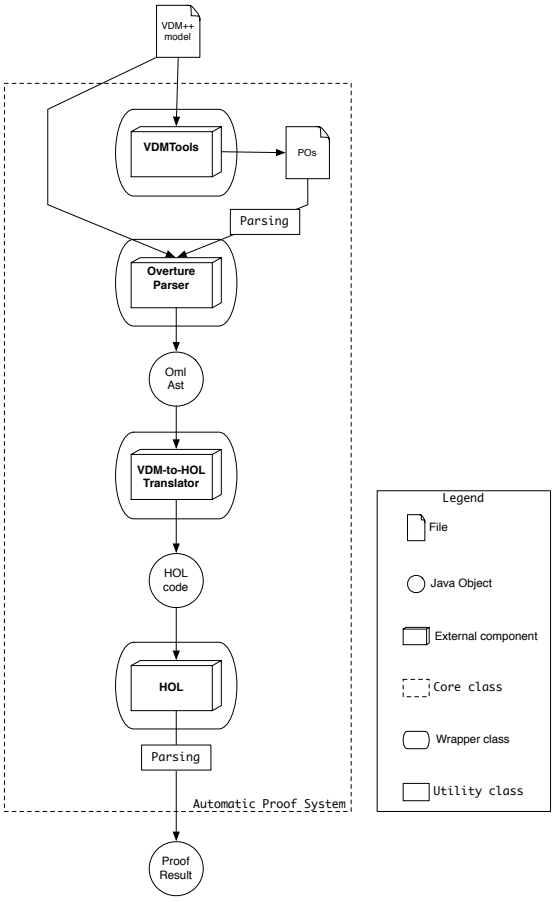

Figure 1: The Automated Proof System tool-chain.

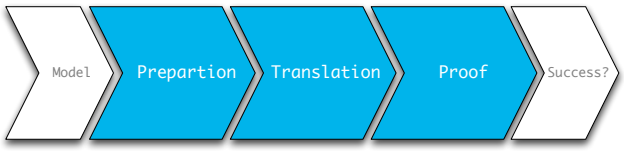

Figure 2: Automatic Proof System workflow as designed by Vermolen.

2 Design

The APS workflow as originally designed by Vermolen divides the system's operation in three sequential steps, as depicted in Figure 2: preparation, translation, and proof.

Preparation. Preparation is the step that converts a VDM model concrete syntax into the abstract syntax tree (AST) format, as expected by the translator tool. It's also in the preparation step that the POs, arising from the model, get generated by a PO generation (POG) tool. The POG performs an analysis of the model's abstract representation and produces the PO expressions in the same abstract format (Figure 3).

Translation. Given both model and POs abstract representation, in the translation step, an abstract representation of an equivalent HOL model is created. The translated HOL model consists of a theory, obtained from the VDM model, and a set of proof commands, obtained from the VDM model's POs (Figure 4).

Proof. In last step of the workflow, the proof, the concrete HOL syntax is generated from its abstract representation and a proof tactic is selected for each PO, according to its type[1]. The last activity in the workflow is the actual proof carried out in the theorem prover (Figure 5).

Following from the described workflow, the architecture of the system is composed of three high level components, one per each step. Each of these high level components can be decomposed in subcomponents, as depicted in Figures 3, 4 and 5.

2.1 Adjustments

The architecture, as described up until now, can be regarded as the ideal architecture for a scenario where all external tools provide the necessary features. However, this is not yet the case and two adjustments have to be made.

The first thing to notice is that, although Vermolen's design makes perfect sense, there is no available combination of POG tool and public VDM AST

[1]There are several types of PO. For more detailed information see [1]

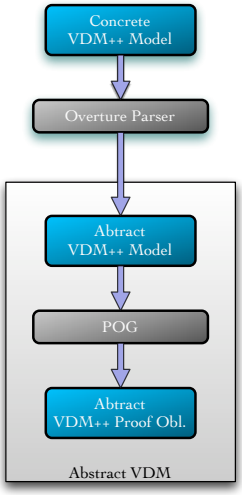

Figure 3: Preparation step workflow as designed by Vermolen. (Diagram obtained from [9].)

