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

Abstract

Executable UML is a well defined UML subset supported by an Action Language that enables the construction of executable models from which reliable target code can be automatically generated. SPARK Ada is a safe Ada subset with formal annotations that renders programs amenable to static analysis and formal verification. This paper describes a hybrid approach where formally annotated Executable UML (xUML) models are automatically transformed into annotated SPARK programs. The resulting programs can then be examined using existing and trusted SPARK tools in order to infer the correctness of the source UML model

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 [1] 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 [2] to much more well defined formalism such as the Unified Modelling Language (UML) [3].

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 *method* to be used. *Executable UML* [4, 5] addresses both these issues.

Executable UML 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.

The core formalism provides for the specification of systems in terms of classes, each capable of providing a number of synchronously invoked operations and up to one asynchronously executing state machine. The state machines eschew the more complex possibilities in UML in favour of flat Moore style machines.

In addition, Executable UML provides an *Action Language* (ASL) [6] 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. ASL is an example of a concrete surface language that is compatible with the UML Action Semantics standard.

Executable UML also supports the partitioning of the system model into a number of components, termed domains. Each domain has its own name and type space and deals with a separate subject matter within the system. Typical domains might include the application itself and those dealing with user interface and hardware I/O as well as generic and highly re-usable components such as logging. The Executable UML formalism provides the ability to define anonymous coupling between the domains through abstract interfaces. This enhances the replaceability and reusability of the lower level domains.

Figure 1 shows the overall model structure using the example of an automatic train control system. Fragments of the Action Language can be seen embedded in the model.

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 that the emphasis in the modelling of each domain is on the *problem domain* itself rather than the solution space. This is particularly evident at the class modelling stage when the focus is on capturing the entities that inhabit the problem domain and the relevant rules and policies that apply to them.

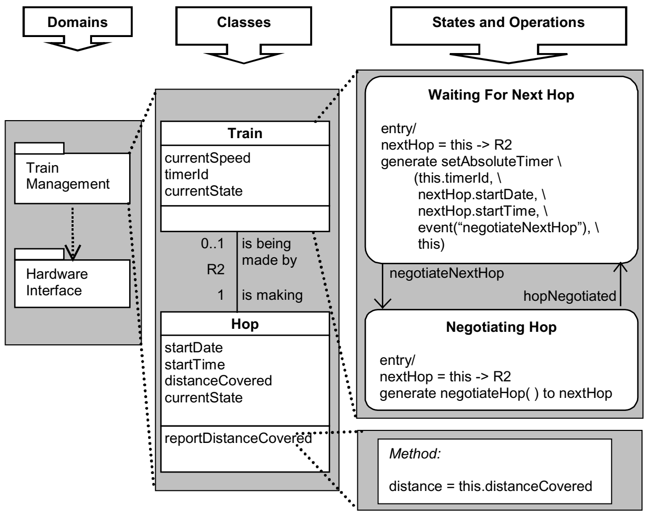

Figure 1 – xUML Model Structure

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 can be performed by automatic translation engines.

This technique can be, and has been, used to transform Executable UML models into target environments ranging from small single task embedded systems up to large distributed systems and using languages as diverse as Assembler, C, C++, Fortran and Ada. The Executable UML models thus have considerable *platform independence*.

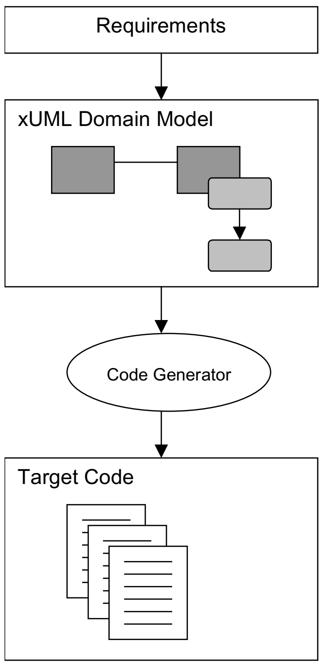

Figure 2 - Overall xUML Process

Interestingly, the problem of understanding the target environment and implementing the required translation is one that can *itself* be modelled using Executable UML. In this technique, software design is another subject matter for analysis.

