Modeling SPARK Systems with UML

Xavier Sautejeau
Sodius
6 rue de Cornouaille, BP 91 941
44319 Nantes Cedex 3, France
+33 (0)2.28.23.60.60
xsautejeau@sodius.com

ABSTRACT
In this paper, we will consider two aspects of UML in order to assess how well suited it is for modeling SPARK systems. The first aspect is the ability to represent SPARK in UML from a theoretical perspective. The second aspect is more from a hands-on perspective and evaluates what makes a UML CASE-tool more suitable for modeling SPARK systems than another.

Categories and Subject Descriptors
D.2.2 [**Software Engineering**]: Design Tools and Techniques – *computer-aided software engineering (CASE),*

D.2.1 [**Software Engineering**]: Requirements/Specifications – *languages.*

D.3.2 [**Programming Languages**]: Language Classifications – *data-flow languages, object-oriented languages.*

General Terms
Design, Languages

Keywords
SPARK, UML, Ada, profile, metamodel, INFORMED

1. INTRODUCTION
SPARK is a language derived from a subset of Ada which, through the addition of annotations to Ada packages and operations, provides strong static analysis of programs. We recommend the reading of [1] for an introduction to the subject.

UML [2] is a visual notation for modeling systems, and more specifically their object-oriented aspects. It is also a widely used industry standard.

The share of software in systems today is unprecedented. The consequences are that lives and economies depend on software. Very often, software failure means not only additional costs in terms of maintenance, but unacceptable human or financial losses. Preventing them is therefore a major development concern.

SIGAda'05, November 13–17, 2005, Atlanta, Georgia, USA.

One approach to preventing these failures is to detect defects through testing activities. Using technologies such as SPARK, there is an alternative, safer and more cost-effective approach which consists in preventing the creation of defects instead of hunting for them through simulation or execution. The underlying philosophy of this approach is called "Correctness by construction" [3].

The importance of models in software development, especially the ones based on UML is gaining more and more importance.

Combining the benefits of UML, such as visual expressiveness or the ability to transform or generate from its associated models, with the qualities of SPARK, such as its built-in static-analysis capabilities or its cost-saving virtues, is a need that naturally emerges from modern development best practices.

Hence the strategic business need from major actors in the embedded software market for a UML Tool supporting the modeling of SPARK specific features that is usable in their development process.

In this paper, after first discussing the features that the UML notation provides or lacks for modeling SPARK systems, we examine some evaluation criteria of CASE tools from a SPARK support perspective. We will then present an example of such a tool.

2. MAPPING UML TO SPARK
2.1 Readily available features
SPARK Ada programs are written in compliance with a grammar. UML models are created following the UML metamodel rules definition.

The first step in mapping UML to SPARK consists in finding a maximum of equivalences between SPARK language constructs and UML metamodel elements.

Because UML originated from the need to abstract systems implemented or to be implemented using object-oriented programming languages and because one of SPARK characteristics is to be one of these programming languages, there is a number of "natural" equivalences between the two languages for most of the features of SPARK.

Since these are way too numerous to list them all in this paper, we shall only list the main ones in table 1, our main point being to illustrate that a coarse-grained mapping is quite straightforward to obtain.

Table 1. Some possible equivalences from SPARK to UML

SPARK construct	UML model element
Package	Class | Package
Operation	Operation
Variable	Attribute
Annotation	Constraint | Dependency | TaggedValue | Stereotype

2.2 Using profiles to complete the mapping

"Pure" UML does not suffice to cover all the modeling needs. That is why profiles have been created. Profiles are a way of extending UML to integrate new paradigms.

A profile is a way to augment the descriptive power of a UML model via the addition of dedicated stereotypes, properties or constraints in order to embed additional characteristics of the system into the model.

