Safe and Secure Software

An Invitation to **Ada 2005**

11

Certified Safe with SPARK

Courtesy of AdaCore
The GNAT Pro Company

John Barnes

For some applications, especially those that are safety-critical or security-critical, it is essential that the program be correct, and that correctness be established rigorously through some formal procedure. For the most severe safety-critical applications the consequence of an error can be loss of life or damage to the environment. Similarly, for the most severe security-critical applications the consequence of an error may be equally catastrophic such as loss of national security, commercial reputation or just plain theft.

Applications are graded into different levels according to the risk. For avionics applications the DO-178B standard [1] defines the following

level E none: no problem; e.g. entertainment system fails? – could be a benefit!

level D minor: some inconvenience; e.g. automatic lavatory system fails.

level C major: some injuries; e.g. bumpy landing, cuts and bruises.

level B hazardous: some dead; e.g. nasty landing with fire.

level A catastrophic: aircraft crashes, all dead; e.g. control system fails.

As an aside, note that although a failure of the entertainment system in general is level E, if the failure is such that the pilot is unable to switch it off (perhaps in order to announce something unpleasant) then that failure is at level D.

For the most demanding applications, which require certification by an appropriate authority, it is not enough for a program to be correct. The program also has to be shown to be correct and that is much more difficult.

This chapter gives a very brief introduction to SPARK. This is a language based on a subset of Ada which was specifically designed for the writing of high integrity systems. Although technically just a subset of Ada with additional information provided through Ada comments, it is helpful to consider SPARK as a language in its own right which, for convenience, uses a standard Ada compiler, but which is amenable to a more formal treatment than the full Ada language. Analysis of a SPARK program is carried out by a suite of tools of which the most important are the Examiner, Simplifier, and Proof Checker.

We start by considering the important concept of correctness and contracts.

Contracts

What do we mean by correct software? Perhaps a general definition is: software that does what the user had in mind. And "had in mind" might literally mean just that for a simple one-off program written to do an ad-hoc calculation; for a large avionics application, it might mean the text of some written contract between the ultimate client and the software developer.

This idea of a software contract is not new. If we look at the programming libraries developed in the early 1960s, particularly in mathematical areas and

perhaps written in Algol 60 (a language favored for the publication of such material in respected journals such as the Communications of the ACM and the Computer Journal), we find that the manuals tell us what parameters are required, what constraints apply on their range and so on. In essence there is a contract between the writer of the subroutine and the user. The user promises to hand over suitable parameters and the subroutine promises to produce the correct answer.

The decomposition of a program into various component parts is very familiar and the essence of the programming process is to define what these parts do and therefore what the interfaces are between them. This enables the parts to be developed independently of each other. If we write each part correctly (so that it satisfies its side of the contract implied by its interface) and if we have defined the interfaces correctly, then we are assured that when we put the parts together to create the complete system, it will work correctly.

Bitter experience shows that life is not quite like that. Two things go wrong: on the one hand the interface definitions are not usually complete (there are holes in the contracts) and on the other hand, the individual components are not correct or are used incorrectly (the contracts are violated). And of course the contracts might not say what we meant to say anyway.

Correctness by construction

SPARK encourages the development of programs in an orderly manner with the aim that the program should be correct by virtue of the techniques used in its construction. This "correctness by construction" approach is in marked contrast to other approaches that aim to generate as much code as quickly as possible in order to have something to demonstrate.

There is strong evidence from a number of years of use of SPARK in application areas such as avionics, banking, and railway signaling that indeed, not only is the program more likely to be correct, but the overall cost of development is actually less in total after all the testing and integration phases are taken into account.

We will now look in a little more detail at the two problem areas introduced above, first giving complete interface definitions, and secondly ensuring that the code correctly implements the interface.

Ideally, the definition of the interfaces between the software components should hide all irrelevant detail but expose all relevant detail. Alternatively we might say that an interface definition should be both complete and correct.

As a simple example of an interface definition consider the interface to a subprogram. As just mentioned, the interface should describe the full contract

between the user and the implementer. The details of how the subprogram is implemented should not concern us. In order that these two concerns be clearly distinguished it is helpful to use a programming language in which they are lexically distinct. Some languages present subprograms (functions or methods) as one lump, with the interface physically bound to the implementation. This is a nuisance: not only does it make checking the interface less straightforward since the compiler wants the whole code, but it also encourages the developer to hack the code at the same time as writing the interface and this confuses the logic of the development process.

