

Sex up Overture

Tiago M. L. Alves
tiagomlalves@gmail.com

November 13, 2006

Abstract

In in this informal paper I intend to lay down some personal thoughts about how to make Overture more appealing as a software engineering tool. For this, I will not only focus on how Overture can take advantage over the existent technology but also how Overture can be used as tool integrated with existent technology. I will describe some concerns about the Overture past, used technology, technology integration such as code generation, reverse engineering, re-engineering and validation. Finally I will describe how Overture can take the most out of Eclipse.

1 Introduction

The Overture project is a initiative to provide open-source formal methods tools. However, this initiative does not intend to start a new formalism and set of tools from scratch. Behind it there is a 26 years-old legacy: the Vienna Development Method (VDM), which was supported by tools such as SpecBox and VDMTools.

However, besides the 20 year-old VDM legacy, the Overture also carries the cross of stopped being supported. The question that remains is that, with such a past what is the future of Overture?

When something fails, and we have opportunity, we should look behind, learn from the errors that were made and try again. Develop support for formal methods is a long journey, but as the Japanese saying goes, the longest journey starts with a single step. And this first step was took with the decision of going to open-source with all the advantages of that.

What about the rest of the journey? This paper routes about a possible path for that journey avoiding those errors. My focus will be about technology, since is my belief that VDM did never properly integrated with the existent technology.

In Section 2.1 I describe some concerns about the Overture tool past and some issues that can delay the software evolution. I also describe what is missing about the open-source philosophy.

In Section 2.2 I describe the importance of the choice of the technology, by analyzing the language recognizer. In particular I express some concerns about the use of *lex & yacc*.

In Section 2.3 I describe the importance of supporting target languages through code generation, reverse engineering, reengineering and validation. To aid the reader with some concepts code examples about both code generation and reverse engineering are provided.

Finally, in Section 2.4 I describe some technologies that would allow provide good user experience make the most out of Eclipse.

2 The Overture journey towards technology

The Overture project is fresh air to formal methods community in providing an open-source initiative aiming the development of formal methods tools. This means several things. To the formal method community it means new research possibilities. To the open-source world this means that anyone should be free to use and to contribute to the tool and those contributes can raise new interesting challenges that were never considered before.

2.1 Enable fast evolution

A new software product is interesting if it presents new ideas or concepts if it is something that people believe that can make their life easier. However even if that product gains lot of initial interest, if it does not evolve the interest will decline. Even if the initial great ideas are still there no one will care.

In order to overcome this, it is necessary to enable fast evolution. However, evolution tends to be very difficult if the project is tied up to its past, more specifically to VDMTools by requiring backwards compatibility.

To have freedom to make changes, Overture must be completely independent of any VDMTools effort. It should allow syntax or semantic changes, for instance. If the Overture project must provide compatibility with the VDM++ supported by VDMTools this will make language evolution very hard allowing only extensions that do no harm VDM++. Every time a new extension is implemented this will provoke a series of language patches which will inevitable raise both the language and the tool complexity. This complexity will create additional entropy to anyone willing to extend the tool since they will be required to solve several problems that do not belong to their particular domain.

Another concern about evolution is how the evolution itself is supported. It was decided that Overture should go for open-source, however the release of software under open-source license is not sufficient. It is necessary to follow open-source development methodologies by making all changes immediately available. A good starting point is the *sourceforge* project which provides an extensive list of tools to aid programmer such as support for versioning (via CVS and Subversion), releases control, documentation, mailing lists, and issues tracking.

Committing all changes to *sourceforge* provides a good way to keep track of changes, but also serves as a good teaching example. It is very important to do this to show that the project is not dead and there are people contributing to it. Independent of correcting bugs or just play around trying some ideas, this should be made inside *sourceforge* since it is possible to create branches in the repository such that this does not affect the main development.

The reason for having everything in a public repository, is this allow to share the maximum of knowledge and is more

To enable fast evolution it is necessary to cut with the roots and look forward and from the Overture past we should only take the lessons. Moreover, going to open-source should not only be reflected by releasing software under an open-source license but also develop under open-source spirit.

2.2 Choice of technology

Technology in form of implementation language, libraries and tools, pays an important role both through development and maintenance of any software product. Thus it is very important to choose the adequate technology that fits the project purpose.

In commercial projects, the choice of the technology is made according to

costs, maturity, support, etc. Normally there is a strong demand for proven technology to achieve “robustness” while cut-edge technology are normally avoided.

On the other hand, research projects such as Overture should not be so strict about technology, they should be free to use more cut-edge technology whenever it provides additional value. The choice of a particular technology should be based on reasons such as added value, effort that saves, etc.