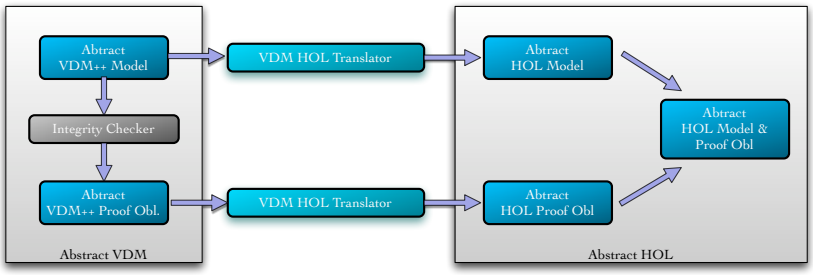

Figure 4: Translation step workflow as designed by Vermolen. (Diagram obtained from [9].)

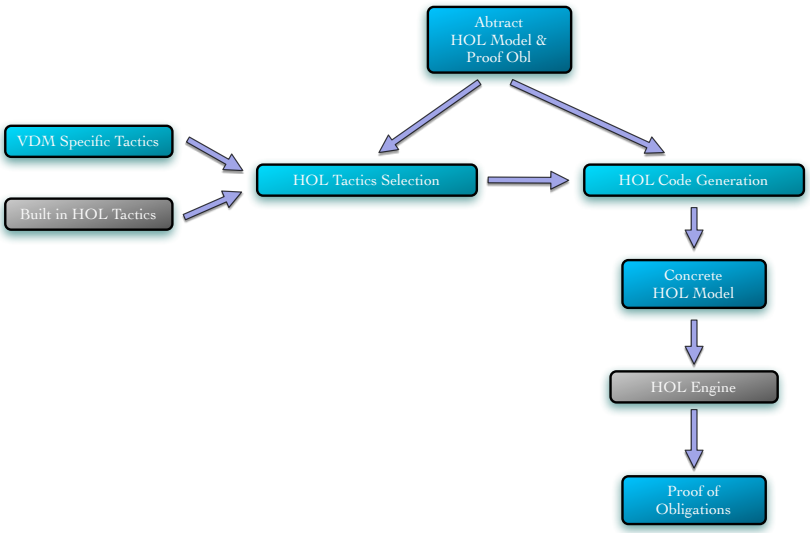

Figure 5: Proof step workflow as designed by Vermolen. (Diagram obtained from [9].)

format to be used in preparation (Figure 3). His design decision was based on Overture's intention to have its own POG tool, which has not yet happened.

There are two POG tools that can be used, both belonging to tool sets that are implemented in monolithic packages: the commercial VDMTools and the open source VDMJ. Moreover, the translation tool, in the translation step, expects both the model and POs to be in the Overture Modelling Language (OML) AST representation, and neither of the POG tools available expects the model to be in this format, let alone generating POs in OML AST. Both POG tools expect the models to be in VDM concrete syntax and produce textual, human readable, representations of the POs. This mismatch between the available tools and the intended architecture makes it necessary to adjust the architecture by introducing the Overture Parser as part of the preparation step, as depicted in Figure 7. In this arrangement the system uses the Overture Parser to generate OML AST representations for the model and each PO individually.

The second adjustment is related with the proof step, more specifically with the HOL concrete syntax generation and tactics selection (Figure 5). Both of these activities are performed by the translation tool (if seen as a black box), and therefore they are performed in the translation step instead of the proof step.

3 Implementation

The APS is a component of the Overture Tool, and therefore must abide by the development standards of the Overture initiative [7]. The Overture Tool is a formal modelling and verification tool suite that is modular, and integrates in the Eclipse platform through several plugins. The main implementation language is Java and the APS is no exception to that.

