VDM vs. Programming Language Extensions or their Integration

Alexander A. Koptelov and Alexander K. Petrenko

Institute for System Programming of Russian Academy of Sciences (ISPRAS),
B. Communisticheskaya, 25, Moscow, Russia
{steve,petrenko}@ispras.ru
http://www.ispras.ru/groups/rv/rv.html

Abstract. The paper considers problems of joint usage of different modeling languages and tools supported these languages. Basically we focus on the technical policy for such integration.
Keywords: VDM, JML, Spec#, UniTesK, modeling methods, formal method integration.

1 Introduction

Nowadays special attention is paid to modeling methods and tools. At the same time there is no common conception of what these tools should look like, how they should be integrated with other CASE tools, and how their usage should fit into real-life software development processes. First we are not intending to propose a general and universal solution. Instead, we will compare different approaches and describe their strengths and weaknesses. We are bringing up a question of integration of these tools into software development processes — a question "for what?" — to be able to bring up a question of integration techniques and answer the second question "how?".

The idea of software development and analysis tools integration is close to many researchers and practical developers. Along with the advent of new technologies and languages, there always remains certainty that there is no, and will never be proposed, common solution for real-life large-scale projects that would satisfy needs and limitations of both problem area and people/organizations involved into design, development, support and usage of the system under implementation. If this is the case, the problem of what components will constitute an engineering environment and how they will be integrated should be solved. Each time thinking of such composition we are guided by two mutually contradictory criteria. On the one hand, we want each activity to be supported by the most adequate tool. On the other hand, we need to take care about minimization of the system's complexity, i.e. to reduce the number of tools, links between them and required skill level of personnel, since we cannot expect that an unlimited number of experts that easily operate with any possible tool can be involved into the project. We consider that technical problem of integration is secondary with respect to definition of a technical policy that determine goals of

this integration. In its turn, a well-founded technical policy should be based on understanding what activities constitute a real-life process and what tools and techniques are most suitable for such activities. That is exactly what we want to achive. The problem of integration is considered in a context of choosing the most suitable tool (language) for modeling. Then, when each language has its own place, we can proceed with question of how to integrate tools supporting each language into a single technological chain.

So, to design interfaces of a tool we need its use cases. To define possible use cases we should determine position the tool or underlying method in software development practice. Education, research projects, and industrial applications are kinds of such practice. We pay the most attention to industrial applications, but other cases also should be concerned. The second aspect of tool's usage in practice is an activity performed — modeling, static analysis, model checking, formal verification, and testing. All of these activities are important for industrial applications. The third aspect is the set of tools that will be interact through the interface to be defined. These tools can support different notations (we consider an example below) or a single notation (like JML, which is supported by many different tools [1]).

2 VDM vs. Eiffel, Spec# and UniTesK

VDM-SL and VDM++ are widely known specification and modeling languages. VDM is a language with formally specified semantics that allows investigating a model's behaviour in a formal way, in particular, to perform verification. VDM and other languages developed for formal methods (Z, B, RSL etc.) will be called "specification" in this article. Recently, besides specification languages both programming languages (Eiffel [2]) and languages that are extensions of commonly used programming languages were proposed for modeling, requirements and behavior description, e.g.:

JML [3] – Java language extension
Spec# [4] – C# language extension
UniTesK family [5] – extensions of C, C# and Java

The languages mentioned above are allied to each other. At the same time, each of them is special in some aspects. These specialities are mostly caused by a model of language usage in development practice (e.g. modeling, software contract definition, testing). In its turn the model of usage causes tendencies in the language evolution and in tool suite surrounding the language.

The model of VDM usage is as follows.

- VDM is used at early stages of design. Even at the draft stage it allows to strictly declare interfaces – data types, signatures of operations, constraints for data (invariants) and operations (pre- and postconditions). Rigour of these declarations allows detecting ambiguities and gaps in target setting that is one of main advantages of the VDM method.

