Combining Modeling and Programming – Towards Advanced Languages for Software Development

Manfred Broy, *Technical University of Munich, Munich, Germany*
Klaus Havelund, *NASA Jet Propulsion Laboratory, Pasadena, USA*

1 Introduction

Over the last several decades we have observed the development of a large collection of specification and modeling languages and associated method methodologies, and tools. Their purpose is to support modeling of requirements and high-level designs before programming is initiated. Agile approaches advocate to avoid explicit modeling entirely and suggest to go directly to coding. Other approaches advocate avoiding manual "coding" in a programming language entirely and suggest instead the generation of code directly from the models. This way modeling languages replace programming languages. We can divide these modeling languages into formal specification languages (formal methods), usually focusing on textual languages based on mathematical logic and set theory, and associated proof tools (theorem provers, model checkers, etc.), and on the other hand model-based engineering languages (UML, SysML, Modelica, Mathematica, …), focusing more on design, code generation and simulation. Many of these modeling languages have similarities with programming languages.

In parallel, and frankly seemingly independent, we have seen the development of numerous new programming languages. Few languages have had the success of C, which still today is the main programming language for embedded systems. The success is so outstanding that nearly no progress wrt. praxis has been made in this domain (embedded programming) since the 1970ties. In application programming a collection of new languages came ago such as Ada, C++, Eiffel etc. At the same time we have seen several high-level languages appear targeting the softer side of software engineering (such as web-programming, user interfaces, scripting), including languages such as Java, JavaScript, Ruby, Python and Scala. More academic languages include Haskell and the ML family, including OCaml.

In this white paper we will give a brief overview of some of the, in our view, important developments in modeling and programming languages. Subsequently we will propose en effort to develop a specification, design and implementation language that combines modeling and programming.

2 Observed concepts in modeling

2.1 Formal methods

Early work on formal methods include the work of John Mclarthy, Robert Floyd (Assigning Meanings to Programs), Edsger Dijkstra (A Discipline of Programming), Tony Hoare (An Axiomatic Basis for Computer Programming), and Dana Scott and Christopher Strachey (Towards a Mathematical Semantics for Computer Languages), to mention a few. These ideas were theoretic in nature and deeply influential. They brought us the ideas of annotating programs with assertions, such as pre- and post-conditions, and invariants, correct by construction development (refinement), and giving semantics to programming languages (denotational semantics).

These ideas were subsequently the basis for several, what we could call second generation, formal specification languages such as VDM, RAISE, Z, TLA, and CIP. Each of these languages were full specification languages, with rich type systems and detailed rules (grammars) for what constituted a valid specification. These languages were ahead of their time wrt. language constructs in the sense that many of the language features found in these languages slowly are finding their way into modern programming languages of today.

The VDM language for example was a wide-spectrum specification language offering a combination of high-level specification constructs and low level programming constructs. The methodology consists in part, as in CIP, of refining a high-level specification into a program in a stepwise manner. The language offered concepts such as the combination of imperative (procedural and later object oriented in VDM++) and functional programming; exceptions; algebraic data types and pattern matching; functions as values and lambda abstractions; built-in collection types such as sets, lists and maps, with mathematical notation for creating values of these types, such as for example set comprehension; design-by-contract through pre- and post conditions and invariants; predicate subtypes (so one for example can define natural numbers as a subset of the integers); and predicate logic including universal and existential quantification over any type as Boolean expressions. VDM and Z are so-called model-oriented specification languages, meaning that a specification is an example model of the desired system. This means that such specifications are somewhat close to high-level programs. This is in contrast to so-called property oriented specification languages, such as OBJ.

A different branch of formal methods include theorem proving and model checking. In theorem proving we have seen specification languages, which resemble

functional programming languages, including for example ACL, PVS and Coq. In model checking early work focused on modeling notations somewhat removed from programming languages. However, recent research has focused on software model checking, where the target of model checking is code, as for example seen in the Java PathFinder model checker, JPF. JPF was created due to the observation that a powerful programming language might be a better modeling language than the, at the time existing, model checker input languages. Today's efforts in model checking furthermore include numerous efforts in model checking of C programs.

As can be seen from the above discussion, formal specification languages have for a long time been flirting with programming language like notations, and vice versa. However, the two classes of languages have by tradition been considered as belonging to strictly separate categories. VDM for example was always, and still is, considered a specification language, albeit with code generation capabilities. It has never, in spite of the possibility, been considered (named) a programming language, which one may consider being as one of the reasons it is not more wide spread. Writing specifications in VDM and generating code in Java, for example, has not become popular. Programmers feel uncomfortable working with two languages (a specification language and a programming language) when the two languages are too similar. This is an argument for merging the concepts into a specification, design and implementation language.