Ada has a structure separating interface (the specification) from the implementation (the body). This applies both to individual subprograms and to groups of entities encapsulated into packages and this is a key reason why Ada forms such a good base for SPARK.

SPARK requires additional information to be provided and this is done through the mechanism of annotations which conveniently take the form of Ada comments. A key purpose of these annotations is to increase the amount of information about the interface without providing unnecessary information about the implementation. In fact SPARK allows the information to be added at various levels of detail as appropriate to the needs of the application.

Consider the information given by the following Ada specification

```
procedure Add(X: in Integer);
```

Frankly, it tells us very little. It just says that there is a procedure called Add and that it takes a single parameter of type Integer whose formal name is X. This is enough to enable the compiler to generate code to call the procedure. But it says nothing about what the procedure does. It might do anything at all. It certainly doesn't have to add anything nor does it have to use the value of X. It could for example subtract two unrelated global variables and print the result to some file. But now consider what happens when we add the lowest level of annotation. The specification might become

```
procedure Add(X: in Integer);
--# global in out Total;
```

This states that the only global variable that the procedure can access is that called Total. Moreover the mode information tells us that the initial value of Total must be used (**in**) and that a new value will be produced (**out**). The SPARK rules also say more about the parameter X. Although in Ada a parameter need not be used at all, nevertheless an **in** parameter must be used in SPARK.

So now we know rather a lot. We know that a call of Add will produce a new value of Total and that it will use the initial value of Total and the value of X. We also know that Add cannot affect anything else. It certainly cannot print anything or have any other unspecified side effect.

Of course, the information regarding the interface is not complete since nowhere does it require that addition be performed in order to obtain the new value of Total. In order to do this we can add optional annotations which concern proof and obtain

```
procedure Add(X: in Integer);
--# global in out Total;
--# post Total = Total~ + X;
```

The annotation commencing **post** is called a postcondition and explicitly says that the final value of Total is the result of adding its initial value (distinguished by ~) to that of X. So now the specification is complete.

It is also possible to provide preconditions. Thus we might require X to be positive and we could express this by

```
--# pre X > 0;
```

An important aspect of the annotations is that they are all checked statically by the SPARK Examiner and other tools and not when the program executes.

It is especially important to note that the pre- and postconditions are checked before the program executes. If they were only checked when the program executes then it would be a bit like bolting the door after the horse has bolted (which reveals a nasty pun caused by overloading in English!). We don't really want to be told that the conditions are violated as the program runs. For example, we might have a precondition for landing an aircraft

```
procedure Touchdown( ... );
--# pre Undercarriage_Down;
```

It is pretty unhelpful to be told that the undercarriage is not down as the plane lands; we really want to be assured that the program has been analysed to show that the situation will not arise.

This thought leads into the other problem with programming – ensuring that the implementation correctly implements the interface contract. This is often called debugging. Generally there are four ways in which bugs are found

(1) By the compiler. These are usually easy to fix because the compiler tells us exactly what is wrong.

(2) At runtime by a language check. This applies in languages which carry out checks that, for example, ensure that we do not write outside an array. Typically we obtain an error message saying what structure was violated and whereabouts in the program this happened.

(3) By testing. This means running various examples and poring over the (un)expected results and wondering where it all went wrong.

(4) By the program crashing. This often destroys much of the evidence as well so can be very tedious.

Type 1 should really be extended to mean "before the program is executed". Thus it includes program walkthroughs and similar review techniques and it includes the use of analysis tools such as those provided for SPARK.

Clearly these four ways represent a progression of difficulty. Errors are easier to locate and correct if they are detected early. Good programming tools are those which move bugs from one category to a lower numbered category. Thus good programming languages are those which provide facilities enabling one to protect oneself against errors that are hard to find. Ada is a particularly good programming language because of its strong typing and runtime checks. For example, the correct use of enumeration types makes hard bugs of type 3 into easy bugs of type 1 as we saw in the chapter on Safe Typing.