Figure 1 clearly shows that the APS is mainly a tool-chain of components that contribute to the system's goal. All the computations that are in fact implemented in the APS are either to promote interoperation of the components, or to increase usability.

As far as the integration of components goes the implementation uses two ways to achieve it:

- direct calls from Java code using external component's public APIs;
- or through the command line interface (CLI) of external processes.

Both Overture Parser and the Overture VDM-to-HOL Translator have publicly available Java APIs and therefore can be seamlessly integrated with the APS. As for the generation of POs, the VDMTools was chosen to perform the underlying tasks, and because its CORBA interface doesn't provided the necessary API, the interoperation was done through the CLI. The choice here could have been VDMJ as it offers a Java API that could be called directly from the APS code. However, VDMJ's POG was still under development at the time of the implementation, and the one from VDMTools is much more mature and heavily tested. This doesn't mean that the APS relies exclusively on the VDMTools, because the PO generation and parsing of their textual representations are abstracted by Java interfaces (Figure 7), which provide the desired decoupling from implementations. Although VDMJ is not yet supported as a POG tool for the APS, the system is prepared to allow its integration in the tool-chain (see Section 4 for more information on this subject). The CLI is also used to interoperate with the theorem prover HOL.

The interoperation of the different tools though CLI raises some difficulties in terms of parsing the output of a tool to supply the needed data to the next tool. These difficulties are due to the nature of CLIs, as these are meant to be operated by humans. The most notorious example is the parsing of the list of POs generated by the POG component. If the tool (in this case the VDMTools) would provide a machine readable format, like XML for instance, it would make the implementation much simpler and cleaner. Also the output from HOL must be parsed to check whether the translated HOL code is correct or contain errors, as well as to check the result from a proof.

Interoperation with the CLI of the different components was implemented through a *console* abstraction layer (Figure 6). The *console* mimics the behaviour a human expects from the CLI of a tool, as well as the behaviour the tool expects from its users. This implementation decision helped clear the clutter involved in the interaction with the CLIs, as well as to better structure and

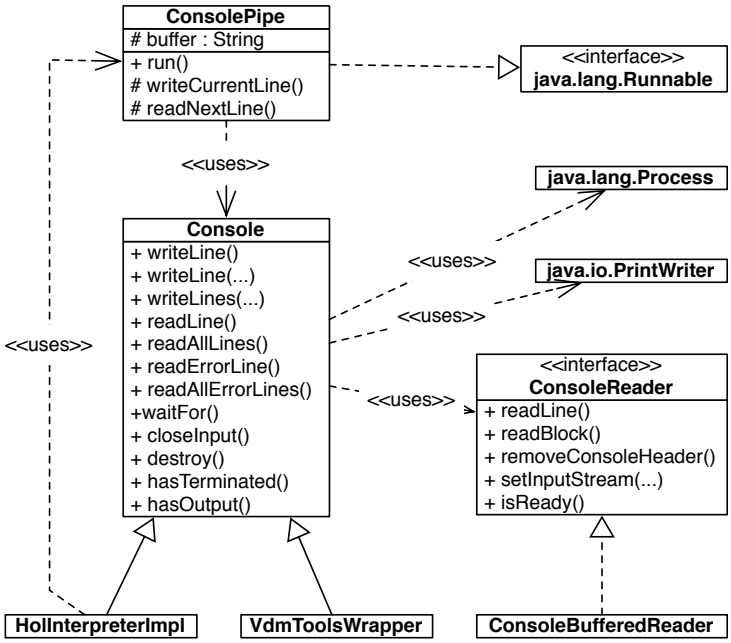

Figure 6: Console abstraction layer class diagram.

re-utilize code. On top of the *console* abstraction layer wrapper objects were implemented for each component, taking advantage of methods the layer provides and adding more, component specific, features. Furthermore all external component wrapper implement a specific interface to interact with the APS core (Figure 7).