2.2 Model-based engineering

Model-based engineering includes modeling frameworks that are usually visual/graphical of nature. One of the main contributions in this field is UML for software development, and its derivatives, such as SysML for systems development. The graphical nature of the UML family of languages has caused it to become rather popular and wide-spread in engineering communities. Engineers are at first encounter more willing to work with graphical notations, such as class diagrams and state machines, than they are working with sets, lists and maps and function definitions. It seems clearly more accepted than formal methods as described in the previous section. At JPL for example, there are three people working with formal methods and over hundred working with UML/SysML technology.

One of the important notations in UML/SysML is class diagrams. Class diagrams are – just like E/R-diagrams – really a simple way of defining data, an alternative to working with sets, lists and maps as found in VDM and modern programming languages. For example, to state that a person can own zero or more cars one draws a box for Person and a box for Car and draws a line between them. It is an idea that quickly be picked up by a systems engineer, quicker than learn to program with sets. Another notation is that of state machines, a concept that strangely enough has not found its way into programming languages, in spite of its usefulness in especially embedded programming. UML/SysML also focuses to some extent on requirements,

a concept that usually is not embedded as a first class object in programming. It would be interesting to see requirements as part of programming.

The above observations are rather positive. However, UML/SysML are very complex and weakly defined formalisms. The combined syntax for all UML for example corresponds to the sum of approximately 20 programming languages (approximately 17,000 lines of abstract syntax, a programming language can normally be defined in between 500 and 2500 lines of grammar rules). The UML/SysML standards are long and complex documents. The connection between models and code is fragile, relying on the correctness of translators from for example UML state machines to code.

3 Observed concepts in programming

Several new programming languages have emerged over the last decade, which include abstraction mechanisms known from the formal specification languages mentioned above. Such languages include Eiffel, Java, Python, Scala, Fortress, C#, Spec#, F#, Dafny, D, RUST, Swift, Go, Agda, and SPARK. Some languages support design-by-contract with pre-post conditions, and in some cases with invariants. These languages include for example Eiffel, Spec#, Dafny, SPARK, and to some limited extent Scala. Java supports contracts through JML, which, however, is not integrated with Java, but an add-on comment language (JML specifications are comments in a Java program). Most of the languages above support abstract collections such as sets, lists and maps. It is interesting to observe that SUN's Fortress language (which unfortunately was not finished by the designers) supports a mathematical notation for collections very similar to VDM. The Dafny language is interesting since it is developed specifically with specification and verification in mind.

A trend on the rise is likely the combination of object oriented and functional programming, as seen in perhaps most prominently Scala, but also in the earlier Python, and now in Java which got closures in version 1.8. Ocaml is a similar earlier attempt to integrate object oriented and functional programming, although in a layered manner, and not integrated with the standard module system. Some interesting new directions of research include dependent types as found in Agda (to some extent related to predicate subtypes in VDM) and session types. Session types are temporal patterns that can be checked at compile time. They are much related to temporal logic as used within the formal methods community to express properties of concurrent programs. At the same time there are also attempts to make more conservative moves away from C, but without losing too much efficiency. Examples include the languages D and RUST.

4 Requirements for a programming language

A specification, design and implementation language will have to serve quite different goals. First of all, it has to represent the concepts of the application domain at an adequate level of abstraction such that the specialities of the applications domains are directly represented and not covered by awkward implementation concepts. This may for example include support for user-defined extensions of the language with domain-specific languages (DSLs). Second it has to address the structuring of algorithms and data structures in a way such that programs stay understandable, modular and support the most important methods of structured program development. And finally it has to allow addressing specific implementation properties of execution machines including their operating systems, such that it can be controlled how the implementation uses resources and exploits the possibilities of the execution platform and its hardware.

These three different goals are clearly in some contradiction. Nevertheless in a piece of software all three goals have to be addressed. What we would like to have is a specification, design, and implementation language, which represents the concepts of the application domain as adequate as possible, which allows at the same time to structure the software in a readable and manageable form, and which allows addressing of particular execution concepts on the execution platform and the machine to the extent needed.

An obvious problem here of course is to what extent then the particular application domain influences the programming languages, and to what extent this is true for the execution platform and also for the different forms of structured design concepts. In this section we shall try to outline what we see as the desired requirements of a programming language.