Take as example the implementation of the front-end of the tool (also known as the language recognizer). For a commercial project, if we had to choose the most proven technology, the election would be the twenty years old couple *lex & yacc*. Even known that using such technology will require to implement a lexer, a parser, an AST and maintain code to create AST, *lex & yacc* is always a safe choice since their algorithms and performance are well known.

However in twenty years many alternatives arose, such as *ANTLR* and *SDF*, to name a few. Making a general comparison between *ANTLR* and the “safe choice”, from a single *ANTLR* grammar it is possible to automatically generate the lexer, parser and tree building. Thus the effort of maintaining code would be fairly small compared to the *lex & yacc* approach. Nevertheless, one should pay attention that *ANTLR* uses a LL(k) algorithm that is less powerful than LALR (algorithm used by *yacc*) meaning that the implementation of the parser will be more complicated, and grammar changes difficult. On the other hand, *SDF* not only supports automatic generation of the lexer, scanner and AST but also it is possible to generate a pretty printer and serialization support¹ for both Java and Haskell programming languages. Additionally, *SDF* uses a *GLR* algorithm which is much more powerful than *LALR* meaning that grammar changes cause only minor impact.

In sum, although technology can provide benefits, the technology choice should be made in a carefully way in order to minimize the impact that it can have in a near future. In the particular case of the language recognizer although the technology provided very fast development initially, grammar evolution will be very hard to maintain.

¹Other components, such as VDM types, can also be easily generated.

```

class Test
  types
    Status = <Running> | <Stopped> | <Aborted>;
end Test

public enum Status {
  Running,
  Stopped,
  Aborted
}

```

Figure 1: Code generation example using enumerated types.

2.3 Support target languages

Although the formal specification is an important artifact for software development it is very difficult to convince users to adopt formal methods if there is not a good support for target languages through code generation, reverse engineering and reengineering, and validation.

2.3.1 Code generation support

Code generation is a need that was recognized in VDMTools, and such both Java and C++ code generation was supported. However, code generation as feature is not enough, good code generation is needed!

Lets consider the VDM++ specification presented on the left hand side of Figure 2.3.1, in which it is specified a quoted type having three different values: *< Running >*, *< Stopped >* and *< Aborted >*.

Using the VDMTools to generate Java from this simple specification three classes are outputed, namely: Running, Stopped and Aborted. Taking as example the *Aborted* java class definition, the file is 66 lines long from which 25 lines are comments, 23 empty lines and 18 lines of code (in which in the last line there is a “;” that is useless). The generated class do not reference in any way the Status data type which was defined in the formal specification meaning that any function that have this datatype as input or output will be redefined to Object. Moreover because I didn’t specified the Java package the tool decided to generate it own package name called “quotes”!

If I would implement this specification I would write the Java code presented on the right hand side of Figure 2.3.1 which defines a single enumeration type. This implementation not maintains the meaning of the specification but also is type safe (meaning that it can also be checked by the Java compiler). On the other hand, if in VDM++ I specify any function using this type, no reference for the Status datatype will be made, only to the generic Object type.

This is a simple example of how the Java code generation can be improved. Although to have a Java code generator is very important, since it is one of the most popular programming languages, other target languages should be considered. Examples are SQL, as proven to be possible in the VooDooM prototype, but also CORBA IDL, C, Haskell, etc.

In sum, Overture should pay particular attention to support code generation such that it should be possible to extend the tool with code generations that fit anyone's purpose.

2.3.2 Reverse engineering support

Software reverse engineering is an important research challenge and it has been explored using many different approaches such as graphs, UML, and formal specifications.

VDMTools supported software reverse engineering from Java code in two modes: “stubs only” and “code transformation”. In the first mode, only method signatures are analyzed while in the second mode source code is completely analyzed and translated to VDM++.

However, the “stubs only” provides too much abstraction since it ignores the Java methods body, and the “code transformation” provides too much details by making a direct translation from Java to VDM++. A possible alternative to this is to use the VDM++ primitive types as another way to achieve abstraction (since this is one of the key factors of the VDM++ expressiveness). Therefore a possible transformation could be to detect Java libraries such as Set, Map and List and use the VDM++ equivalents.

As an example consider the bag data structure Java implementation presented in the left side of Figure 2.

The *elements* variable is of the interface type *Map* and uses a particular implementation: *HashMap*. If VDMTools was used to reverse engineer this piece of code, it will interpret Map and HashMap as objects while it could recognize them and abstract them to the primitive VDM++ datatype *map* as shown in the right side of Figure 2.

