Towards Enabling Overture as a Platform for Formal Notation IDEs

Luís Diogo Couto   Peter Gorm Larsen   Miran Hasanagić   Georgios Kanakis
Kenneth Lausdahl   Peter W. V. Tran-Jørgensen

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

`{ldc,pgl,miran.hasanagic,gkanos,lausdahl,pvj}@eng.au.dk`

Formal Methods tools will never have as many users as tools for popular programming languages and so the effort spent on constructing Integrated Development Environments (IDEs) will be orders of magnitudes lower than that of programming languages such as Java. This means newcomers to formal methods do not get the same user experience as with their favourite programming IDE. In order to improve this situation it is essential that efforts are combined so it is possible to reuse common features and thus not start from scratch every time. This paper presents the Overture platform where such a reuse philosophy is present. We give an overview of the platform itself as well as the extensibility principles that enable much of the reuse. The paper also contains several examples platform extensions, both in the form of new features and a new IDE supporting a new language.

1 Introduction

The Vienna Development Method (VDM) is one of the most mature Formal Methods (FM) [26, 15]. The method focuses on the development and analysis of a system model expressed in a formal language. The formality of the language enables developers to use a wide range of analytic techniques, from testing to mathematical proof, to verify the consistency of a model and its correctness with respect to an existing statement of requirements. The VDM modelling language has been gradually extended over time. Its most basic form (VDM-SL), standardised by ISO [29] supports the modelling of the functionality of sequential systems. Extensions support object-oriented modelling and concurrency (VDM++) [16], real-time computations [36] and distributed systems (VDM-RT) [43, 42]. All these dialects of VDM are supported by the Overture platform [30].[1]

The mission of the Overture open-source project is twofold:

- 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.

As is the case with other FM tools, the Overture Integrated Development Environment (IDE) consists of a common Abstract Syntax Tree (AST) representing the model and various plug-ins providing the different kinds of analysis available in VDM as shown in fig. 1. The broad variety of analysis possible is common in many formal methods. In such cases, it is important to ensure that all analyses are implemented in a consistent way to facilitate maintenance. Such consistency would also aid in the development and integration of new functional extensions.
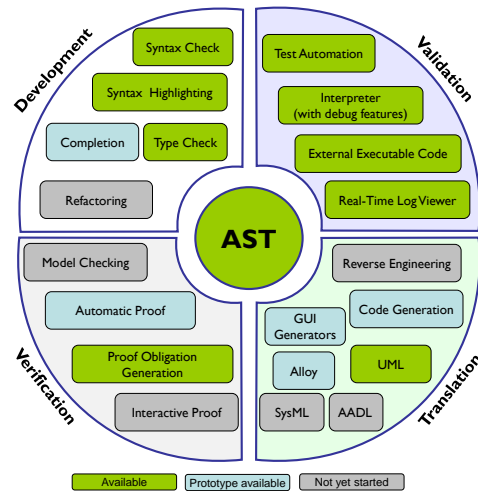
---

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

Figure 1: Overture Tool Components

In addition to the above, this platform-based architecture allows for the reuse of common features across all extensions. This reuse can be taken further by supporting language extensions that would allow other formal notations to reuse the same platform. Recently, Overture has been re-factored to enable such a reuse [8, 12]. The main contribution reported in this paper is the Overture platform itself and its extensibility principles which are described in sections 2 and 3. An extensible platform facilitates the development of new features for an IDE and in section 4 we demonstrate how several features of the Overture IDE have been developed on top of the platform. Furthermore, in section 5 we demonstrate how the platform has been integrated with an external tool.

The extensibility principles of the Overture platform also affect the notation itself. The platform is capable of supporting a base language (VDM in the case of Overture) as well as multiple notation extensions. This allows for the development of IDEs for new notations with heavy reuse of common features. Section 6 describes one such IDE, which also includes integration with several external tools.

Open issues remain in the platform, most notably in terms of integration with external tools. Section 7 lays out future work for addressing some of these issues and also summarises the paper. It is our hope that this paper demonstrates the advantages of platform-based IDE development and that it can be beneficial for multiple FM tool builders to share a common platform.

