Executable UML and SPARK Ada: The Best of Both Worlds

Dr. Ian Wilkie
Technical Director, Kennedy Carter Ltd. Guildford, U.K.
ian.wilkie@kc.com, www.kc.com

Motivation

The demands of high integrity and safety critical systems development have lead to the use of a number of different formal specification and design techniques such as Z and SPARK together with well designed and understood implementation languages such as Ada.

While considerable success has been achieved using these tools it is recognised that the more abstract and mathematical techniques present a steep learning curve for developers and this has perhaps hindered their uptake. At the same time the wider software development industry has been making use of a number of new techniques. In particular the use of diagrammatic modelling techniques has emerged from the early days of relatively imprecise formalisms such as Structured Analysis to much more well defined formalism such as the Unified Modelling Language (UML).

The work described in this paper emerged from the desire to explore the possible benefits of the more formal approaches with the accessibility of visual modelling.

Executable UML

Although the UML formalism provides more precision in its definition and semantics than many early formalisms there are still areas where the precise meaning of models will be open to interpretation. Further, the UML itself does not prescribe or define any particular development *process* to be used.

Executable UML addresses both these issues and defines a coherent *subset* of UML to form a core *executable* language. This is achieved by omitting features of UML that either on their own, or in combination, have undefined or poorly defined semantics. In addition, Executable UML provides an *Action Language* (ASL) that is used to define the state entry actions in the state machines and in the methods of the operations. This ensures that the models can be computationally complete.

The xUML method is supported by a well defined development *process* that provides a specification of modelling techniques and deliverables. There are two key aspects to the process. The first is a strategy that supports the partitioning of the models into multiple separate domains each of which deals with a separate subject matter. This separation enhances both the reuse and the *platform independence* of the models.

The second key aspect to the process is that, since the models can be executed, they can not only be tested for correct operation but can also be systematically translated into target code for production. This systematic translation (code generation) can be performed by automatic translation engines and has been successfully used on a large number of projects ranging from small single task embedded systems up to large distributed systems and using languages as diverse as Assembler, C, C++, Fortran and Ada. This code generation thus further enhances the *platform independence* of the models.

Executable UML is supported by a number of toolsets. This project used the iUML suite from Kennedy Carter.

The SPARK Approach

In an interesting parallel with Executable UML, SPARK defines a “safe” Ada subset in which a number of Ada features (for example Access Types) are not permitted. The main reason for excluding these is that they hinder the execution of various forms of static analysis that might otherwise be performed on the source code.

The SPARK approach also provides a set of annotations, expressed as Ada comments with a defined syntax, which make assertions about the content of the code. These assertions can then be cross checked against the operational Ada code by static analysis.

For example, in the following SPARK Ada:

```
procedure Multiply (A, B: in Float;
                    RESULT out Float)
--# derives RESULT from A, B;
is
begin
  RESULT := A * B;
end Multiply;
```

the line beginning with the text “`--#`” is an annotation that asserts that the procedure in which it is embedded derives the value of the “`RESULT`” output parameter from the values of the input parameters “`A`” and “`B`”. The SPARK examiner can then (through static analysis) cross check this assertion against the body of the procedure.

Now, clearly, this cross-check has no value if the programmer first writes the code and then simply annotates it in order to satisfy the examiner. Instead, the SPARK process emphasises the activity of Software Design prior to implementation. In this process a designer will, for example, design the module hierarchy of the system first. The designer will then annotate the hierarchy with, in effect, a *specification* of what the hierarchy is designed to do. Finally, the internals of the modules in the hierarchy are implemented.

Annotations are provided to allow the examiner to perform:

- Data flow analysis (e.g. finding variables initialised before use)
- Information flow analysis (e.g. finding deviation of the code from the "specification" formed by the annotations)
- Proof of absence of run-time errors

The Hybrid Approach

The work undertaken in this project attempted to marry these two approaches together and to:

- Use SPARK Ada as the target language for the implementation of executable UML models
- Implement an analogous process to the SPARK design process whereby the Executable UML models are annotated early on in the development process, prior to their full elaboration with Action Language.

This has the advantages of providing the accessibility of the Executable UML approach, while imposing the discipline and rigour of the SPARK approach on both the modelling activity and on the code generator. Further, the approach uses a tried and trusted tool such as the SPARK examiner to assess the characteristics of the target code.

The Technical Challenges

There were a number of technical challenges in the implementation of this approach:

- An analogous process to the SPARK design process had to be devised taking into account the difference in structure from a typical xUML model to an Ada module hierarchy.
- Suitable mappings from the xUML modelling formalism to the comparatively restrictive and extremely types SPARK Ada subset had to be developed.
- The transformation had to take into account constraints emanating from the requirements of the SPARK examiner for how the SPARK Ada code is arranged. The Ada packages and the code within them must be arranged so that the examiner has access to all of the information that it needs. For example, forward procedure references are not permitted in code bodies, even if the specs are already visible. A particularly difficult problem arose from a conflict between the xUML domain partitioning (which encourages information hiding) and the SPARK requirement for full examination of all code.

Progress

A hybrid process has been defined and a prototype code generator has been constructed that translates xUML models into (partial) SPARK ADA code. This code generator not only translates the necessary structural and behavioural elements of the xUML model that enable model execution, but also translates modeller supplied annotation in the xUML model into suitable SPARK annotations in the generated code.

Conclusions

We have demonstrated that it is possible to define an enhanced xUML process that applies the SPARK ideas to the xUML model. Mappings from xUML (with associated annotations) to SPARK Ada have been defined and partially implemented in a prototype code generator. We believe that all of the technical difficulties with this approach have been identified and eliminated.

The result is that by following the enhanced process, xUML models can be automatically verified against their initial specification. This significantly enhances confidence in the correctness of the models. Systems can them be implemented by using the generated SPARK code or by using alternative code generators.

Acknowledgements

This paper describes work carried out by Damian Curtis, Wilson Ifill, Alun Lewis and Colin Marsh of AWE plc, Janet Barnes of Praxis High Integrity Systems and by Mike Finn and Jeff Terrell of Kennedy Carter Ltd. We gratefully acknowledge the support by AWE plc of this work.