A major goal of SPARK is to strengthen interface definitions (the contracts) and so to move all errors to a low category and ideally to type 1 so that they are all found before the program executes. Thus the global annotations do this because they prevent us writing a program that accidentally changes the wrong global variables. Similarly, detecting the violation of pre- and postconditions results in a type 1 error. However, in order to check that such violation cannot happen requires mathematical proof; this is not always straightforward but the SPARK tools automate much of the proof process.

The kernel language

Ada is a very comprehensive language and the use of some features makes total program analysis difficult. Accordingly, the subset of Ada supported by SPARK omits certain features. These mostly concern dynamic behavior. For example, there are no access types, no dynamic dispatching, generally no exceptions, all storage is static and hence all arrays must have static bounds (but subprogram parameters can be dynamic) and there is no recursion.

Tasking of course is very dynamic and although SPARK does not support full Ada tasking it does support the Ravenscar profile mentioned in the chapter on Safe Concurrency.

Another restriction that helps analysis is that every entity has to have a name. And each name should uniquely identify one entity. Hence all types and subtypes have to be named and overloading is generally prohibited. But the traditional block structure is supported so that local names are not restricted. Moreover, tagged types are permitted, although class wide types are not.

The idea of *state* is crucial to analysis and there is a strong distinction between procedures whose purpose is to change state and functions whose

purpose is simply to observe state. This echoes the difference between statements and expressions mentioned in the chapter on Safe Syntax. Functions in SPARK are not permitted to have any side effects at all.

The resulting kernel has proved to be sufficiently expressive for the needs of critical applications which would not want to use features such as dynamic storage.

Tool support

There are three main SPARK tools, the Examiner, the Simplifier and the Proof Checker.

The Examiner is vital. It has two basic functions

- It checks conformance of the code to the rules of the kernel language.
- It checks consistency between the code and the embedded annotations by flow analysis.

The Examiner performs these checks largely by analyzing the interfaces between components and ensuring that the details on either side do indeed conform to the specifications of the interfaces. The interfaces are of course the specifications of packages and subprograms and the annotations say more about these interfaces and thereby improve the quality of the contract between the implementation of the component and its users.

Incidentally, the Examiner is itself written in SPARK and has been applied to itself. There is therefore considerable confidence in the correctness of the Examiner.

The core annotations ensure that a program cannot have certain errors related to the flow of information. Thus the Examiner detects the use of uninitialized variables and the overwriting of values before they are used. This means that care should be taken not to give junk initial values to variables "just in case" as mentioned in the chapter on Safe Startup because that would hinder the detection of flow errors.

However, the core annotations do not address the issue of dynamic behavior. In order to do this a number of proof annotations can be inserted such as the pre- and postconditions we saw earlier which enable dynamic behavior to be analysed prior to execution. The general idea is that these annotations enable the Examiner to generate conjectures (potential theorems) which then have to be proved in order to verify that the program is correct with respect to the annotations. These proof annotations address

- pre- and postconditions of subprograms,
- assertions such as loop invariants and type assertions,

- declarations of proof functions and proof types.

The generated conjectures are known as verification conditions. These can then be verified by human reasoning, which is usually tedious and unreliable, or by using other tools such as the Simplifier and the Proof Checker.

Even without proof annotations, the Examiner can generate conjectures corresponding to the runtime checks of Ada such as range checks. As we saw in the chapter on Safe Typing, these are checks automatically inserted to ensure that a variable is not assigned a value outside the range permitted by its declaration or that no attempt is made to read or write outside the bounds of an array. The proof of these conjectures shows that the checks would not be violated and therefore that the program is free of runtime errors that would raise exceptions.

Note that the use of proof is not necessary. SPARK and its tools can be used at various levels. For some applications it might be appropriate just to apply the core annotations because these alone enable flow analysis to be performed. But for other applications it might be cost-effective to use the proof annotations as well. Indeed, different levels of analysis can be applied to different parts of a complete program.

There are a number of advantages in using a distinct tool such as the Examiner rather than simply a front-end processor which then passes its output to a compiler. One general advantage is that it encourages the early use of a V & V (Verification and Validation) approach. Thus it is possible to write pieces of SPARK complete with annotations and to have them processed by the Examiner even before they can be compiled. For example, a package specification can be examined even though its private part might not yet be written; such an incomplete package specification cannot of course be compiled.