Other examples of FM platforms with comparable functionalities include the Asmeta tool set for ASM [4, 3], the Rodin platform for Event-B [1, 2, 14] or TLAToolbox for TLA$^+$ [28, 41]. The extension philosophies of the software tools differ as do the actual extensions that are available. A detailed comparison is beyond the scope of this article, but more information about these platforms and the modelling languages they support can be found by following the references provided. The general philosophy of reuse has also been employed effectively for theorem provers [39].
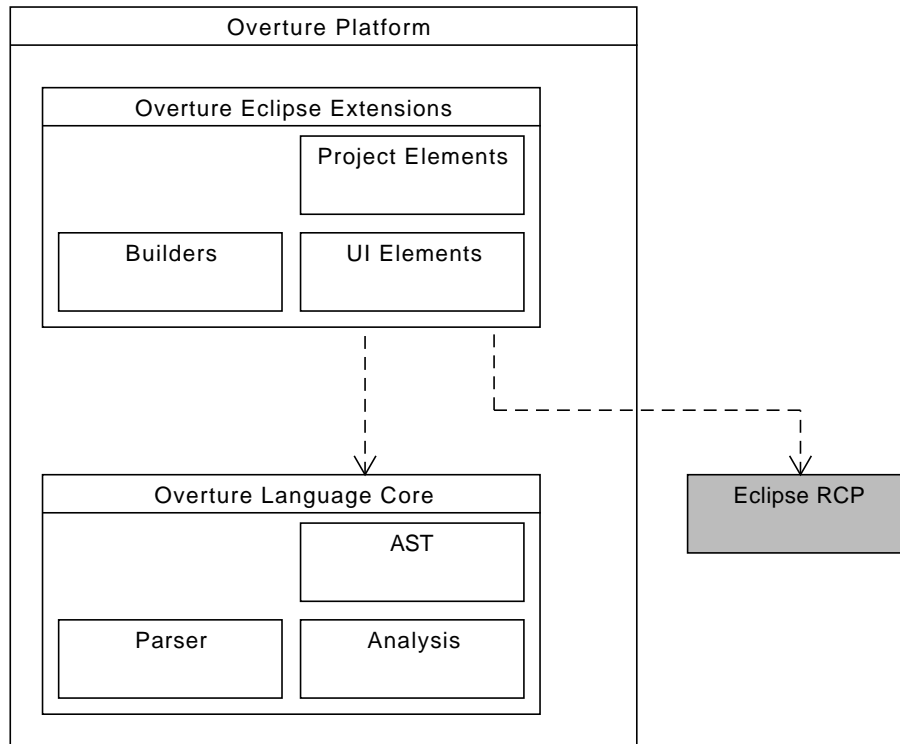
Figure 2: The Overture Platform.

2 The Overture Platform

2.1 Overview

The Overture platform supports the development of FM IDEs. It was originally developed to support the development of the Overture IDE for VDM but has since evolved into a more general platform. It is comprised of two parts: the *Overture Language Core* and the *Overture Eclipse Extensions*, as shown in fig. 2.

2.2 Overture Language Core

The language core encapsulates and handles any language and notation-related concerns, including parsing, representation and analysis, in order to facilitate decoupling between the core language and User Interface (UI) implementations. In addition to the general benefits of separation of concerns, the language core also opens the possibility of migrating the IDE implementation to another UI technology as well as providing the base tool functionalities for command line access, batch processing or as an external tool to be accessed by others.

The language core consists of an extensible AST that is automatically generated by the *AstCreator* tool[2], as well as a parser for constructing the AST from model sources. In addition, *AstCreator* also

---

[2]See `http://github.com/overturetool/astcreator`

generates machinery for traversing and processing trees in a consistent way in the form of a visitor framework [21]. Any kind of analysis of the AST such as type checking or interpretation should be implemented using the visitor framework.

One of the key features of the language core is its extensibility mechanism which allows language extensions or new languages to be implemented in the Overture platform while reusing as much existing code as possible. This mechanism is described in further detail in section 3

2.3   Overture Eclipse Extensions

The Overture platform also consists of a set of extensions to the Eclipse Rich Client Platform (RCP) that are used to help build the UI components of the IDE. The Eclipse RCP is a generic framework for building rich client applications using the Eclipse OSGi plug-in model and UI toolkits. It is is powerful and generic but comes with a cost: significant amounts of boilerplate source code and configuration files must be written in order to prepare it to build an IDE.

The Overture Eclipse extensions automate some of the configuration and preparation work by providing the aforementioned boilerplate code targeting FM notations. The extensions provides an extensible application framework on top of the RCP. It significantly reduces the amount of code that needs to be written in order to contribute an extension to the IDE. To put it another way, the RCP API is very wide and the Overture Eclipse Extensions summarise a portion of it, thus giving developers faster access to the functionality at the cost of some flexibility. However, the Overture extensions are fully interoperable with the RCP so any other extension that requires direct access to the RCP can still be used.

There are other frameworks similar to the Overture extensions in the Eclipse project, such as the Dynamic Languages Toolkit (DLTK) [13] and *Xtext* [45]. DLTK is designed to support the implementation of IDEs for dynamic programming languages and *Xtext* is designed to support the implementation of IDEs for for Domain-Specific Languages (DSLs) or small programming languages. Neither framework is particularly suitable for VDM – VDM is similar in notation to a statically typed general-purpose programming language – which was the original target language to be supported by the Overture IDE.

Broadly speaking, the Overture Eclipse extensions can be divided into three groups:

- a set of UI elements for editors, launch configurations, etc. that interact directly with the Eclipse RCP.

- a set of project elements that represent the FM model and associated concepts such as source units, according to the Eclipse project model. Also included are connectors and providers for accessing these various entities from within the IDE.

- a set of builders that interact with the language core in order to process language sources to construct an internal representation of the model and load it into the project elements.

Both the builders and the project elements are developed according to standard Eclipse conventions so that new versions of these packages for other notations may be contributed.