4.1 Target domain

We can observe three major domains of interest, namely modeling; programming of non-critical systems, such as web applications, including scripting; and finally programming of embedded/cyber-physical systems. It is clear that these three domains till date have been addressed by different communities and different languages, as outlined above. Our goal is to address the three domains in one language. Such a goal usually provokes a reactions which can be summarized as “a silver bullet does not exist” and “each problem requires its own solution”. However, we do question this constraining view based on the observation that all of the languages above share a big set of language constructs.

4.2 Support for modeling

This item is less well defined, and generally means support for modeling and understanding a problem in addition to programming the solution. This is about combining modeling and programming into one language. No separate UML models etc. It includes design-by-contract as we know it, including pre- and post-conditions, as well as class invariants. Such can for example be found in Eiffel as well as in SPARK. However, we believe that it can be carried further to for example include such topics as temporal logic, program monitoring, program visualization, including diagramming of static structure as well as dynamic behavior, as built in concepts, and of course verification to the point where it is practical, unit testing built in as in Fortress, etc.

4.3 Design and architecture: programming in the large

Large programs have to be structured. They have to be structured on one hand in independent or at least rather independent pieces that can be reused, independently changed, translated and executed on different hardware, such that a flexible deployment is achieved. They have to offer appropriate techniques for encapsulation and parameterisation. This structuring may also address issues of execution such as deployment and parallel execution.

4.3.1 Components, modularity and encapsulation

What is needed, in particular, is an appropriate notion of component as a unit of modularity and encapsulation. Such concepts exist in a lot of programming languages. However since most of the programming languages we are using today are inherently sequential, an independent deployment and execution model often is not directly achieved.

4.3.2 Variability

A key to efficient software evolution is the identification of components that can be used and reused at several places in a program. This requires a sufficient amount of variability. If such variability cannot be achieved then code cannot be reused at many places and as a result we have to form clones, meaning similar pieces of code with just small differences, such that they can be used at the individual places. Another issue for variability is the usage of different variations of software in the context of software families. Variability is an important concept, which is not very much supported explicitly by nowadays programming languages.

4.4 Support for high-level programming

The specification, design, and implementation notation has to be sufficiently abstract. Many formalisms used and suggested for that including a lot of the

programming languages force the programmers to write too many details enforcing a particular style, which is related to a way to describe algorithms. Therefore the resulting programs get very long and more difficult to understand. The key question is how to provide programming concepts that are expressive, understandable, and do not enforce the explicit formulation of a lot of details due to a particular algorithmic style.

4.4.1 Merging object-oriented and functional programming

The elegance and the implicitly of functional programs have been praised many times. Nevertheless they never had an absolute breakthrough. In contrast, object oriented programming languages, which in particular address encapsulation and reuse were very successful. They provide entities of implementation called classes, which at the same time are able to present concepts in the application domain and units of execution. However, for all nowadays object oriented programming languages there are a number of properties, which do not allow using them in the required universal modelling style. One reason for that is that object oriented programming languages are inherently sequential due to their remote procedure call concept. All attempts to provide parallel execution models such as threads make things ugly and very complex. Therefore a good idea would be to use many of the good ideas in object oriented and functional programming and bringing them together in powerful generalizations. A language such as Scala has made this attempt, and even early versions of LISP had this (CLOS). Functional programming means for example functions as values (lambda abstractions) and pattern matching, and of course reliance on recursion. Functional programming is by some considered the best approach to use multi-core systems due to no shared state updates.

4.4.2 Built-in collections

Perhaps specialized notation for these as in Fortress (very similar to VDM), or as libraries. Easy ways of iterating through collections – to avoid indexing problems for example. Support for parallel computation over such.

4.4.3 Domain-specific data modeling

A key to programming is to capture the relevant concept of the application domain and to present them in the specification design and implementation notation. This is exactly where UML and also SysML are quite successful. They provide a number of concepts which originally were created in the area of programming and good enough to allow presenting quite a number of application domain issues. A typical examples are class style diagrams, which at a level of programming are describing more or less architectures in terms of classes and how they are connected, but can

also be used as possibilities to describe data models and finally ontologies as we find them under the heading of meta models. In any case it is important to support powerful modelling approaches in the specification, design and implementation notation.

4.4.4 Typing and physical dimensions

We believe a language should be largely statically typed. It can potentially allow for going type-less in clearly defined regions, in case such makes modeling and programming easier. Scripting languages are popular, in part because they are type-less. It would be interesting to see if one could allow both approaches to be used within the same language. Otherwise decades of experience in strong type systems should of course be harvested, including more recent topics such as dependent types, session types, and units.