Using profiles, we can refine the mapping between SPARK and UML. Table 2 gives a few examples of elements that could be part of a SPARK profile

Table 2. SPARK profile candidate elements

Profile element(s) [kind]	SPARK construct	UML construct
<<Proof>> [stereotype]	Proof type | Proof Operation	Type | Operation
<<OwnMode>> [tagged value, Enum (None, In, Out)]	Own variable mode	Attribute
globalSpec, globalBody [tagged value, String]	Global annotation(s)	Operation
<<Global>> [stereotype]	Global annotation(s)	Dependency (from operation to attribute)

2.3 Enhancing the fit

A priori, a good mapping between SPARK and UML can be defined for most of common constructs. However, there is still a gap between SPARK and UML that cannot be satisfactorily filled even using profiles. In this section, we will describe why some system aspects central to SPARK should be given more consideration in UML and how abstracting them away is penalizing in terms of model exploitation.

a) **Support for "write-only" operations**. UML is biased towards C++ in many aspects, but particularly in the domain of instance operation. In UML, one does not have to specify a parameter for the object an instance operation is applicable to (in C++, such a parameter is referred to as the implicit "this" parameter and is not required when declaring or calling an operation - in Ada or in SPARK this parameter has to be provided explicitly in the operation signature and when calling it). But this does not take into account some needs related to data/information-flow analysis, which are at the core of SPARK.

In UML, it is possible to set the "isQuery" attribute on an operation to indicate that the method does not modify the state of the object it is called on, in other words, the object is "read-only" (when mapping this to Ada or SPARK, it means that the explicit parameter denoting the current instance has a passing mode set to "in"). If this attribute is not set, it means that the object the operation is called on is modified, i.e. the object is not "read-only" (which in Ada or in SPARK is translated by a passing mode set to "in out" for the explicit current instance parameter). But in that case, this does not indicate if the previous state of the object is used to derive its new state, meaning the object is accessed in a "read-write" fashion (then "in out" is appropriate from an information-flow analysis point of view) or not, meaning the object is accessed in a "write-only" way (in that case "out" shall be used as the explicit current instance parameter passing mode).

The sample code in figure 1, taken from [1], is a partial specification of a Stack data type written in SPARK. Figure 2 is a UML view of the same Stack class. Figure 3 exposes the properties of some of the operations of this class from which we can derive the passing mode for the current instance parameter in the process of code generation. Although we mean to generate an "out" passing mode for the "Clear" operation, the UML does not provide us with a way to specify if the operation is "read-write" or "write-only".

```
package Stacks is

  type Stack is private;

  procedure Clear(
    S: out Stack
  );
  --# derives S from ;

  function Is_Empty (
    S : in Stack
  ) return Boolean;

  function Is_Full (
    S : in Stack
  ) return Boolean;

  procedure Pop (
    S : in out Stack;
    X : out Integer
  );
  --# derives S, X from S;

  procedure Push (
    S : in out Stack;
    X : in Integer
  );
  --# derives S from S, X;

end Stacks;
```

Figure 1. The stack data-type specification

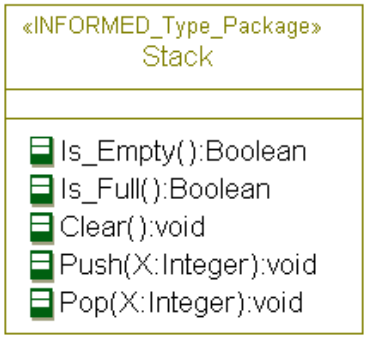

Figure 2. A UML model of the Stack class

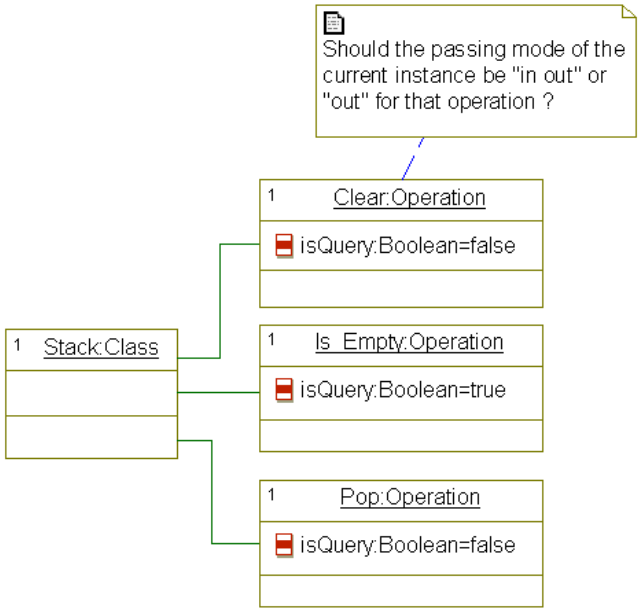

Figure 3. UML properties of some of the Stack class operations related to the passing mode of the current instance parameter