Currently, the Overture Platform primarily supports the Overture IDE. The Overture IDE is comprised of Eclipse plug-ins that use components implemented with the language core to perform analysis of the VDM AST and UI components that wrap the analysis and use the Eclipse extensions to implement the interaction with the user.

3   Extension Principles of the Overture Language Core

The basic principles of extensibility in the Overture language core are related to the generation of ASTs from specification files, similar to parser generators like SableCC [20]. In addition to generating the classes representing the tree structure, it is important to generate auxiliary machinery to allow developers to implement analysis of the AST in a consistent manner.

The main way to construct extensions in the language core is by extending the AST. Generally speaking, an AST is extended by adding new subtrees that are either entirely new or that contain some existing base nodes. In addition, the extended tree needs to reuse the existing base node classes wherever possible.

In addition to extending the tree itself, it is important to also extend the analysis machinery. Particularly, this extended machinery needs to be able to analyse trees made up of extension and base nodes. Furthermore, the extended analysis machinery needs to reuse the base machinery when processing base nodes – this is essential for achieving reuse of functionalities already implemented as base analysis.

Whether speaking of a tree made of only extension nodes or a hybrid tree with extension and base nodes or even a base tree, the AST classes have a limited ability to enforce the structure of each particular instantiation of the tree. It is the syntax of the language, as encoded in the parser, that ultimately controls which trees are admissible. Along the same lines, it is the parser that controls which base nodes are reused when constructing hybrid trees as the extended tree specification can only set an upper limit on this.

The extensibility principles of the Overture language core are primarily realised through the *AstCreator* tool. *AstCreator* provides provides an automated way of generating trees and auxiliary machinery from specification files as shown in fig. 3. The tool is capable of taking an existing AST as well as an extension specification and generating the extension nodes and visitor framework upon which to implement analyses.[3] Both nodes and visitors are aware of the base classes thus ensuring interoperability with the base trees.

It is also possible to use *AstCreator* to build a completely new AST supporting a language that is unrelated to VDM (see section 4.1 for an example). In this case, a new base tree and visitor framework will be produced and it will not be possible to reuse existing components of the language core. As such, we focus on the case where the new language being supported is an extension of an existing notation where it is possible to reuse parts of the base AST, and the corresponding analysis. Typically, constructs like arithmetic or logical expressions or imperative statements can be reused. This leads to hybrid trees where nodes from the base and extended trees are blended together.

The semantics of such a language extension should be implemented as various AST analyses such as type checking or interpretation. Each analysis should be implemented as an independent component that processes the tree in a consistent way. The visitor framework that is generated as part of the extension provides a way to achieve this. Since the visitor framework itself is extension-aware it enables selective and controlled reuse of existing base analyses as necessary. The extension-aware visitor is illustrated in fig. 4.

An example of these extension principles at work can be seen in the Symphony IDE, as described in section 6.

---

[3] *AstCreator* is also capable taking a base and extension specification and producing both sets of classes, though this is done less frequently.
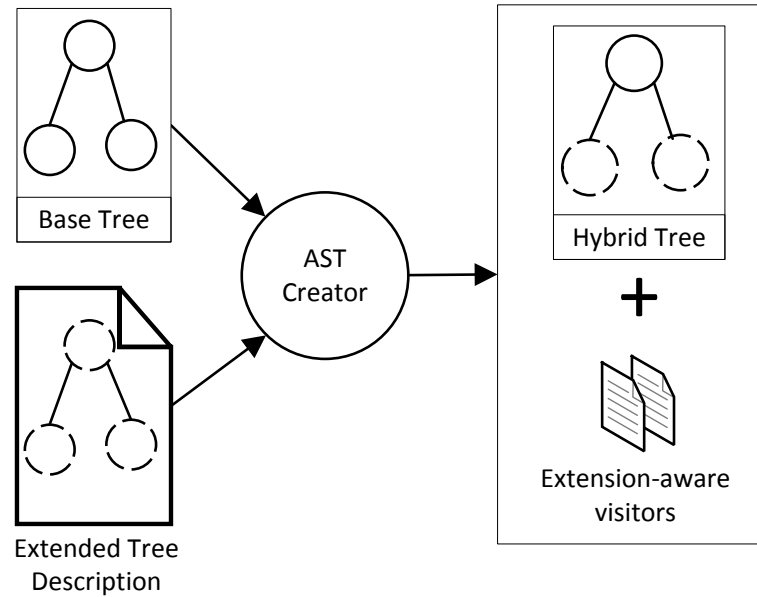
Figure 3: Extending AST specifications.

4   Functionality extensions

New functionality can be contributed in the Overture platform either by using the language core, the Eclipse extensions or a combination thereof. The language core provides the necessary mechanisms to interact with the AST as well as extending it, whereas the Eclipse extensions provide the means to expose functionality to the user. In this section, we provide examples of how both can be used to add new functionality to Overture.

4.1   The code generation platform

The code generation platform aims to facilitate integration of VDM code generators into Overture with minimum effort [27]. Like many other Overture components, the code generation platform interacts with the language core by analysing a type checked VDM AST in order to generate code in some target language. Currently, the code generation platform is used to develop VDM code generation support to Java and C++, and in addition, there is ongoing work on generating Isabelle/HOL syntax [11].