This, in turn, means that the code generator shown in Figure 2 is, itself, built using executable UML[7]. Executable UML is supported by a number of toolsets. This project used the iUML suite from Kennedy Carter[8].

The SPARK Approach

In an interesting parallel with Executable UML, SPARK defines a "safe" Ada subset in which a number of Ada features:

- Unnamed Types
- Generics
- Dynamic Memory Allocation
- Nested Package Specs
- Access Types
- Variant Records
- Multi-tasking

are not permitted. The main reason for excluding these constructs is that they hinder the execution of various forms of static analysis that might otherwise be performed on the Ada 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 (which can statically analyse the code) will 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

Throughout the implementation process the SPARK examiner can be used to check the correctness of the program. A key aspect of this checking is that it can be performed on isolated program fragments given only the "specification" of what the missing parts of the program do. The implementation of the specification can then be checked a later stage.

There are a total of 11 annotations available to the SPARK user. Of these, two were of primary concern to this prototype project:

- Derives (defining derivation of outputs from inputs)
- Global (defining access to global data)

Use of these annotations enables the examiner to perform both data flow and information flow analysis.

Another important feature of SPARK is a very strong emphasis on type safety. For example, this means that an array may not be accessed by a variable of unconstrained Integer type. Instead it must be accessed by a variable of a type which has been explicitly declared to be an Integer constrained by the same range as the array declaration.

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
- Being able to use a tried and trusted tool such as the SPARK examiner to assess the characteristics of the target code.

The Challenges and Solutions

There were a number of technical challenges in the implementation of this approach and we discuss some in this section.

The first challenge was that an analogous process to the SPARK design process had to be found for creating the executable models. Although the similarity between the Ada notion of a procedure and the Executable UML notion of an operation is obvious, the Executable UML idea of a state machine is somewhat different to any direct concept in SPARK.

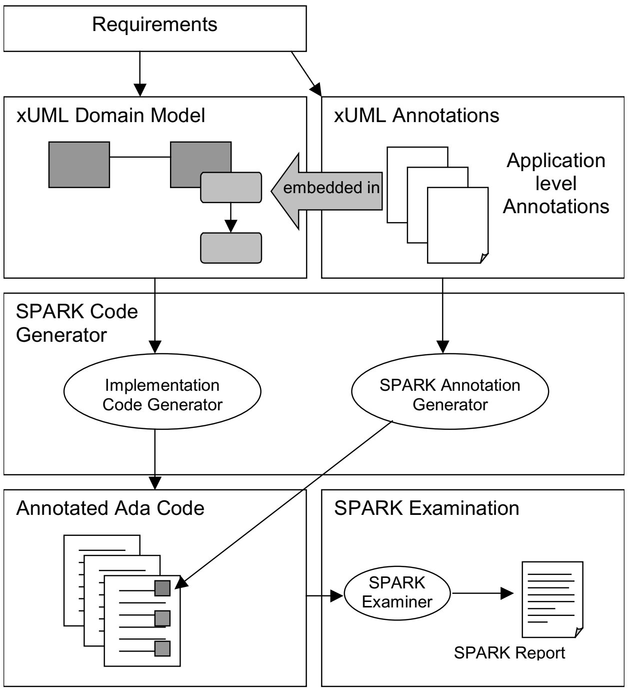

Figure 3 – Enhanced xUML Process

Figure 3 outlines the enhanced xUML process that was developed. In the normal xUML flow (shown down the left hand side of the diagram), the modeller develops the application level xUML domain model on the basis of the system requirements. The xUML model is then translated into operational code that implements the model on the target system.