There is a temptation to take an existing piece of Ada code and then to add the annotations (often referred to as "Sparking the Ada"). This is to be discouraged because it typically leads to extensive annotations indicative of an unnecessarily complex structure. Although in principle it might then be possible to rearrange the code to reduce the complexity, it is often the case that such good intentions are overridden by the desire to preserve as much as possible of the existing code.

The proper approach is to treat the annotations as part of the design process and to use them to assist in arriving at a design which minimizes complexity before the effort of detailed coding takes one down an irreversible path.

Examples

As a simple example here is a version of the stack with full core annotations (but not proof annotations)

```
package Stacks is

  type Stack is private;

  function Is_Empty(S: Stack) return Boolean;
  function Is_Full(S: Stack) return Boolean;

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

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

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

private
  Max: constant := 100;
  type Top_Range is range 0 .. Max;
  subtype Index_Range is Top_Range range 1 .. Max;
  type Vector is array Index_Range of Float;
  type Stack is
    record
      A: Vector;
      Top: Top_Range;
    end record;
end Stacks;
```

We have added functions Is Full and Is Empty which just read the state of the stack. They have no annotations at all.

Derives annotations have been added to the various procedure specifications; these are not mandatory but can improve flow analysis. Their purpose is to say which outputs depend upon which inputs – in this simple example they can in fact be deduced from the parameter modes. However, redundancy is one key to reliability and if they are inconsistent with the modes then that will be detected by the Examiner and perhaps thereby reveal an error in the specification.

The declarations in the private part have been changed to give names to all the subtypes involved.

At this level there are no changes to the package body at all – no annotations are required. This emphasizes that SPARK is largely about improving the quality of the description of the interfaces.

A difference from the earlier examples is that we have not given an initial value of 0 for Top but require that Clear be called first. When the Examiner looks at the client code it will perform flow analysis to ensure that Push and Pop are not called until Clear has been called; this analysis will be performed without executing the program. If the Examiner cannot deduce this then it will report that the program has a potential flow error. On the other hand if it can actually deduce that Push or Pop are called before Clear then it will report that the program is definitely in error.

In this brief overview it is not feasible to give serious examples of the proof process but the following trivial example will illustrate the ideas. Consider

```
procedure Exchange(X, Y: in out Float);
--# derives X from Y &
--#             Y from X;
--# post X = Y~ and Y = X~;
```

which shows the specification of a procedure whose purpose is to interchange the values of the two parameters. The body might be

```
procedure Exchange(X, Y: in out Float) is
  T: Float;
begin
  T := X;  X := Y;  Y := T;
end Exchange;
```

Analysis by the Examiner generates a verification condition which has to be shown to the true. In this particular example this is trivial and is done automatically by the Simplifier. In more elaborate situations the Simplifier will not be able to complete a proof in which case the Proof Checker is then used. This is an interactive program which, under human guidance, will hopefully be able to find a valid proof.

Certification

As earlier chapters have shown, Ada is an excellent language for writing reliable software. Ada allows programmers to catch errors early in the development process. Even more errors can be detected by using SPARK without having to rely on testing – a difficult and error-prone process in itself, yet an indispensable part of the software process.

For the highest level of safety-critical and security-critical applications it is not enough for a program to be correct. It also has to be shown to be correct. This is usually called certification and is performed according to the methods of a relevant certification agency. Examples of such agencies in the US are the FAA for safety-critical applications and the NSA for security-critical

applications. SPARK is of great value in developing programs to be certified as safe or secure as appropriate.

It might be thought that using SPARK adds to development costs. However, a recent study concerning a security system for the NSA [5] showed that using SPARK proved cheaper than conventional development methods. This again is perhaps surprising because SPARK clearly requires effort for the writing of annotations. But again that effort is well spent and reduces time needed for correcting errors. In the particular application concerned it is claimed that no errors were ever introduced anyway because of the careful way in which the program was constructed.

North American Headquarters
104 Fifth Avenue, 15th floor
New York, NY 10011-6901, USA
tel +1 212 620 7300
fax +1 212 807 0162
sales@adacore.com
www.adacore.com

European Headquarters
46 rue d'Amsterdam
75009 Paris, France
tel +33 1 49 70 67 16
fax +33 1 49 70 05 52
sales@adacore.com
www.adacore.com