4.5 Support for low-level programming

Embedded programing these days often means: no dynamic memory allocation after initialization, no garbage collection, some knowledge of memory layout, even to the point where computation with addresses is used to improve speed. This again means use of low level programming languages such as C. C, however, allows for memory errors and makes programmers less effective as they would otherwise be were they allowed to program in higher-level languages. We need to satisfy the needs encountered by typical C programmers, including offering comparable speed and memory control. This includes support for hardware control.

4.6 Concurrency

Concurrency is an essential part of modern programming, especially considering the emergence of multi-core computers. However, concurrency is important at the modeling level as well, where it can serve as a natural way to describe interacting agents. Important concepts include agent systems, message passing based communication, parallel data structures (programming concurrent without knowing it), and distributed programming.

4.7 Execution model

The programming notation has to allow to control execution aspects. Often in today's practice, extensions are introduced that allow controlling execution platforms. This is important in order to provide programs that are efficient. On the other hand, it is very dangerous since it mixes up domain specific concepts, data modelling, and algorithms on one hand, and issues of specific execution concepts of

the execution platform. Such a mix also makes it very difficult to port and migrate software. A promising approach could be that the specification, design, and implementation notation provides possibilities to target a specific execution platform by separate profiles that are in addition to the description of the domain specific concepts in the algorithms. This idea could be applied for time, concurrency, deployment and distribution.

4.8 Analysis

A key concept in a combined modeling and programming environment is the support for advanced analysis of models/programs, including, but also beyond, what is normally supported in standard programming environments. This ranges from basic built-in support for unit testing, over advanced testing capabilities, including test input generation and monitoring, to concepts such as static analysis, model checking, theorem proving and symbolic execution. A core requirement, however, must be the practicality of these solutions. The main emphasis should be put on automation. The average user should be able to benefit from automated verification, without having to do manual proofs. However, support for manual theorem proving should also be possible, for example for core critical algorithms. Integration of static and dynamic analysis will be desirable: verify what is practically feasible, and test (monitor) the remaining proof obligations.

5 Conclusion

We have in this document outlined some views on the potential in combining modeling and programming, supported by analysis capabilities such as static analysis, model checking, theorem proving, monitoring, and testing. We believe that the time is right for the formal methods/modeling and programming language communities to join forces. To some extent this is already happening in the small. However, we believe that we are standing in front of a major wave of research creating a united foundation for modeling, programming and verification. A cynical argument is that this is all obvious, which may be true.

6 References

6.1 Modeling

- CIP: http://en.wikipedia.org/wiki/Wide-spectrum_language
- Coq: https://coq.inria.fr
- JML: http://www.eecs.ucf.edu/~leavens/JML//index.shtml

- Mathematica: http://www.wolfram.com/mathematica
- Modelica: https://www.modelica.org
- OBJ: http://c2.com/cgi/wiki?ObjLanguage
- PVS: http://pvs.csl.sri.com
- RAISE: http://spd-web.terma.com/Projects/RAISE
- SysML: http://www.omgsysml.org
- TLA: http://research.microsoft.com/en-us/um/people/lamport/tla/tla.html
- UML: http://www.uml.org
- VDM: http://en.wikipedia.org/wiki/Vienna_Development_Method
- Z: http://en.wikipedia.org/wiki/Z_notation

6.2 Programming

- Agda: http://wiki.portal.chalmers.se/agda/pmwiki.php
- C: http://en.wikipedia.org/wiki/C_(programming_language)
- C#: https://msdn.microsoft.com/en-us/library/67ef8sbd.aspx
- D: http://dlang.org
- Dafny: http://research.microsoft.com/en-us/projects/dafny
- Eiffel: https://www.eiffel.com
- F#: http://fsharp.org
- Fortress: java.net/projects/projectfortress/pages/Home
- Go: https://golang.org
- Haskell: https://www.haskell.org
- Java: https://www.oracle.com/java/index.html
- JavaScript: http://www.w3schools.com/js/
- LISP (CLOS): http://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html
- Ocaml: http://caml.inria.fr/ocaml
- Python: https://www.python.org
- Ruby: https://www.ruby-lang.org/en
- RUST: http://www.rust-lang.org
- Scala: http://www.scala-lang.org
- SPARK: http://libre.adacore.com/tools/spark-gpl-edition
- Spec#: http://research.microsoft.com/en-us/projects/specsharp
- Swift: https://developer.apple.com/swift