In the modified approach, an additional flow is added (down the right hand side of the diagram). In this approach the modeller first develops an outline model on the basis of the requirements. This outline model (analogous to the module hierarchy of the SPARK approach) contains Classes, Operations and skeleton State Charts. It does not, however, contain any Action Language. The modeller then annotates the model with SPARK-like annotations that document the intent of the design. These annotations are "embedded" in the xUML model.

In this prototype project, the model can be annotated with the equivalent of the SPARK "derives" and "global" annotations. The annotations are applied to the operations in the xUML model, the state charts as a whole and to the actions of states within the state charts. This process is directly analogous to the SPARK activity of annotating the module hierarchy.

Following this annotation process, the modeller then completes the Action Language for the model.

The Code Generator then consists of two components. The first (on the left side of Figure 3) generates the functional code that implements the model in the usual way. The second (on the right side of Figure 3) takes the annotations found in the model and generates the equivalent SPARK annotations in the target code.

This annotated code is then run through the SPARK examiner in the usual way. Any problems picked up by the examiner can then be used to infer problems with either:

- The Action Language that completes the xUML model

or:

- The code generator itself

If the examiner finds no problems, this increases confidence both in the xUML model and the overall process.

The second challenge was that of code generating a model to the SPARK language. It was thought (naively) beforehand that the obvious restrictions in the SPARK language would present the biggest problem. However, this turned out not to be the case. Instead the "difficulties" arose from:

- The extreme type-safeness of SPARK
- The requirements that the SPARK examiner places on code layout within source code files.

The type-safe approach of SPARK forces the code generation down the route of generating large amounts of very specific (typed) code from the models. This is in contrast to the technique more frequently used with Executable UML which is to have a "generic" (model independent) run time library and generate relatively small amounts of code from the model to use the run time features.

For example, the ASL statement:

```
newRobot = create unique ROBOT
```

creates a new object in the class "ROBOT". In a typical "traditional" xUML implementation this might translate to a line of implementation code (in this case in "C") such as:

```
newRobot= (robotHandle)targetCreateObject
                    (CLASS_ROBOT, sizeof(S_ROBOT));
```

where `targetCreateObject` is a generic function that allocates enough memory for the object (the "sizeof" parameter) and appends it to a data structure identified by the "CLASS_ROBOT" parameter.

By contrast, the requirements of SPARK require an implementation such as:

```
ROBOT_DATA.CREATE(newRobot);
```

in which there is a separate creation routine for each different class.

The type-specific approach, of course, has the considerable advantage that many errors in the Code Generator are immediately visible at compile time rather than run time. The approach does have the disadvantage that it can lead to a significant increase in the code size of the resulting application.

Another problem is the inability of the SPARK examiner to trace accesses to individual components of records. A typical mapping from xUML would result in the use of record (or structure) types to hold object data. Such a mapping produces code which is which is compact and is quite intuitive to the casual reader. However, if that approach were employed here, the annotations to the xUML model could only be performed at a coarse level, i.e. the class as a whole rather than individual attributes. As a result, each attribute of a class in the xUML model was mapped to a separate data array in the SPARK code.

Interestingly this issue, combined with the lack of dynamic memory allocation, meant the gross structure of the generated SPARK was closest to that which had been previously used for generating embedded C, where the main emphasis was on saving as much dynamic memory as possible.

Another difficulty stems from the requirements of the SPARK examiner that the code be generated into a file structure which is not typical of Executable UML models.

For example, the examiner cannot deal with forward procedure references within individual source files (even if the "specs" for the procedure are already visible). This means that the code generator must analyse the "call tree" of operations within the xUML model and then generate the SPARK implementation of these operations in an order that eliminates any forward references.

Another such issue arises from a conflict between the domain partitioning strategy of xUML and the requirement of the SPARK process that, at some stage, *all* of the code in the system be fully analysed.