Usability issues also arise from the experience obtained using the system as it was still a prototype. On the one hand, sometimes the VDM specific tactics are not enough to discharge a PO. On the other hand, sometimes the proof attempts may take a very long time which the user is not willing to wait. So if the APS was able to detect a failing, or non terminating, proof and allowed the user to try a different tactic, it could increase the overall success rate of the system (see Section 4 for more information).

3.1 Difficulties

The two main difficulties encountered during the implementation of the APS were the lack of up to date documentation for some components, and the way Java handles external processes. These difficulties are described next.

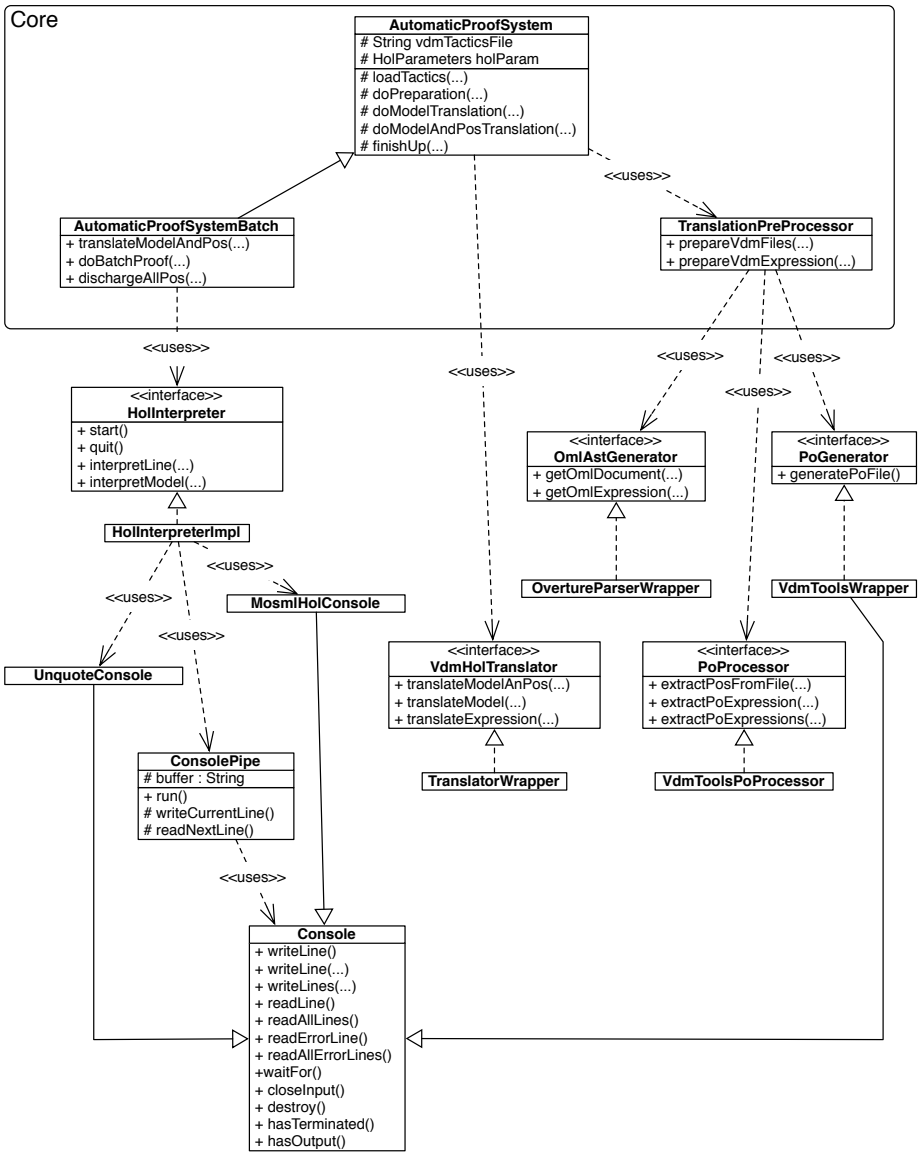

Figure 7: Automatic Proof System class diagram.