In order to promote reuse the code generation platform works with an Intermediate Representation (IR) of the generated code, which is independent of any particular target language. In addition, the code generation platform provides mechanisms for rewriting or *transforming* the IR into a semantically preserving form that is easier for a particular backend to code generate. Furthermore, since transformations work directly on the IR it becomes easier for different backends to use and contribute new
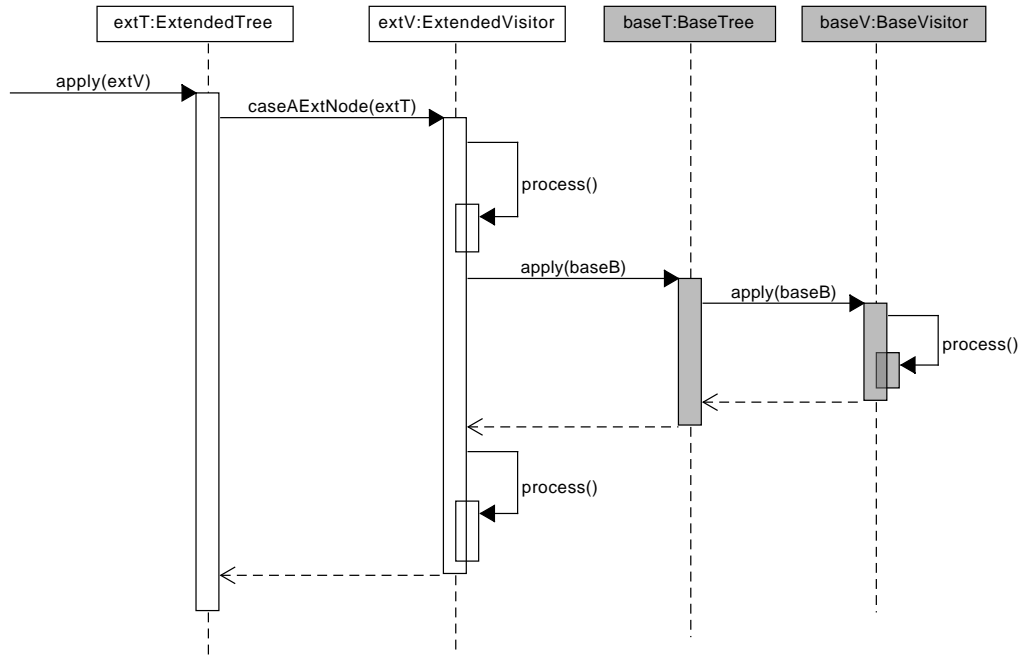
Figure 4: Extended AST and analysis.

functionality to analyse and modify the IR.

The code generation platform allows new nodes to be added to the IR as well as extending existing nodes with additional fields as enabled by *AstCreator*, which is used for the specification of the IR. The Isabelle/HOL code generator exploits this, since mutually recursive functions must be grouped explicitly in Isabelle/HOL and therefore the code generator adds function groups to the IR. This is done in a supplementary tree extension file and demonstrates how users of the code generation platform can extend and change the IR as needed. Although most of the work involved in developing code generation support includes traversing and transforming the IR, thus interacting with the language core, the Eclipse extensions provide the necessary mechanisms to read preferences and configure the code generation process.

4.2 Interpreting implicit specifications using ProB

In VDM functions and operations can be either explicitly or implicitly (using pre and post conditions) defined. An explicit description defines how the output is obtained from the input, which enables the description to be evaluated directly in the VDM interpreter [33]. Implicit descriptions, on the other hand, only specify the constraints that must be met but without defining how the output is obtained. Therefore, attempting to evaluate an implicit description in the interpreter yields a runtime error. To avoid restricting analysis of implicit descriptions to static analysis only, Overture has recently integrated the ProB constraint solver in order to enable evaluation of implicit descriptions [32].

Interpretation of implicit descriptions adds an additional step to the model execution where the pre and post state as well as the constraints imposed by the implicit description, is converted to ProB syntax to form a formula, which is submitted to the ProB constraint solver. This formula is constructed as a

string by analysing the type checked AST.

If ProB is able to find a solution to the given problem the solution is converted back to VDM format and used throughout subsequent execution of the model. Intercepting the interpretation of implicit descriptions is primarily enabled through extension of the language core, and interacting with ProB via an external Java API.

4.3   VDMTools integration

VDMTools [17] is an industrial strength IDE, maintained by the SCSK corporation, for analysing models written in VDM. Among many features also supported by Overture, VDMTools provides extensive semantic checking, execution and Java/C++ code generation of models written in VDM. To facilitate use of different IDEs, Overture provides an option to export an Overture VDM project to a VDMTools compatible format. This plugin is developed through the Eclipse extensions that are used to convert meta data from an Overture project to a format compatible with VDMTools.