With domain partitioning, individual xUML models are constructed so as to be self-contained and individually executable. Interfaces to other domains (models) are represented by anonymous interfaces that are, in due course, bridged to (realised by) interfaces provided by other models.

This anonymous coupling enhances both the extent to which domain models can be re-used in different systems and to which models representing low-level technology can be replaced as the technology changes.

Unfortunately, a simplistic application of the SPARK approach would require that the domain models be explicitly annotated with detailed knowledge of the internal global data of all of the models to which it is bridged. This would compromise the whole domain partitioning rationale. Fortunately, SPARK has a notion of abstract state modelling which can be used to represent the state of the connected models without describing the details. Nevertheless, the resulting code must still be packaged carefully so that *all* of the code (including the xUML bridge code) is examined at some point in the development process.

As a result of these complexities, the full power of Executable UML analysis was used to model the code generator itself, with a domain concerned almost exclusively with how a user model is partitioned into source files.

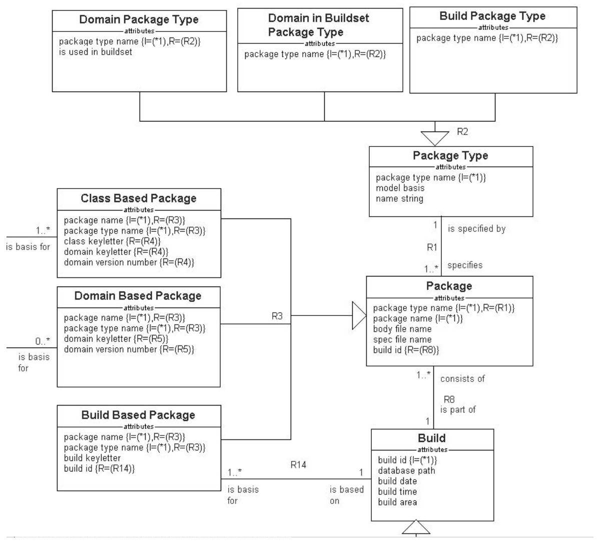

Figure 4 - Fragment of Internal Model of the Code Generator

Figure 4 shows part of the relevant model in the code generator. This fragment captures the different types of SPARK Ada packages that must be generated and their relationship to elements in the source xUML model. This code generator model is quite generic and is configured by instance data at the build-time of the code generator. This makes it comparatively straight forward to change the packaging strategy should it become necessary.

The Code Generator

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. This, therefore, supports the enhanced xUML process shown in Figure 3.

In the following example, user written annotations are attached to the state "`Idle`" of a state machine in the source xUML model:

```
--# global in out PTX.CURRENT_STATE,
--#      PTX.te, PTX.tf, PTX.tc;
--# derives PTX.CURRENT_STATE,
--# PTX.te, PTX.tf, PTX.tc
--#        from *, INSTANCE_HANDLE;
```

Here the modeller has stated that the action associated with the state to which this is attached will update the attributes "`te`", "`tf`" and "`tc`" of the xUML class "`PTX`" as well as changing the state of the state machine ("`CURRENT_STATE`"). The modeller has also stated that the action will derive the updated value of the globals from their existing values ("*") and from using the value of the object reference ("`INSTANCE_HANDLE`") pointing to the object which is executing the action.

When run through the code generator, this results in the following SPARK procedure definition for the state action:

```
procedure STATE_1_Idle (
  INSTANCE_HANDLE : in
  PSC_TYPES.PTX_IH_TYPE);
--# global in out
--#  PSC_PTX_DATA.CURRENT_STATE_ARRAY,
--#  PSC_PTX_DATA.te_ARRAY,
--#  PSC_PTX_DATA.tf_ARRAY,
--#  PSC_PTX_DATA.tc_ARRAY;
--# derives
--#  PSC_PTX_DATA.CURRENT_STATE_ARRAY,
--#  PSC_PTX_DATA.te_ARRAY,
--#  PSC_PTX_DATA.tf_ARRAY,
--#  PSC_PTX_DATA.tc_ARRAY from *,
--#  INSTANCE_HANDLE;
```

Note that xUML model references (e.g. "`PTX.te`") have been transformed into references to the appropriate data items in the generated code ("`PSC_PTX_DATA.te_ARRAY`").

