A Development Language

Klaus Havelund and Rahul Kumar

Jet Propulsion Laboratory
California Institute of Technology
California, USA

In this white paper we suggest to build a language, named NotC, for developing software systems at a high level of abstraction as well as at a low level of implementation. The language should allow a user for example to first specify a system at a high level, and then at later stages refine this specification into a detailed program, all within the same language. Checks and balances should be performed to ensure that the detailed refinement implements the abstraction. Such checks can be performed statically as well as dynamically. That is, NotC language should be supported by analysis tools of various kinds, including static analysis, verification, and dynamic analysis (testing/runtime verification). In general it should be a vehicle for trying out all our ideas about how to write correct programs fast.

1 Background

The work is inspired by various sources. First of all, early days of formal methods produced specification languages such as VDM and Z. VDM++ (a variant of VDM) for example offers a wide-spectrum **specification** language that contain features such as functional programming/modeling with higher order functions; algebraic data types; pattern matching; object oriented programming/modeling with variables; assignment statements; loops etc; design by contract with pre/post conditions and class invariants; and general higher order predicate logic for writing Boolean expressions. VDM++ never became mainstream, although it was one of the main formal methods before theorem proving and model checking became popular. One reason for not lasting was probably in part that it was a specification language, and not a programming language, and in part of course, because there was no serious tool support. Proofs, if any, would have to be carried out manually. Our objective is to turn this around and define a wide-spectrum **programming** language inspired by the same ideas, and use the latest results in automated verification as basis for tools.

A second inspiration is our current work to provide a textual language, named K, for SysML , to be used to design the proposed Europa Clipper mission. SysML is a graphical language for creating models of systems (in contrast to models of software, as is the target of UML). SysML contains selected elements of UML but also extends it. SysML engineers recognize that there is a need for a textual modeling language in addition to the graphical, such that ideally one can create/view a model in either format, depending on taste and situation. As it turns out, the SysML team at JPL has asked for a textual language very much along the lines of VDM and Z. This in itself is interesting: that the SysML/UML community sees a need for the kind of languages that formal methods came up with decades ago. The main challenge here is to make the relational view adopted in SysML (there are only atomic objects and relations between them) co-exist with the data type view on most other languages, where there are data types such as collections (sets, lists and maps). This challenge corresponds to combining traditional programming with database/logic programming.

A third inspiration is the current trend in programming language design, which seems to move towards the combination of object oriented and functional programming with high-level concepts such as collections (sets, lists, and maps), iterators, and generally all the concepts from VDM++ except general predicate logic in Boolean expressions. There has also been programming languages in the past, such as SETL, which focused on set theory, also an inspiration.

Finally, the work of Mihai Florian guided by Gerard Holzmann was in this line, although there was less emphasis on high-level programming. The focus was exactly to design a programming language with specification constructs, and with a verification environment around it. Similar attempts are made with languages such as Spec# and Dafny from MSR. One might also mention the work on SPOT here at JPL, and perhaps even unite with that work.

All in all, attempts have been made are being made in this direction. Yet, it is an interesting avenue of research that we could take on.

2 The Language

We would consider the current K language (textual version of SysML) as our starting point, or we simply consider the K language as the language, and as a group collectively focus on supporting the Europa project, the latter probably being more practical. The Europa project wants this language, they claim they rely on it to *"clean up the SysML mess"*. There is a certain pressure to get this right.

K is so-far designed based on all the inspirations mentioned above. Whether NotC = K or whether NotC a new language can be a discussion point. The language should be one that each of us would want to program our systems in. Hence it should be as fast as C when staying in the low level fragment. We would discuss which fundamental paradigm the language should built on: object-oriented? functional?, logic-programming?, data-base technology? (currently K has relations, which resemble relational databases). We would discuss whether it will make sense to let the language allow abstract specification as well as low-level C-like programming. Is it possible to merge these concepts? Ocaml is claimed to be as fast as C and combines the two levels. Garbage collection could get in the way.

The language should include:

– Programming constructs:
  - object-oriented features, including classes and objects, variables, assignment (side effects), sequential composition, loops, etc, standard stuff.

  - functional programing, including functions as values, algebraic data types, pattern matching.
  - Built-in collections such as sets, lists and maps, including iterator constructs over such. Can be libraries, but special syntax for sets, lists and maps is attractive.
  - Should be garbage-collected for ease of programming. However, this is a challenge in the context of embedded systems. Will require research.
  - Concurrency constructs inspired by modern actor frameworks (message passing).
  - Support for extending the language and define DSLs. This includes serious reflection capabilities, such that a program can examine itself. For example, a coding standard can just be a library one imports.
  - The work on the relational view in SysML opens the question whether it would be useful with a graph-model as part of the language: relations, logic-programming.
- Specification constructs:
  - pre/post conditions of functions.
  - class invariants.
  - temporal logic-like specifications over execution traces (can be rule-based, regular expressions, whatever is deemed most useful).
  - General predicate logic for writing Boolean expressions.
  - refinement: the language should support the notion of refinement within the language: for example that a class implements another class, resulting in a proof-obligation to show that it is true. That can be checked statically or dynamically during execution.

3 The Tools

The language should come with a collection of tools:

- compiler, compiling to a high-level language, such as Scala or C for prototyping purposes, and later to low-level machine code such as LLVM.
- verifier, including static analysis, model checking, theorem proving (Dafny style), etc.
- tester, including runtime verification, test-case generation, and unit-testing. Testing would include machine learning: learning specifications from execution traces, which can later be turned around to monitors, be visualized, or even perhaps proven against the code using some form of theorem proving.
- visualizer, static (of program structure) as well as dynamic (of execution traces). The visualization can be interactive, such that code can also be generated from graphics. It is important, however, that the code is the main representation form, which can be visualized.