Of course the specification presented in the right part of the figure is for illustrative purposes only, it is not based on any previous study, and intends to show another possible path for reverse engineering under the Overture umbrella.

```

import java.util.HashMap;
import java.util.Map;

public class Bag {
    private Map elements;

    public Bag() {
        elements = new HashMap();
    }

    public Object add( Object element) {
        Integer counter = 0;

        if( elements.containsKey( element)) {
            counter = (Integer) elements.get( element);
        }
        counter++;

        return elements.put( element, counter);
    }
}

class BagSpec

instance variables

private elements: map Object to Object;

operations

public add: Object ==> ()
add( element) ==
    ( dcl counter : int := 0;
      if element in set dom elements
        then
          ( counter := elements(element)
            );
          counter := counter + 1;
          elements := elements ++ { element |-> counter }
        );
end BagSpec

```

Figure 2: Example of a Bag implementation and a possible reverse engineered specification in VDM++.

2.3.3 Reengineering support

Reengineering support in the Overture tool could be a great weapon to promote formal methods.

The idea is that formal specification and implementation should co-exist and be always be updated. To achieve this, the Overture tool must support automatic change detection and synchronization, otherwise if this work is left to the user sooner or later specification and implementation will diverge.

This implies a closer integration with a target technology. Using as target language the Java language an ideal scenario would be a situation that an addition in the Java implementation (method or instance variable) should warn the user that the formal specification misses changes done in the implementation and vice-versa, changes in the formal specification could be generated to the implementation.

There is some specific concerns to be taken into account when changes in the formal specification occur such as generation should be done in an incremental manner due parts of implementation can be done by the user.

Supporting reengineering can bring several benefits such as the introduction of formal methods as a complement to the traditional existent technologies and methodologies and doing this in a non-intrusive way.

2.3.4 Validation support

One of the purposes of the formal specification is to provide means of validation. However, in the actual implementation of the VDMTools, validation is only done in the specification and no validation is possible against the actual implementation.

Validation should start by checking if the implementation respects the specification in terms of variables and API (functions and methods) by checking if all the methods are specified or if they respect the same signature as in specification. Invariants, pre and post conditions should be generated and executed by the implementation during run-time through code instrumentation or other which would allow to give more confidence to the programmer when implementing the functionality by hand. Automatic tests (in form of JUnit for the Java programming language) should be generated which would allow test both current implementation and regression tests.

Furthermore other types of validation should be possible depending of the application domain. Imagine that could be possible to use Overture for formal specifying databases which has been prove to be possible with the VooDooM prototype. From the formal specification SQL schema could be generated and from the datatypes invariants specification code could be generated to check for database inconsistencies.

Applications like this could lead to the development of embeddable formal methods in which a formal specification can be used not only for as proof, but also for generation and validation making Overture a very powerful software engineering tool.

2.4 Provide good user experience

A known truth in software engineering is that a product can be great in terms of architecture, features, technology but if it does not offer a good user experience the product is doomed.

One possible way to overcome this is to offer a good integrated development environment (IDE) with a clean graphical user interface (GUI). Fortunately, today it is not very hard to create good GUIs for programming languages since there are several IDEs that can be extended with other languages and functionalities. On the top of the list is the Eclipse platform, which not only allows extension mechanism to support new languages or functionalities, but also allows the interaction with already existent features

such as UML editors or versioning systems (such as CVS and Subversion).

Integration with Eclipse it is not straight forward, but in longer term it compensates. From the technology point of view, Eclipse allows good modularization using the concept of “features” which allow the development of separated modules for sets of functionalities. Additionally since both Java Development Tools and C/C++ Development tools are based on Eclipse, the development of the Overture has the benefit that it could use the already existent functionality to provide formal methods support for those languages.

Additionally, JUnit testing and coverage are already built-in (directly or via external modules, respectively) in Eclipse which means that one could use Overture to generate those components and run them from Eclipse (without being intrusive).

To finalize the list to provide good user experience, two more technologies could be supported ANT for building and running and UML through one of the several available UML editors for Eclipse.

3 Conclusion

In this paper, I describe some technical issues in the current implementation of the VDMTools and I provide some ideas how to overcome these problems in Overture.

I start with some considerations about the Overture past, and defend that Overture should be free for previous VDMTools efforts and choose its own path in an independent way. Choosing open-source community is a very good first step, but it is also necessary to have an open-source development way of thinking and develop according open-source rules.

Some considerations about technology are also done and in particular about the language recognizer which, in my belief it was not a proper choice according to future requisites of the Overture tool, such as supporting language extensions.

Continuing about technology, I describe the importance of the target-languages and ways to support code generation, reverse engineering, re-engineering and validation.

Finally, I focus in how to provide good user experience describing the importance of the Eclipse platform to deploy a tool such as Overture.

The considerations provided in this paper are based in personal opinion formed during the contact I had with formal methods during my previous

studies and about my short industry experience.