4.4   Combinatorial Testing

Combinatorial Testing (CT) in VDM provides automated generation and execution of a large collection of tests as an extension to the language core functionality [31]. The addition of CT in Overture has trigged several changes to the language core components. First, the AST was extended to support the trace nodes. Second, the type checker was updated to support type checking of both traces and generated tests. Finally, the interpreter was extended to support trace expansion as well as test execution.

In addition to extending the language core a CT view has been added to Overture as a new Overture Eclipse plugin extension. This plugin serves to provide a convenient way for users to inspect the test execution results, filtering large collections of tests in order to obtain a reduced representable subset of tests, and re-executing tests individually.

5   Building a Co-Simulation tool with the Overture Platform

The Crescendo tool supports collaborative modelling and co-simulation of Cyber Physical Systems (CPSs) [18], and has been developed by extending the Overture platofrm. This extension enables co-simulation between co-models, which are composed of a discrete time model described in the VDM-RT language, and a continues time model described using differential equations. The extension is composed of a co-simulation engine that connects an extended version of the VDM interpreter [33] from the Overture tool with the simulator in 20-Sim [9].

The Crescendo tool primarily extends the Overture platform using ordinary Eclipse extension points for: builders, debug related UI and views. However, it also uses Overture Eclipse extensions for e.g. editors, and debugging related components.

An extension was also made to the language core by extending the VDM interpreter used for evaluating and debugging with two main features: *a*) the ability to only simulate until a certain time bound, and *b*) the ability to detect when a shared co-simulation variable is accessed. This is necessary in order to support co-simulation such that the two simulators can synchronize their time steps.

6 Building a new IDE with the Overture Platform

Thus far, this paper has shown how to contribute extensions to the Overture IDE and how to extend existing components to support co-simulation with an external tool. These examples consist of extensions that either make very small or no changes to the VDM language, in terms of new syntax, the semantics thereof or the concepts introduced. This makes the extensions relatively simple to support in comparison to an extension for a new full-blown FM notation, especially considering the wide variety of formal notations as well as their associated semantics, paradigms and problem domains.

It is possible to use the Overture platform to build an IDE for a new notation that shares nothing with the VDM language. However, this means that the new IDE will be unable to reuse much of the language core since the AST and associated analyses will be entirely different. On the other hand, when building an IDE for a notation that reuses or shares parts of VDM, then the relevant parts of the language core can be reused. The remainder of this section shows how such reuse was achieved in the construction of the Symphony tool [7] in the COMPASS EU FP7 Project. Symphony supports the COMPASS Modelling Language (CML) notation [44] that was introduced in the COMPASS project and combines VDM with Communicating Sequential Processes (CSP) [22].

The syntax of CML differs significantly from that of VDM, especially as it relates to the new constructs inherited from CSP. As such, it was necessary to construct a parser to recognize CML notation. Tools such as ANTLR [38] greatly aid in parser construction and Symphony has an ANTLR parser built from scratch that processes CML sources to construct ASTs that are compliant with the Overture language core.

The static analysis of CML ASTs (type checking and proof obligation generation) significantly reuses relevant Overture components [10]. In the case of proof obligations, reuse led to reduction in lines of code from 2596 to 978 as well as a reduction in duplicate code from 37.2% to 3.1%. In general, any existing analysis for VDM was reused whenever possible. A good example lies in the processing of VDM expressions inside CSP actions – also an example of hybrid tree processing.

The validation of CML models could not reuse Overture components so easily since the paradigms of CML notation are different from those of VDM. In particular, CML is a process algebra and its models are interpreted as sequences of events, as opposed to VDM's imperative approach based on state transformations.

Due to the difference in paradigms between the languages, significant portions of the Symphony interpreter had to be built from scratch. However, in spite of the differences in behaviour, the Symphony interpreter still manages to reuse the Overture one for evaluating expressions and reused statements.

For all cases of reuse in Symphony, the same basic principle applies: the extended analysis processes the hybrid tree and when it encounters a base node, it submits the node to its counterpart base analysis, with a mechanism in place for the extension to re-assume control and preventing the base analysis from hijacking the analysis of the remaining tree.

Finally, it is important to discuss the underlying semantics of the various analyses as it should be ensured that consistent semantics are in place across all components of the tool, lest errors be introduced in the overall results due to gaps between the various semantics. In the COMPASS project this was addressed by using the Unifying Theories of Programming [23] that provides a common framework for the various semantic models used in the project. This work eventually led to a mechanisation of a subset of CML in Isabelle [19].