b) **Support for declaration order modeling**. A model is by definition an abstraction, meaning it hides irrelevant details. UML models being by essence graphical, the UML metamodel does not provide a way to fully specify the relative order of the elements they contain, which in a SPARK system is inherent to its syntax tree. Mapping the nodes of this tree to UML, which has been outlined in sections 2.1 and 2.2, must be completed with the mapping of its branches. Indeed, declaration order, which could be considered an "implementation detail" not worth the abstraction a few years ago, when the CASE tools code generators were not as powerful as today and when there was not as much code generated from them, is now a valuable model asset. Without declaration order information from the UML model, a code generator cannot guarantee that the generated source files will compile (perhaps more problematic in Ada and SPARK than with most other languages) or that information-flow analysis can be performed on it (a spoiler for SPARK).

Over the years, UML has integrated the ability to model some aspects of a system as the perceived value of representing them increased. We advocate that the integration of SPARK in a UML-based framework, the progresses made in the area of model based transformation and generation, and the increasing reliance of everyday activities on high integrity systems show evidence of the need to evolve the UML metamodel to provide better support for modeling information-flow constraints on systems.

Some of these constraints such as the the need to generate code from a set of model elements in a specific (i.e. analyzable in SPARK or compilable in Ada) order are relevant to implementation dependent modeling. Such constraints may eventually disappear depending on the actual generation target.

Some others belong to the business logic modeling; even if their automated analysis might only be possible after transforming the domain model in an implementation-dependent model, the modeling of these constraints provides added-value earlier in the development cycle.

2.4 Evaluating the mapping

Because of the flexibility of UML there is not one single possible mapping with SPARK but many. For example, one might prefer to use constraints on UML operations to represent annotations on SPARK operations over using tagged values because it is semantically closer. This is one evaluation criteria.

There are many others, but we think one of the most important is tool integration. The mapping may exploit all the nice features of the UML metamodel and profile capabilities, if these are not well supported by the tool used to develop the system, the usefulness of the mapping is to be questioned.

3. CASE TOOL support

This section does not pretend to be an exhaustive survey of what it takes to make a good CASE tool. The interested reader can refer to [4] for a comprehensive coverage of these features.

It focuses on the non-trivial features we consider a UML CASE tool should exhibit to support the efficient design of SPARK systems.

3.1 User-interface

As with source code, there are users who create models and users who read them. The needs for each set of users have to be taken into account.

Thus, in order to properly support profiles, a UML tool shall have a customizable/customized interface that facilitates the modeling of profile specific characteristics for a given model element by the model designer and that highlights them on the various diagrams it is represented on.

From the SPARK user standpoint, it means that a CASE tool shall provide a way to focus on the SPARK aspects of a model element when editing its features or when viewing it on a diagram.

3.2 Methodological support

If UML does not impose a predefined development process, a UML CASE tool should nevertheless be able to support the one chosen by the user.

There is a recommended process when designing SPARK systems, the INFORMED design methodology [5]. Although one does not have to follow this methodology by the letter to produce a working SPARK system, providing built-in support (as opposed to enforcement) is a nice feature to have.

3.3 Generators

All security and safety considerations excluded, a well designed system shall not require the user to input the same information more than once. So after the user has gone through the effort of modeling his SPARK system using UML, a modern CASE tool shall offer him the possibility to generate added-value from his model. The most common manifestations of this added-value in the context of model based generation in software development are documentation and source code. In the context of this paper we are more interested in the source code.

3.4 Compensating for UML shortcomings

As mentioned earlier, for a proper mapping from SPARK, even with a well-defined profile, UML is still lacking some features such as sufficient control over the declaration order of class members and over the information flow of instance level operations.

In order to support SPARK, a UML CASE tool must provide a way to overcome these shortcomings.

4. REAL-WORLD EXAMPLE

In collaboration with Praxis High Integrity Systems and with I-Logix, we produced a SPARK profile that can be used in conjunction with the Rhapsody in Ada UML CASE tool.

4.1 User interface

The UML formalism is not necessarily as concise as the one of the implementation language of the target system, especially in the case of well-designed languages such as SPARK. And when it is, modeling an element of a system the "UML-way" can be quite tedious, depending on the quality of both the SPARK to UML mapping and the CASE tool GUI.

The act of modeling must provide a return on investment (ROI) compensating the loss of productivity and flexibility gained from the use of an integrated development environment.

