Languages for Systems not Software

Peter Amey
Praxis Critical Systems
pna@praxis-cs.co.uk

Extended Abstract

Historically programming languages have largely been concerned with the coding of designs. The facilities of the language are wholly concerned with the software realm; expressing the behaviour of the system of which software forms a part is the role of some other notation such as UML, Z or HOOD. This is inevitable given the genesis of most programming languages which all grew out of machine code and were focussed on providing more powerful and abstract ways of expressing the behaviour of software.

In parallel with the development of programming languages has come the development of program analysis tools. Originally these were concerned with the reverse engineering of legacy systems to try and understand and evaluate their behaviour. Examples of this approach are MALPAS and SPADE form the UK and some of the more recent ASIS-based tools which have been developed. After developing SPADE, Program Validation Ltd (now wholly absorbed into Praxis Critical Systems) noted that experience with analysis tools indicated that the real issue was encouragement for correct design in the first place. Well designed systems were straightforward to analyse whereas poorly designed ones defied even the most powerful analysis tools. The answer was therefore not ever more powerful analysis techniques but early deployment of analysis as an integral part of the development process leading to “correctness by construction”. This philosophy was embodied in the SPARK language and its analysis tool the SPARK Examiner. SPARK defines a programming language with completely unambiguous semantics which is amenable to rigorous analysis. Furthermore SPARK facilitates early use of static analysis in the development process because it includes features allowing the analysis of incomplete program fragments.

Early in the design of SPARK language authors recognised the need to provide more information to the analysis tools than could be gleaned just from executable Ada statements. To this end annotations in the form of “formal comments” are provided; these are ignored by the compiler but convey information to the SPARK Examiner. SPARK’s annotations were introduced for two purposes: to allow detection of certain language abuses (such as aliasing or function side-effects) in an efficient manner; and to allow the analysis of partial systems as described above. In effect the annotations strengthen the specification of packages and subprograms so that bodies are not required when analysing the uses made of them. Initially SPARK annotations also took a very software-centred view of the information that was required. For example, to analyse uses made of a package that contains static data or “state” we need to know that the package does indeed contain state. In SPARK this is done with an “own variable” annotation placed in the package specification (the term “own variable” was borrowed from Algol 60). In effect this states that the package owns the variable(s) named in the annotation. Initially this was seen simply as way of promoting the variable’s visibility for analysis purposes; i.e. making it visible to the Examiner
without requiring it to be declared in the package specification. Further evolution of SPARK led to the realisation that effective analysis only required an indication that the package contained some state, full details of the form that state took in the body was not required. To support this the own variable concept was extended (in about 1992) to include the idea of abstract own variables. These are names, chosen by the software designer, to describe the state whose concrete details which remain hidden in the package body. A refinement annotation in the body binds the actual package data variables to the abstract name. Thus a stack package can admit that it contains some state without revealing that it comprises an array and a pointer, or perhaps a linked list which actually implements the stack.

The introduction of abstract own variables is a crucial step in the evolution of a language which describes systems rather than just software. For the first time we have, in an annotation, an entity which cannot be found anywhere in the compilable Ada statements of the program. To continue the above example “Stack.State” is a design concept; since there is no Ada variable called State in the package, the annotation is doing more than just re-stating information deducible from the code.

A second strand in the development of SPARK as a system language was the discovery that SPARK annotations provided a powerful design quality metric and could be used to guide design rather than just to facilitate analysis of software. For example, the derives annotation, which describes interactions between the flows of information into and out of a subprogram is a highly sensitive measure of the degree of coupling between software components. Since loose coupling is a design goal we can use the size of derives annotation as a direct design driver; small annotations are easy to write and maintain and are also an indication of a well-designed system. These ideas are incorporated into the INFORMED design method which makes minimisation of information flow a primary design goal.

The final link in chain is the way that SPARK users use abstract own variables to provide a description of entities in the environment with which the SPARK software interacts. This approach exploits the fact the abstract own variable have a name chosen by the software designer and that the Examiner can analyse uses of an annotated package without needing to see its body. For packages handling devices on the periphery of the system such as physical sensors and actuators we can provide an own variable clause giving a suitable name of our choice that represents that external device. Furthermore, by suitably annotating operations involving that device we can capture the property of volatility associated with such entities; i.e. the property that successive reads from a memory-mapped inputs may return different values because they are being changed by something outside of and invisible at the purely software level.

To support this important use of the own variable concept, the SPARK language has recently been extended again to allow abstract own variables to be marked as system-level inputs or outputs. This is done by adding modes to the own variable annotation. So “---# in Sensor” defines an abstract own variable which is identified as carrying information into the SPARK program from the external environment. The SPARK Examiner is in the process of being modified to recognise the volatility of such external variables and perform appropriate analysis on them.
We now have all the tools we require to describe the behaviour of software-intensive system that interacts with external devices. The executable Ada statements provide the operational semantics for a machine which we believe will have the desired behaviour. The SPARK annotations provide a parallel, abstract, system-oriented description of the system. The rules of SPARK and the analysis performed by the Examiner bind the two descriptions together. For example, the top level derives annotation for a trivial engine control system might read:


despite the fact that none of these names appear in the program as Ada variables. Pressure.Value, for example, is an abstract name of our choosing representing perhaps three memory-mapped Ada variables (Registers_1, _2 and _3 for example), together with some smoothing and calibration routines. The consistency of the abstract, system-oriented description in the annotations and its embodiment in executable statements can be confirmed by rigorous analysis including proof.

The full paper expands on the idea of SPARK as a language which provides a simultaneous system description and software implementation and includes examples of the technique in practice.