Most functionalities of the Symphony IDE were implemented as Eclipse plug-ins using a combination of the Overture language core (exported via a counterpart Symphony core) and the Overture Eclipse extensions (used to build the main UI components of the Symphony IDE). In addition to

its native functionalities, the Symphony IDE also uses its various plug-ins to integrate externals tools such as Maude [5, 6], Isabelle [37] (via Isabelle/Eclipse [24]), FORMULA [25], RT-Tester [40] and ProB[34, 35]. The most relevant external tool integrations are shown in fig. 5. Note how this aims at following the principle of reusing existing functionality rather than re-developing from scratch.
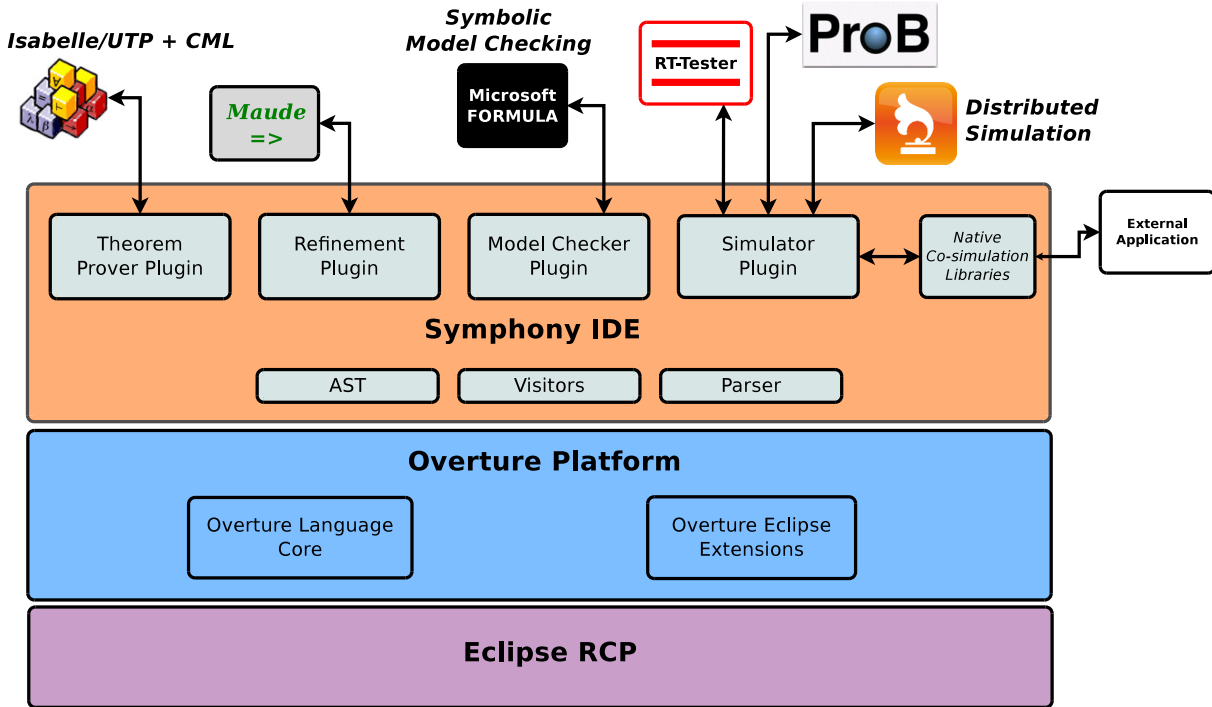

Figure 5: The COMPASS tools

7   Concluding Remarks and Future Work

This paper has described the Overture IDE and its underlying platform. We have shown the extensibility principles of the platform and demonstrated how they support multiple functional extension plug-ins. Furthermore, we have demonstrated how the platform can support notation extensions and, as such, be used as a basis platform by other FM tool builders. The ability to reuse existing functionality and build on the work of other teams can help improve the quality of FM tools in general.

Going forward, there are various potential improvements that can be made to the Overture platform and we discuss a few of them here. The first improvement is in terms of the *AstCreator* tool's specification files. At the moment, *AstCreator* is only capable of generating the Java code for the AST from a fairly simple tree specification file. This is by design. *AstCreator* does not aim to address issues of parsing when tools such as ANTLR already do an excellent job of it. However, it may be beneficial to integrate *AstCreator* with parser generators. Either by deriving an *AstCreator* specification file from the parser generator grammar or by creating a stub grammar from the *AstCreator* specification.

Another potential improvement lies in making more use of the code generation platform when integrating external tools. Integration with external tools often consists of translating the VDM syntax into

that of the external tool and submitting it to the tool as is done for example in the ProB integration. These translations are often implemented manually using the visitor framework. However, by using the code generation platform, significant gains may be attained in terms of the amount of code that is necessary. We are currently undertaking work in this direction and early results are very promising [11].

The final improvement under consideration is also related to external tool integration, but is a somewhat open-ended question at the moment. Integration of external tools is currently done on a case-by-case basis. Each external tool is integrated in its own way with entirely handwritten code. While the syntax translation issue may be addressed, the invocation of the external tool, passing of data to it and collection of results is completely non-standardized. This is mostly a consequence of all external tools having different ways of accessing them. However, at a high-level, most external tool interactions can be reduced to a general case such as external command invocation, protocol-based communication or API access. It would be beneficial to have mechanisms in the platform to help deal with each of the general cases. Another alternative would be a methodological approach where guidelines are produced to help developers implement each kind of integration in a consistent manner.

Acknowledgments

References

[1] Jean-Raymond Abrial (2010): *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, doi:10.1017/CBO9781139195881. Available at `http://www.cambridge.org/uk/catalogue/catalogue.asp?isbn=9780521895569`.

