FORMAL VALIDATION OF AEROSPACE SOFTWARE

David LESENS [(1)], Yannick MOY [(2)], Johannes KANIG [(2)]

[(1)] Astrium Space Transportation, 51-61 route de Verneuil 78130 Les Mureaux France, david.lesens@astrium.eads.net
[(2)] AdaCore, 46 rue d'Amsterdam 75009 Paris France, <name>@adacore.com

ABSTRACT

Any single error in critical software can have catastrophic consequences. Even though failures are usually not advertised, some software bugs have become famous, such as the error in the MIM-104 Patriot. For space systems, experience shows that software errors are a serious concern: more than half of all satellite failures from 2000 to 2003 involved software.
To address this concern, this paper addresses the use of formal verification of software developed in Ada.

1. INTRODUCTION

Software validation activities mandated for critical software are essential to achieve the required level of confidence. They are becoming increasingly difficult and costly as, over time, they require the development and maintenance of a large body of functional and robustness tests on larger and more complex applications. Testing and code review, the most widely deployed techniques for software validation, suffer from severe shortcomings. Indeed, both methods are very time consuming and labour intensive processes. For most critical systems, testing represents more than half of the total development costs. And, despite this high cost, it is impossible to discover all bugs with testing. In short, as E. W. Dijsktra puts it: *"Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence."*

Formal program validation offers a way to reduce these costs while providing stronger guarantees than testing. Addressing validation activities with formal validation is supported by upcoming standards such as DO-178C for software development in avionics. The Hi-Lite project has pursued the integration of formal validation with testing for projects developed in Ada: formal validation can be applied independently to subprograms that fall in the SPARK subset of Ada, while testing can be applied to all other subprograms. AdaCore and Astrium Space Transportation have been working together since 2010 to define the subset of Ada that can be analyzed formally, and extensions to Ada that facilitate specification and validation. AdaCore developed tools for static and dynamic validation that programs implement their specification. Astrium has applied these tools to various case studies from the space domain.

This paper provides the details of the chosen formal validation approach and its application results.

2. ADA PROGRAMS FORMAL VALIDATION

SPARK [2] is a subset of Ada augmented with special annotations (in stylized comments) to specify the expected functional behaviour of the program. Among these annotations, "Subprogram contracts" are the most important, specifying both in which context a subprogram is called (its precondition) and the expected outcome of the subprogram (its postcondition). These specifications can be either partial or total. A set of tools allows the formal validation that a program implements its specification, and that it cannot raise a run-time error when executed (no uninitialized read, no out-of-bounds array access, zero divide, etc.). These validations are done independently for each subprogram, by using the contract (both precondition and postcondition) of a subprogram at each calling points in the main program.

In the Hi-Lite project, we have redefined a new version of the SPARK language, where all annotations are now taken from Ada itself. This was made possible because the new version of Ada issued in 2012 includes specification features, such as preconditions and postconditions, as well as a richer expression language to use in contracts.

In the example below, one can specify that a function *Find* only applies to pairs of a *Table* and a *Value* where the *Value* appears in the *Table* (see the precondition below), and that it then returns the first such index (see the postcondition below). This contract uses the new quantified expressions that allow expressing the usual mathematical quantifications over a finite range: existential quantification introduced by *for some* and universal quantification introduced by *for all*. The result

of the function is designated by *Find'Result* in the postcondition.

```
function Find (Table : MyArray; Value : T1) return T2
  with
    Pre  => (for some Index in Table'Range =>
       Table (Index) = Value),
    Post => Table (Find'Result) = Value  and then
       (for all Index in Table'First .. Find'Result - 1 =>
          Table (Index) /= Value);
```

Ada defines other language aspects, pragmas and attributes to facilitate the expression of specifications. The Ada Reference Manual precisely defines the meaning of these features in terms of execution. For example, a *for all* quantification executes as a loop, and returns the result *True* if the expression evaluated is *True* for all values in the range, or else it returns the result *False* as soon as one evaluation of the expression is *False*. The precondition (resp. the postcondition) evaluates as an assertion that raises an error at run time if the expression evaluates to *False* on subprogram entry (resp. subprogram exit).