- When the model gets more detailed and model implementations are created, ability of algorithms verification appears, so it possible to prove that the algorithms comply with the requirement specification. The refinement of a model can be done incrementally. Proofs can be done separately for each step of refinement. In practice, on each step of refinement inconsistencies can be found in both specification and task definition. It is also valuable advantage of the method that it allows early detection of errors.
- When the model is detailed enough and complete (i.e. all algorithms are described), it is translated (automatically or by hand) to a programming language making a prototype system as a result.
- Then the implementation can be developed. As far as its interfaces remain unchanged (in both structural and functional matters), it is possible to perform a simple comparison of model and implementation behavior. In particular, it is possible to use common test suites (paying attention to notations they are written in). If interfaces diverge, such comparison gets harder.

The process of Eiffel usage is so-called Design-by-Contract that is a parallel development of specification and implementation (specification should be developed slightly earlier). In the Design-by-Contract, less attention is paid to sequential refinement. It is supposed that the architecture of the system appears at once, what is typical to many practical cases. Thus, the problem of proving the correctness of transition between less and more detailed models disappears. It should be noted that specifications and implementations are always synchronized (at least as it is stated in the Design-by-Contract ideology). Also, since the Eiffel language doesn't confine programmer very much, it is easy to write a program conformed to specification's constraints where would not be easy to detect inconsistencies between algorithms and specifications. The Eiffel language provides all facilities needed, so stage of translation to a programming language is not needed here. Conceptually it is an advantage, but in a real-life project it is often needed to use external components like databases, libraries etc., so it is not quite true to consider that a Eiffel programmer is working in a single-language environment.

These programming languages extensions are aimed at avoiding two-lingual programming environments. A two-lingual programming environment has two serious disadvantages from a practical point of view. The first is a technical one: additional means are required to link components developed in different languages. The second one is rather regarding to organizational and economical issues. The skills in two languages and tools of their integration are required to work in two-lingual environment. This point becomes one of the first barriers on the way of using specification languages in industrial applications.

The JML and Spec# came along with the Eiffel. The only difference is that Java and C# were taken as base languages, so authors did not propose any new notation where basic language features can be used. JML, Spec# and Eiffel are close to each other, not only by notation. They follow a common ideology and support the same Design-by-Contract development processes. The main point of this process is development of implementation (exactly implementation, not

model) and constraint specifications in parallel. This process is congenial to the Test Driven Development scheme that has its apologists in XP in particular. All these three languages are aimed at software quality improvement. This is supposed to be reached by means of both specifications development and systematic and automatic checking of specifications and implementation correspondence. JML and Spec# stake on analytic and model checking. To allow this, specification extensions contain corresponding limitations as compared to their basic languages.

These three languages are tools for software developers. Architecture of models and specifications is identical to architecture of system under implementation on corresponding language platform. There's no special possibility that simplify parallel development of specifications and implementation, since they both are in one person's hands — in the developer's hands.

UniTesK is a family of testing tools that supports specification extensions of various programming languages. At the present time such support for C, C# and Java languages has been implemented. These extensions are somewhat similar to JML and Spec# (though Spec# was developed essentially later than UniTesK for C#). The main difference is that UniTesK extensions are developed as real languages instead of the language with pseudo-comments. Moreover, UniTesK provides ability to separate specification classes and implementation classes (so, specification and implementation developers can work relatively independently). In addition, specifications support multilevel models, where different levels correspond to different levels of abstraction. Consequently, more abstract models are preserved as more detailed ones are developed. They are maintained in an actual state, what allows analyzing interfaces and functionality at both the current and earlier stages of development. It simplifies requirements traceability from informal requirements untill implementation. One more essential difference is the presence of language constructs for test design. Moreover, the fact that UniTesK is aimed at testing, determines both language itself and set of tools that supports it. Aiming at testing considers specific processes where UniTesK demonstrates its best characteristics. Under the assumption that new software is developed (on the contrary to updating some legacy software), UniTesK assumes concurrent development of specification and implementation. Thus, such process of UniTesK or other language usage can be called "co-verification". Because specification classes of UniTesK in both C# and Java specification extensions can be separated from each other there is a real possibility to run development and specification and design in a concurrent fashion. Notice, both JML and Spec# differs from UniTesK. JML and Spec# focus on specification of separate modules and basically ignoring problems of specifying a (sub-)system as a whole. Therefore they basically describe functional requirements in implementation terms. Such an approach makes JML and Spec# tools closer to debuggers that operate with code but not with specification design.