[2] Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta & Laurent Voisin (2010): *Rodin: an open toolset for modelling and reasoning in Event-B*. STTT 12(6), pp. 447–466, doi:10.1007/s10009-010-0145-y.

[3] (2015): *The Asmeta tool set for ASM*. `http://asmeta.sourceforge.net`.

[4] Egon Börger & Robert F. Stärk (2003): *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, doi:10.1007/978-3-642-18216-7.

[5] M. Clavel, F. Durn, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer & J. F. Quesada (1999): *The Maude System*. In: *Rewriting Techniques and Applications*, Springer, LNCS1631, doi:10.1007/3-540-48685-2_18.

[6] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer & Carolyn L. Talcott, editors (2007): *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. Lecture Notes of Computer Science 4350, Springer-Verlag, doi:10.1007/978-3-540-71999-1.

[7] Joey W. Coleman, Anders Kaels Malmos, Peter Gorm Larsen, Jan Peleska, Ralph Hains, Zoe Andrews, Richard Payne, Simon Foster, Alvaro Miyazawa, Cristiano Bertolini & André Didier (2012): *COMPASS Tool Vision for a System of Systems Collaborative Development Environment*. In: *Proceedings of the 7th International Conference on System of System Engineering, IEEE SoSE 2012*, pp. 451–456, doi:10.1109/SYSoSE.2012.6384150.

[8] Joey W. Coleman, Anders Kaels Malmos, Claus Ballegaard Nielsen & Peter Gorm Larsen (2012): *Evolution of the Overture Tool Platform*. In: *Proceedings of the 10th Overture Workshop 2012*, School of Computing Science, Newcastle University.

[9] Controllab products (2013): *http://www.20sim.com/*. 20-Sim official website.

[10] Luís Diogo Couto & Richard Payne (2013): *The COMPASS Proof Obligation Generator: A test case of Overture Extensibility*. In: *Proceedings of the 11th Overture Workshop*.

[11] Luís Diogo Couto & Peter W. V. Tran-Jørgensen (2015): *Extending the Overture code generator towards Isabelle syntax*. In: *13th Overture Workshop*, Oslo, Norway.

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

[13] Eclipse (2015): *Dynamic Languages Toolkit*. Available at `http://eclipse.org/dltk/`.

[14] (2015): *Event-B and the Rodin Platform*. `http://www.event-b.org`.

[15] John Fitzgerald & Peter Gorm Larsen (2009): *Modelling Systems – Practical Tools and Techniques in Software Development*, Second edition. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, doi:10.1017/CBO9780511626975. ISBN 0-521-62348-0.

[16] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat & Marcel Verhoef (2005): *Validated Designs for Object–oriented Systems*. Springer, New York, doi:10.1007/b138800. Available at `http://overturetool.org/publications/books/vdoos/`.

[17] John Fitzgerald, Peter Gorm Larsen & Shin Sahara (2008): *VDMTools: Advances in Support for Formal Modeling in VDM*. *ACM Sigplan Notices* 43(2), pp. 3–11, doi:10.1145/1361213.1361214.

[18] John Fitzgerald, Peter Gorm Larsen & Marcel Verhoef, editors (2014): *Collaborative Design for Embedded Systems – Co-modelling and Co-simulation*. Springer, doi:10.1007/978-3-642-54118-6. Available at `http://link.springer.com/book/10.1007/978-3-642-54118-6`.

[19] Simon Foster, Frank Zeyda & Jim Woodcock (2015): *Isabelle/UTP: A mechanised theory engineering framework*. In: *Unifying Theories of Programming*, Springer, pp. 21–41, doi:10.1007/978-3-319-14806-9_2.

[20] Etienne M. Gagnon & Laurie J. Hendren (1998): *SableCC, an Object-Oriented Compiler Framework*. In: *Proceedings of the Technology of Object-Oriented Languages and Systems*, TOOLS '98, IEEE Computer Society, Washington, DC, USA, pp. 140–154, doi:10.1109/TOOLS.1998.711009.

[21] E. Gamma, R. Helm, R. Johnson & R. Vlissides (1995): *Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series*, Addison-Wesley Publishing Company.

[22] C.A.R Hoare (1978): *Communicating Sequential Processes*. *Communications of the ACM* 21(8), doi:10.1145/359576.359585.

[23] Tony Hoare & He Jifeng (1998): *Unifying Theories of Programming*. Prentice Hall, doi:10.1007/11768173.

[24] Isabelle/Eclipse (2015): *Isabelle/Eclipse*. Available at `http://andriusvelykis.github.io/isabelle-eclipse/`.

[25] Ethan K. Jackson, Dirk Seifert, Markus Dahlweid, Thomas Santen, Nikolaj Bjørner & Wolfram Schulte (2009): *Specifying and Composing Non-functional Requirements in Model-Based Development*. In Alexandre Bergel & Johan Fabry, editors: *Software Composition*, Lecture Notes in Computer Science 5634, Springer Berlin Heidelberg, pp. 72–89, doi:10.1007/978-3-642-02655-3_7.