The new version of SPARK comprises most of Ada, only excluding features that make it notably more difficult to specify programs, or to prove them automatically. The most notable restrictions are:

- The use of access types and allocators is not permitted.
- All expressions (including function calls) are free of side-effects.
- Aliasing of names is not permitted.
- The goto statement is not permitted.
- The use of controlled types is not permitted.
- Tasking is not currently permitted (it is intended that this will be included in a future version of the SPARK language).
- Raising and handling of exceptions is not permitted (exceptions can be included in a program but proof must be used to show that they cannot be raised; these restrictions may also be relaxed in a future version of the language).

We have defined additional language constructs (aspects, pragmas and attributes) in SPARK to further facilitate formal specification and validation, in particular for:

- Specification of subprogram data dependences (the *globals* annotation in SPARK 2005). The following example declaration specifies that the procedure reads and writes the global array *Table*:

```
procedure Update_Index (I : Index)
  with Global => (In_Out => Table);
```

- Specification of subprogram data flows (the *derives* annotation in SPARK 2005). The following example declaration specifies that the new value of parameter X depends on the value of Y only:

```
procedure P (X : in out Integer; Y, Z : Integer)
  with Depends => (X => Y);
```

- specification of contracts by disjoint cases
- proof of subprogram with loops (loop invariants)
- proof of loop termination (loop variants)

Constructs for specifying the functional behaviour of a program are defined in terms of execution. For example, the failure to respect a loop invariant or variant leads to a run-time exception during execution. Constructs for specifying data dependences and flows are defined in terms of validation only.

The tool GNATprove [3] developed in the context of project Hi-Lite aims at providing for this new version of SPARK the same functionality provided by the validation tools for SPARK 2005. The main differences between the two versions of the technology are that:

- GNATprove interprets annotations (like preconditions and postconditions) in exactly the same way as they are interpreted during execution. In particular, GNATprove needs to prove that expressions cannot raise run-time errors when evaluated.
- GNATprove can be applied on units that do not fall completely in the SPARK subset. In that case, it ignores the part of the unit that are not SPARK compliant.

The fact that specifications have the same meaning in proofs and during execution is very useful for debugging specifications: a run-time failure during testing might reveal that a precondition is wrong, and then classical debugging can be used to understand the failure. This perfect match between dynamic and static interpretation of specifications is also the basis for the

combination of formal validation and testing. This allows discharging by testing the assumptions made during formal validation, when a program is only partially proved. These assumptions may be both contracts written by the user (preconditions and postconditions) that need to be exercised during testing, or implicit contracts added by the proof tools concerning initialization of subprogram inputs/outputs and non-aliasing properties. Under special switches, the GNAT compiler inserts the corresponding checks for these implicit contracts, so that they can too be verified dynamically during testing.

3. AEROSPACE CASE STUDY

3.1 Objectives of the case study

In 2011, the "Full Model Driven Development for On-Board Software" project (co-funded by ESA, Astrium Space Transportation, Esterel Technologies, IRIT, Altran Praxis and Verimag) has experimented the use of formal methods for the development of space software. The automatic generation of Ada code and the use of SPARK 2005 were specifically analysed on a case study developed by Astrium Space Transportation with the following results:

- A certifiable automated code generator from SCADE Suite models to SPARK 2005 code has been developed and is now commercialized by Esterel Technologies.
- The ability to develop highly critical software in SPARK 2005 was assessed. On the one hand, it allowed Astrium Space Transportation to efficiently develop a case study which was exhaustively proved to be free of run-time errors. On the other hand, the following drawbacks were identified:
    - The restrictions imposed by SPARK 2005 lead to non negligible overcost and restrict its scope of use.
    - Software engineers may accept only with great difficulties these constraints, even after an adequate training.
    - The formal proof activity needs to be performed independently from the classical test activity (due to the fact that the SPARK 2005 contracts are not executable).
    - The use of the interactive proof tool of the SPARK suite is highly complex and expensive.

The Hi-Lite approach has been assessed with the objectives to keep the benefits of SPARK 2005 (detection of potentially dangerous code and formal exhaustive proof) and at the same time to extend its scope and to facilitate its use.