Turning back to UniTesK, it should be noticed that UniTesK can be used not only in co-verification process. It demonstrate the same power in continuing development and maintenance phase. During such a phase regression testing

becomes an important problem. Well organized regression testing supposes relatively low level of effort related to test suites maintenance in case of incremental, gradual evolution of implementation. In contrast to JML and Spec#, UniTesK meets requirements of regression testing. For this purpose UniTesK provides mediator (adapter) classes describing mapping of specification interfaces into implementation interfaces. The presence of this intermediate layer and splitting specification and implementation classes drastically facilitate regression testing.

Let's compare modeling languages facilities and evaluate suitability of each of the languages in different processes. Some short comparison is presented in the table below.

		VDM	Eiffel	JML	Spec#	UniTesK
Subprocesses	Requirements definition, prototyping	++				-/+
	Architecture design	++	-/+	-/+	-/+	+/-
	Implementation		+	++	++	+
	Testing					++
	Maintenance and regression testing					++
Users	Researchers, teachers, students	++	+	+	+	
	Architects	-/+				-/+
	Developers (and verifiers)		+	+	+	+
	Test designers					+
Features	Modeling, verification of models	+	-/+	+	+	-/+
	Generation of prototypes from models	+				
	Model Based testing of implementation					++

"++" means "top of suitability"
"+" means "quite suitable"
"-/+" and "+/-" means "almost suitable"

Table 1. Comparison of modeling approaches based on different languages.

This table suggests what kind of composition of languages, techniques, and tools we need. Notice, we need "seamless" integration that solves any kind of problems: technical, human, organizational, etc. In the set of languages under consideration VDM is the best language (and tool) for sketch design because of its expressiveness, closeness to mathematics, and means for abstraction. In addition VDM is the best in learning processes because of the same reason. The same characteristics are important for experimenting with prototypes and architectural design. VDM is quite acceptable for all project phases until model becomes too detailed. When a project enters the implementation phase and development focuses on module implementation, JML and Spec# are better than VDM (it is only our vision), because of their closeness to implementation language platforms. UniTesK can be involved in a real project if the customer or developer plans long development and regression testing. In this case separation

of specifications from implementation and specific support of test generation will be very important.

3 Multilanguage processes and tools integration

So, on the one hand we try to find the best tool for each of the works. On the other hand we advice to avoid any multilingual solution. It seems the only compromise that exists. We have to build a composition of languages but we should minimize intersections of bi-lingual (or multilingual) periods. A "Good enough" solution is to use VDM during prototyping phase, translate VDM models into JML/Spec# /UniTesK (target language depends on developer's preferences) and then do not support relations/traceability between VDM models and specifications in implementation platform. Application of tools developed for JML (for example model checkers) or for UniTesK (regression test suites generation) to VDM models becomes superfluous.

We have suggested technical policy: "Minimize period of multilingual design but support smooth integration of languages through boundaries of life cycle phases ". Let's come back to the technical question: "How to integrate?". Each tool of UniTesK family can be considered as a toolkit - it actually consists of extended language checker, translator, interactive templates for test components construction, test engine that generates test sequences on-the-fly, trace collector, and tools operating with test traces (analyzers, navigators, animators, report generators, etc.). These tools use two kinds of interfaces — specification language model and test execution trace. Specification language model is a kind of abstract syntax tree represented in TreeDL framework [6]. Trace is recorded in XML. These two interfaces are sufficient. To improve the performance both in terms of processing time and memory spend we maybe will use other trace format than XML. So, we can conclude that JML and UniTesK from on side and VDM++ with Overture project [7] on the other side use different means to provide interfaces between tools. In relation with that, we should once more consider possible usage of tool integration to provide suitable interface for it in the Overture project.

References

1. http://www.gemplus.com/smart/rd/publications/pdf/BCC_03jm.pdf
2. http://www.eiffel.com/
3. http://www.cs.iastate.edu/~leavens/JML/index.shtml
4. http://research.microsoft.com/SpecSharp/
5. http://www.unitesk.com/
6. http://treedl.sourceforge.net/
7. http://www.overturetool.org/