This ensures that the annotations in xUML model are kept isolated from changes in the structure of the generated code and that the model level annotations can be understood with reference to the model alone.

In another example, the following ASL from an entry action in a state machine:

```
  nextCount = this.count + 1
  this.count = nextCount
  myPong = this -> R1
  if this.count > 10 then
       generate PING3:die() to this
  (…)
```

is translated to the following SPARK code:

```
procedure STATE_2_Sending(
 INSTANCE_HANDLE: in PP_TYPES.PING_IH_TYPE)
is
 -- Declare Local Variables
 ASL_nextCount: XUML.INTEGER_TYPE;
 ASL_nextCount_EXISTS:
               XUML.EXISTS_TYPE := FALSE;
 ASL_myPong: PP_TYPES.PONG_IH_TYPE;
 ASL_myPong_EXISTS:
               XUML.EXISTS_TYPE := FALSE;
begin
 -- Update current state of
 -- instance state machine
 PP_PING_DATA.WRITE_CURRENT_STATE
   (INSTANCE_HANDLE,
    PP_PING_DATA.STATE_2_Sending);
 -- Ada Translated From ASL:
 ASL_nextCount := PP_PING_DATA.READ_count
   (INSTANCE => INSTANCE_HANDLE) + 1;
 PP_PING_DATA.WRITE_count(INSTANCE =>
   INSTANCE_HANDLE, VALUE =>ASL_nextCount);
 PP_PING_DATA.NAVIGATE_ONE_R1 (
   FROM_INSTANCE => INSTANCE_HANDLE,
   FROM_INSTANCE_EXISTS => TRUE,
   LINK_EXISTS => ASL_myPong_EXISTS,
   TO_INTSANCE => ASL_myPong);
 if PP_PING_DATA.READ_count(INSTANCE =>
   INSTANCE_HANDLE) > 10 then
   PP_PING_QSIGNAL.GENERATE_die_Signal_Q
     (IH => INSTANCE_HANDLE,
     SELF_DIRECTED => TRUE);
(…)
```

Note that there is a significant transformation from the ASL into the resulting code. For example, accesses to attributes of objects (`this.count`) are transformed into accesses to the relevant implementation arrays:

```
    (PP_PING_DATA.READ_count(INSTANCE =>
                               INSTANCE_HANDLE))
```

The fact that the class data structures are managed as separate SPARK arrays is kept hidden from the ASL and thus maintains the platform independence of the models.

The resulting code from this code generator has been successfully run through the SPARK examiner.

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, by Mike Finn and Jeff Terrell of Kennedy Carter Ltd. and by Janet Barnes of Praxis High Integrity Systems. We gratefully acknowledge the support by AWE plc of this work.

References

[1] High Integrity Software: The SPARK Approach to Safety and Security, John Barnes, Addison-Wesley, 2003

[2] Structured Development for Real-Time Systems, Paul Ward and Stephen Mellor, Yourdon Press, 1985

[3] UML is managed by the Object Management Group: www.omg.org/technology/documents/modeling_spec_catalog.htm

[4] Model Driven Architecture with Executable UML, Chris Raistrick et al, Cambridge University Press 2004

[5] Executable UML: A Foundation for Model-Driven Architecture, Stephen Mellor and Marc Balcer, Addison-Wesley, 2002

[6] The Action Specification Language Reference Manual, Ian Wilkie et al, Kennedy Carter Ltd, 2003

[7] Configurable Code Generation in MDA using iCCG, Kennedy Carter 2002

[8] See www.kc.com