RAISE in Perspective

Klaus Havelund

NASA's Jet Propulsion Laboratory, Pasadena, USA
Klaus.Havelund@jpl.nasa.gov

1 The Contribution of RAISE

The RAISE [6] Specification Language, RSL, originated as a development from VDM [9] during a five year effort involving several researchers. The purpose was to improve VDM by augmenting it with a module system, a process description language, a formal semantics standard, and tool support. A goal was to keep the language a wide spectrum language including high level specification constructs as well as low level programming constructs, allowing specification and program fragments to be mixed arbitrarily with each other without imposing a linguistically layered language. The effort resulted in a language inspired by, but in many ways different from VDM. A main deviation from VDM is the emphasis on an algebraic specification-style logic where a module consists of a signature and a set of axioms over the names introduced in the signature. Derived forms exist which reflect the classical VDM definitional style. Correspondingly, types can be abstract sorts as in algebraic specifications or they can be defined through type definitions as in VDM using what is normally referred to as a model oriented style. The pure model oriented style usually uses such explicit type definitions and a definitional style for functions. The pure algebraic style uses sorts and more liberal equations with arbitrary terms on the left hand side as well as on the right hand side. In the axiomatic style emphasis is on operations and how they relate to each other. In the model oriented style there is a specific emphasis on type equations. When modeling a problem in VDM, the problem is usually first approached by writing down a set of type definitions using set, list and map type constructors. Operations are subsequently defined referring to the operators allowed on values of these types. The same style is possible in RSL, although the added module system suggests the association

of operations with types in a compartmentalized manner.

Although the language is completely uniform, RSL is often for pedagogical purposes presented as supporting three paradigms: functional, procedural with side effects on a state, and process algebraic. In addition each paradigm can be presented in an axiomatic style or in a model oriented definitional style, conceptually forming a 3 by 2 matrix. However, there is only one notion of function, which potentially can have side effects or communicate on channels, and which can be defined axiomatically, with a special case being a model oriented definitional style. Considering this 3 by 2 matrix, the theoretical contribution of RSL was the algebraic specification of functions with side effects on state variables and channels, and the linguistic and semantic unification of all these concepts.

2 Relationship to Programming Languages

However, in spite of the axiomatic capabilities of RSL, the model oriented definitional style seems more often used. For example the case study in the RAISE chapter is mostly written in such a style (first functional and then procedural, but both model oriented). The model oriented approach tends to convey information succinctly. For example, by stating that a document is a list of lines, each of which is a list of characters, one has said a great deal before even introducing any functions on documents. In essence, many specifications have the flavor of high level programs. This observation might lead to question what the relationship between specification languages and programming languages should be. Traditionally many specification languages are seen as existing separately from programming languages, only connected with a translator from the specification language to the programming language in case the specification language contains an executable subset. This separation has some advantages and some disadvantages. A main advantage is clearly that the specification language can be used to describe systems independently of the final choice of programming language. The implementation can even be programmed in different programming languages, which is in fact typically the case. Another advantage is that the specification language is liberated from issues of executability. Specifications can be as abstract as required.

Amongst the disadvantages is the fact that a software project has to administer artifacts written in a specification language as well as in a programming

language. One can imagine that some parts of the system have been implemented already in a programming language, while other parts have been captured in a specification language, resulting in a multi language situation. This problem seems even less necessary when considering that specification language and programming language often have many constructs in common. It is not always possible to rely on a translator from the specification language to the programming language. Typically such a translator will not yield code that is efficient enough. The programmer will not trust the complicated translation process and would be more comfortable with a real one-step compiler. Finally, the link between specification and program cannot in practice be formal. In theory it can be formalized, but it would require a formal semantics of the programming language and a proof that every specification is translated to a semantically equivalent program.

Whether one will argue for or against a separation between specification language and programming language, it is clear that there are advantages of combining specification and programming into one language. An interesting language in this context is the widely used scripting and programming language Python [5]. Python has built-in succinct notation for sets, lists and maps, and iterators over these, exactly the core data types of VDM and RSL. These concepts also exist in Java [3], although as libraries. These are examples of using high level constructs for programming. Some programming language extensions incorporate specifications in a layered manner, where specifications are separated from the actual code, as axioms [2] or as pre/post conditions [1, 4, 7].

In general, several concepts have shown to be useful in specification as well as programming, and hence could be considered candidates for integration into a single programming language. These include object orientation, functional programming, algebraic data types generated with constructors and pattern matching over these, as well as succinct notation for operating sets, lists and maps, as well as logic inspired constructs such as pre/post conditions and existential and universal quantification over finite sets. It is even conceivable that equational rewriting rules could be merged with traditional programming constructs in a programming language. Notation-wise this is allowed in RSL, however, it is currently not supported by a computational model.

If object oriented-ness means that objects are first class values, then RSL is not object oriented. The integration of object orientation into RSL was at the time regarded as a theoretical complication. Furthermore, object

orientation was not yet common practice when RSL was designed. Object orientation has, however, shown to be a useful way of encapsulating state. As an example, an object oriented presentation of the case study without explicitly mentioning the state variable might be more succinct than the functional style where the state is passed as argument to all functions.

The main point of the above discussion has been to emphasize that specification languages and programming languages conceptually overlap and that the gap between the two universes is not as big as one could believe. It is desirable that more ideas from formal specification languages transfer into programming languages and that there is a more elaborate exchange of ideas between the two communities. The formal methods community has much to offer the programming language community, and vice versa.

Verification and Testing

As a final point, it is worth mentioning the use of formal methods for testing. RAISE stands for "Rigorous Approach to Industrial Software Engineering". By "Rigorous" is meant that the correctness of a software artifact developed using the RAISE technology can be justified by a proof, relating formal artifacts. Rigor is an important and essential element of a formal method like RAISE. However, rigor comes with a price: generating proofs is hard. Testing still seems to be the most practical approach for large specifications. The testing method referred to in the paper resembles various unit testing methods found in programming. The user writes a set of tests, each of which performs a sequence of function calls on specific data, and then observes the result. A useful augmentation of this approach would consist of prefixing these tests with universal quantifications over data referred to in the tests, and then use automated selection of data from the types quantified over for automated testing. Selection of data from the types could furthermore be guided by user-provided strategies. A more uniform view would be to regard some equational axioms as test cases, hence avoiding introducing new concepts into the language. A specification should be directly usable for generating test cases. A special view on testing is runtime verification [10], where a specification is used to monitor the execution of the final program. If specifications become part of the programmer's test arsenal, there is a bigger chance that specification technology will be adopted by practitioners.

References

[1] Eiffel. http://www.eiffel.com.

[2] Extended ML. http://homepages.inf.ed.ac.uk/dts/eml.

[3] Java. http://java.sun.com.

[4] JML. http://www.cs.iastate.edu/~leavens/JML.

[5] Python. http://www.python.org.

[6] RAISE. http://www2.imm.dtu.dk/~db/raise.

[7] Spec#. http://research.microsoft.com/specsharp.

[8] Standard ML. http://en.wikipedia.org/wiki/Standard_ML.

[9] VDM. http://www.vdmportal.org.

[10] Runtime Verification Workshops. http://www.runtime-verification.org.