[26] Cliff B. Jones (1999): *Scientific Decisions which Characterize VDM*. In J.M. Wing, J.C.P. Woodcock & J. Davies, editors: *FM'99 - Formal Methods*, Springer-Verlag, pp. 28–47, doi:10.1007/3-540-48119-2_2. Lecture Notes in Computer Science 1708.

[27] Peter W.V. Jørgensen, Luís D. Couto & Morten Larsen (2014): *A Code Generation Platform for VDM*. In: *The Overture 2014 workshop*.

[28] Leslie Lamport (2002): *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley. Available at `http://research.microsoft.com/users/lamport/tla/book.html`.

[29] P. G. Larsen, B. S. Hansen et al. (1996): *Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language*. International Standard ISO/IEC 13817-1.

[30] Peter Gorm Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl & Marcel Verhoef (2010): *The Overture Initiative – Integrating Tools for VDM*. *SIGSOFT Softw. Eng. Notes* 35(1), pp. 1–6, doi:10.1145/1668862.1668864.

[31] Peter Gorm Larsen, Kenneth Lausdahl & Nick Battle (2010): *Combinatorial Testing for VDM*. In: *Proceedings of the 2010 8th IEEE International Conference on Software Engineering and Formal Methods*, SEFM '10, IEEE Computer Society, Washington, DC, USA, pp. 278–285, doi:10.1109/SEFM.2010.32. ISBN 978-0-7695-4153-2.

[32] Kenneth Lausdahl, Hiroshi Ishikawa & Peter Gorm Larsen (2015): *Interpreting Implicit VDM Specifications using ProB*. In: *Proceedings of the 12th Overture Workshop*, *Technical Report Series* CS-TR-1446, Computing Science, Newcastle University, pp. 1–15. Available at `http://www.cs.ncl.ac.uk/publications/trs/papers/1446.pdf`.

[33] Kenneth Lausdahl, Peter Gorm Larsen & Nick Battle (2011): *A Deterministic Interpreter Simulating A Distributed real time system using VDM*. In Shengchao Qin & Zongyan Qiu, editors: *Proceedings of the 13th international conference on Formal methods and software engineering*, *Lecture Notes in Computer Science* 6991, Springer-Verlag, Berlin, Heidelberg, pp. 179–194, doi:10.1007/978-3-642-24559-6_14. Available at `http://dl.acm.org/citation.cfm?id=2075089.2075107`. ISBN 978-3-642-24558-9.

[34] Michael Leuschel & Michael Butler (2003): *ProB: A model checker for B*. In: *FME 2003: Formal Methods*, Springer, pp. 855–874, doi:10.1007/978-3-540-45236-2_46.

[35] Michael Leuschel & Michael Butler (2005): *Automatic refinement checking for B*. In: *Formal Methods and Software Engineering*, Springer, pp. 345–359, doi:10.1007/11576280_24.

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

[37] Tobias Nipkow, Lawrence C Paulson & Markus Wenzel (2002): *Isabelle/HOL: a proof assistant for higher-order logic*. 2283, Springer Science & Business Media, doi:10.1007/3-540-45949-9.

[38] Terence Parr (2007): *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf.

[39] Lawrence C. Paulson (2010): *Three Years of Experience with Sledgehammer, a Practical Link between Automatic and Interactive Theorem Provers*. In Renate A. Schmidt, Stephan Schulz & Boris Konev, editors: *Proceedings of the 2nd Workshop on Practical Aspects of Automated Reasoning, PAAR-2010, Edinburgh, Scotland, UK, July 14, 2010*, *EPiC Series* 9, pp. 1–10.

[40] Jan Peleska, Elena Vorobev & Florian Lapschies (2011): *Automated Test Case Generation with SMT-Solving and Abstract Interpretation*. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann & Rajeev Joshi, editors: *Nasa Formal Methods, Third International Symposium, NFM 2011*, NASA, Springer LNCS 6617, Pasadena, CA, USA, pp. 298–312, doi:10.1007/978-3-642-20398-5_22.

[41] (2015): *The TLA Toolbox*. `http://research.microsoft.com/en-us/um/people/lamport/tla/toolbox.html`.

[42] Marcel Verhoef (2009): *Modeling and Validating Distributed Embedded Real-Time Control Systems*. Ph.D. thesis, Radboud University Nijmegen.

[43] Marcel Verhoef, Peter Gorm Larsen & Jozef Hooman (2006): *Modeling and Validating Distributed Embedded Real-Time Systems with VDM++*. In Jayadev Misra, Tobias Nipkow & Emil Sekerinski, editors: *FM 2006: Formal Methods*, Lecture Notes in Computer Science 4085, Springer-Verlag, pp. 147–162, doi:10.1007/11813040_11.

[44] J. Woodcock, A. Cavalcanti, J. Fitzgerald, P. Larsen, A. Miyazawa & S. Perry (2012): *Features of CML: a Formal Modelling Language for Systems of Systems*. In: *Proceedings of the 7th International Conference on System of System Engineering*, IEEE, doi:10.1109/SYSoSE.2012.6384144.

[45] Xtext (2015): *Xtext*. Available at `https://eclipse.org/Xtext/`.