Because that ROI is not always positive, we strived to provide a mapping with different levels of granularity. For a same kind of annotation, the user has a choice of modeling it via a combination of dependencies, stereotypes and tags (which we qualify as a "model-oriented" style) or via a tag directly representing that annotation (which is more in the spirit of traditional development, and is qualified as a "capture-oriented" style).

For example, to model a package own annotation, the user can either directly type in the annotation to be generated at the package level (capturing the annotation) or specify which package attributes are to be generated as own variables by setting their individual tags appropriately (the annotation is not modeled per se, but generated from the model data).

4.2 INFORMED support

With Rhapsody in Ada, we were able to provide support for concepts defined in the INFORMED methodology, by adding dedicated stereotypes to the SPARK profile.

4.3 Code generator

Dedicated code generation rules are an essential part of the support. We customized the existing ruleset of Rhapsody in Ada in order to produce complete SPARK source code and associated SPARK Examiner commands from the model.

Rather than detailing the annotation generation capabilities that we obviously implemented, we would like to insist on the specific approach adopted for generating code from UML models. One of the usual benefits of code generators is that they auto-generate code for the user to represent the static aspects of a class such as attributes and relations accessors, and the dynamic aspects such as activity and statemachine diagrams implementations. We disabled it all for SPARK.

The main motivation for that is to keep the information flow to a minimum. This guarantees that the code fully results from conscious design decisions from the user.

4.4 Interfacing with the Examiner

In order to optimize the user workflow, We interfaced Rhapsody with Praxis SPARK Examiner so that it can be invoked directly from Rhapsody using the generated Examiner commands.

Besides, the Examiner output gets redirected back to Rhapsody so as to provide navigation capabilities from the reported errors to the model elements they come from.

4.5 Future directions

As Rhapsody in Ada is evolving, the degree of control it provides on model elements declaration order is increasing. The associated Ada/SPARK code generator developed by Sodius is getting regularly updated to fully exploit these new possibilities.

In order to make the usage of the SPARK profile more visible to the user, the rendering of class diagrams will be updated to reflect which stereotypes are being applied to operations and attributes.

Adding support for the RavenSPARK profile is another direction to explore.

5. RELATION TO PREVIOUS WORK

In [6] an instrumentation of SPARK to UML mapping is briefly discussed. In this paper, we provide additional insight on how to bridge the gap between SPARK and UML (and why) and on the modeling requirements tied to the post-modeling exploitation of the design.

6. SUMMARY AND CONCLUSION

By using "native" UML and a dedicated SPARK profile, it is possible to model SPARK systems. However, some evolutions of the UML are needed to integrate requirements derived from information-flow analysis. SPARK covering issues ranging from system business logic to implementation run-time safety, it logically follows that the aforementioned changes are distributed between the different levels of abstraction coverable with UML.

To be any useful, such a mapping has to be adequately tooled. This means the associated CASE software shall provide features in areas such as user-interface, methodological support, generators and if necessary extend the UML support to include SPARK specific modeling needs.

We produced such a tooling and plan on extending it even more.

This work is responding to a growing need from the market. It also illustrates how the high integrity aspects of systems on which SPARK focus should now be an essential part of UML models. Indeed, the technological advances in the field of model engineering requalify what yesterday were sometimes abusively termed "implementation details" (for lack of a way to exploit them before the actual implementation) into a valuable modeling flow.

7. ACKNOWLEDGEMENTS

We wish to thank the Praxis High Integrity Systems and the I-Logix people for the valuable help they provided us in adding SPARK support to Rhapsody in Ada.

8. REFERENCES

[1] Barnes, J., High Integrity Software, The SPARK Approach to Safety and Security, Addison Wesley, April 2003

[2] OMG, Unified Modeling Language Specification, Version 1.5, OMG Document formal /2003-03-01

[3] Amey, P., Correctness by Construction : Better Can Also Be Cheaper, CrossTalk Journal, March 2002.

[4] ECMA TR/55 Reference Model For Frameworks Of Software Engineering Environments. 3rd Edition, June 1993

[5] Amey, P, The INFORMED Design Method for SPARK. Praxis Critical Systems 1999, 2003.

[6] Amey, P., White, N., High-Integrity Ada in a UML and C World. Lecture Notes in Computer Science 3063. A Llamosi and A.Strohmeier (Eds.) : Reliable Software Technologies – Ada-Europe 2004 9th Ada-Europe International Conference, Palma de Mallorca, Spain, June 2004, 225-236