3.1.1 Documentation

Documentation, or the lack of it, was one of the major drawbacks during implementation, even regarding components that are part of the Overture initiative. For instance the Overture Parser has had the capability to parse a single VDM expression separated from the context of a model for a long time now. This feature is what enables the APS system to generate OML AST representations for each PO individually. However, during the implementation of the APS the development was stalled for some weeks because there was no knowledge of this feature. The same goes for the APS, as no documentation whatsoever was created apart from Vermolen's MSc thesis, and this paper.

It is not only on the open source side that documentation lacked. The APS uses the VDMTools to generate the POs through the CLI. However the flag that is used to trigger the PO generation in the VDMTools is also not documented. Neither in the usage information provided through the CLI, nor the actual manual of the tool. It was only through the close connection between Overture and CSK[2], the current suppliers of the VDMTools, that this information was obtained.

3.1.2 External processes

Other major drawback for the implementation of the APS was the way Java handles external processes, and especially child processes[3] of the processes one instantiates within a Java application. In a Java application if a process, say A, is created and in its turn creates a child process, say B, then, if A is terminated and the calling application invokes a wait for method on A before B has (somehow) terminated, then the method issued on A will never return. This means that the wait for method does not return until all child processes are terminated. However, if a destroy method is called on A it's child processes will not be destroyed, as it is the responsibility of the parent process to terminate its child processes. This behaviour is well documented in the Java bug/issue tracking system[4].

The theorem prover HOL, in its version 4, is built on top of the Moscow ML[5] interpreter which is an implementation of the Standard ML language interpreter. This means that HOL runs inside the Moscow ML interpreter. HOL's executable is in fact a script that pipes its input to a first command, named *unquote*, which basically replaces the quotes in HOL concrete syntax by machine readable markers. The output of *unquote* is then re-piped to the *ML interpreter*. So whenever the HOL executable is invoked from the command line two child processes are also invoked, one for *unquote* and another for the *ML interpreter*. This is the case both in Windows™ and Unix™ alike platforms.

It is a good programming practice to wait for the processes to finish. However, in this case after issuing the HOL process termination command (by sup-

[2] `http://www.csk.com/index_e.html`.

[3] A child process is a process that is created within another process.

[4] For more details see `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4770092`.

[5] `http://www.itu.dk/~sestoft/mosml.html`.

plying it with the proper string), the method that waits for it to finish enters a deadlock state. On the other hand, if a forced destruction of the HOL process is issued, the result is even worse. One of HOL's child processes starts to allocate all the memory it can get, until the machine becomes unusable.

To overcome this problem, the implementation never calls HOL's executable directly. Instead, it creates two independent processes, one for *unquote* and another for the *ML interpreter*. These two processes are encapsulated using the already mentioned *console* abstraction, and a new thread is created to pipe the output of the first process to the input of the second (Figure 6). This way both processes are directly managed by the APS, thus avoiding the unwanted behaviour of Java.

3.2 Current Status

The APS is currently in its version 1, and is only available as a command line application. It uses the Args4j[6] library to parse command line parameters and arguments.

Testing was done through the jUnit test framework, and 69 unit tests were created to monitor the APS code. These unit tests amount to a code coverage of 78.2%, covering all major components and their interactions.

There are two execution modes: *translation* that simply translates the VDM model and its POs to HOL syntax; and *proof* that not only translates the model but also attempts to discharge the POs directly in HOL. The execution of the tool in either mode is done in a batch, meaning that inputs are supplied to the tool and it produces an output.

