A Development Language

Klaus Havelund and Rahul Kumar

Jet Propulsion Laboratory
California Institute of Technology
California, USA

We propose to build a language, named K, 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, K language should be supported by analysis tools of various kinds, including static analysis, verification, and dynamic analysis (testing/runtime verification). In general, K should be a vehicle and platform for experimenting with various ideas related to developing 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. 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 present in 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.

Finally, we also draw inspiration from related work done by Mihai Florian and Gerard Holzmann at JPL. Their work laid less emphasis on high level programming but more on specification constructs with a verification environment surrounding them.

2 The Language

K is intended to be a wide-spectrum language that supports various scenario such as programming, modeling, specifcation, or a combination of the same. We now list the features of the language that we believe will make it possible for K to be a wide-spectrum language:

- 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 and invariant specifications for functions
  - class invariants.
  - temporal logic-like specifications over execution traces (can be rule-based, regular expressions, etc.).
  - 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

Apart from the language itself, we envision the development of a suite of tools that enable various scenarios. Some of these include:

Compiler for producing executable code. Initially, for prototyping, this may be to a high-level language, such as Scala or C, and later to low-level machine code such using the LLVM framework.

Verifiers 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.

4 Development Plan

We propose the development of the language in three primary stages that are described below:

Language Design stage where the language syntax, constructs, and features are developed to maturity.

Language Development stage where the language is used to develop models, programs, and specifications. These most likely include the Europa Clipper mission models and specifications. Any feedback from language users would also be reviewed and integrated back into the language.

Tool Development stage where we develop the tools listed above.