4 Future Work

Although the APS is useful as a standalone application, the objective is to have it integrated in the Overture Tool as an Eclipse plugin. This integration can be done in two steps. First through a minimalistic plugin that acts as a "console" for the APS, simply showing the user the results from each PO proof attempt. If all goes right with the first, and minimalistic, plugin then a more sophisticated approach could be taken. This approach would consist in having a graphical user interface (GUI) for the APS, where POs would be displayed in a table aggregating the information of automated proof attempts. In the case that some POs are not automatically discharged, the GUI would give the user the opportunity to assist the theorem prover.

The generation of the POs is an open issue in the context of Overture since there are two available POG tools, but none works at the AST level. This fact implies unnecessary parsing and diminishes the interoperability between components. The APS doesn't yet support VDMJ as a POG generator, although it is prepared for that, and this is also something to be done in the upcoming releases.

[6] `https://args4j.dev.java.net`

Proofs in HOL might not terminate in feasible time. Therefore the APS must be equipped with, configurable, timers to detect those situations, and abort the proof if necessary.

Given the current multi-core trend in computer hardware, the APS could take advantage of multi-core architectures to launch proofs in parallel, and with this improve its performance.

Although the translation of VDM syntax in HOL syntax is the responsibility of other Overture component, this is crucial for the usability of the APS. At the moment the APS, even if it was already integrated in the Overture Tool through Eclipse, wouldn't bring so much value to VDM modellers because of the quite small subset of the language supported by the translator.

There are also problems in the translation that were already identified by Vermolen in his thesis [9, Section 7.3] but remain unresolved. Some of these issues result in syntax errors in the produced HOL code.

5 Conclusion

The APS is basically a tool-chain of components that when properly interoperated work together to discharge VDM POs. Its implementation was not always straight forward but it was still successful. A Java program was developed enabling VDM users to automatically:

- prepare a VDM model for the translation to HOL;
- collect the resulting (HOL) code ans interpret it with the HOL theorem prover;
- collect the results produced by HOL and provide the user with a summary of the POs that were, and were not, discharged.

Vermolen's effort in the formal specification of the translation tool and design of the workflow of the system had a very positive impact in the actual implementation. Implementing a system like the APS following carefully laid out guidelines certainly reduces the time and cost, as opposed to starting to implement it straight away. Furthermore, the value of the VDM++ formal prototype transcended the prototype's function in the design phase, as the actual Java code, that is being used by the APS, was automatically generated from it.

In conclusion, the basic functionality as designed and formally prototyped by Vermolen is now implemented, but there is still room for improvement.

References

[1] CSK. *The Integrity Checking: Using Proof Obligations*, 2007.

[2] M. Ferreira. Verifying Intel® Flash File System Core. Master's thesis, Minho University, Jan. 2009.

[3] M. Ferreira, S. Silva, and J.N. Oliveira. Verifying Intel Flash File System Core Specification. *Fourth VDM/Overture Workshop*, (CS-TR-1099), May 2008.

[4] M.A. Ferreira and J.N. Oliveira. An Integrated Formal Methods Tool-Chain and its Application to Verifying a File System Model, 2009. Accepted for publication.

[5] J. Fitzgerald, P.G. Larsen, and S. Sahara. VDMTools: advances in support for formal modeling in VDM. *SIGPLAN Notices*, 43(2):3–11, 2008.

[6] Michael J. C. Gordon. Introduction to the HOL System. In Myla Archer, Jeffrey J. Joyce, Karl N. Levitt, and Phillip J. Windley, editors, *TPHOLs*, pages 2–3. IEEE Computer Society, 1991.

[7] P.G. Larsen, N. Batle, M. Ferreira J. Fitzgerald, K. Lausdahl, and M. Verhoef. The Overture Initiative Integrating tools for VDM, 2009. In preparation.

[8] P.G. Larsen, K. Lausdahl, and N. Battle. Combinatorial Testing for VDM++. In *Submitted for publication*, October 2009.

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