3.2 Description of the case study

The case study developed by Astrium Space Transportation implements a prototype of a generic OBCP (On-Board Control Procedure) engine following the principles specified in the ECSS-E-ST-70-01C ("Spacecraft on-board control procedures" – 16 April 2010). This standard defines the general principles of a Mission and Vehicle Management functionality. An on-board control procedure is in practice represented by a simplified programming language interpreted onboard the spacecraft. This interpreter is generally at the highest level of criticality of the spacecraft. Our implementation of this interpreter in SPARK is table driven and relies greatly on Ada generic programming.

The OBCP language contains the following features:

- Mathematical expressions
- Events detection with a notion of timed window
- Automated procedures to implement change of modes
- OBCP procedures with complex control (if-then-else, jump, loop, sub-procedures, etc.)

The developers of the case study have taken care to:

- Use only strict variable types approach: For each variable, a range of accepted value has been defined.
- Avoid as far as possible any of the constructs forbidden by Hi-Lite, for instance pointers.
- Strictly define for every single subprogram a set of test cases and a formal contract (see section 2).
- Use advanced constructs of Ada 2012 in order to assess the Hi-Lite scope:
    - Generic packages: This will allow making the OBCP engine generic enough to be used on a launcher such as Ariane 6 or a spacecraft such as the MPCV.
    - Object Oriented design: This feature also increases the genericity of the software, making its customization to a specific launcher or spacecraft easy and safe.
    - Expression functions offering a convenient way to express simple functions.
    - Conditional expressions providing a compact and more readable notation.

- Quantified expressions, used in particular in contracts.

The following subprogram is an example of expression function using a quantified expression:

```
function G (X : T_Record) return Boolean is
   (for all I in X.A'Range => X.A (I));
```

This implementation of a reusable Mission and Vehicle Management relies also greatly on

- Generic packages
- Discriminant

The generic packages allow an easy customization of the code:

```
generic type T_Event_Id is (<>);-- the list of events
package Mvm.Events is ...
```

The discriminants ensure a strict typing of the code, even in case of heterogeneous communication between components of the system:

```
type T_Monitoring is (No_Window, Time_Window,
  Protected_Window);

type T_Event_Status
  (Monitoring_Type : T_Monitoring := No_Window)
is record
    Detection_Time : T_Float32;
    case Type_Of_Monitoring is
    when No_Window =>null;
    when Time_Window Protected_Window =>
      Start_Window : T_Float32;
      End_Window : T_Float32;
    end case;
  end record;
```

3.3 Results of the formal proof activity

All the contracts have been checked by dynamic testing. This phase is quite classical, except for the fact that the testing includes the preconditions and the postconditions defined in the case study. Then, GNATprove has been applied.

This case study contains 10 parts which results are detailed in the following sections. For each part, the duration of the analysis is provided. Then, a first table shows the number of subprograms in SPARK, not in SPARK or partially in SPARK. A second table shows then the number of proved and non proved checks…

Results for "Time Management":

Analysis duration: 10 seconds (0 h 0 mn 10 s)

Table 1. Subprograms in SPARK

Subprograms fully in SPARK	3
Bodies not in SPARK	0
Specifications not in SPARK	0
Bodies not yet in SPARK	0
Bodies not yet in SPARK	0

Table 2. Results

Features	Nb checks	Proved
postcondition	2	100%
range_check	1	100%
Total	3	100%

Results for "Mathematical Library"

Analysis duration: 475 seconds (0 h 7 mn 55 s)

Table 3. Subprograms in SPARK

Subprograms fully in SPARK	83
Bodies not in SPARK	0
Specifications not in SPARK	0
Bodies not yet in SPARK	0
Bodies not yet in SPARK	0

Table 4. Results

Features	Nb checks	Proved
division_check	10	100%
overflow_check	46	100%
postcondition	26	92%
precondition	6	100%
range_check	49	95%
Total	137	97%

The non proved checks are due to the fact that some algoritmic functions are not completely known by GNATprove. It is for instance the case for trigonometric functions:

```
function Arctan (X : T_Float32) return T_Float32
with
  Post => (Arctan'Result >= -C_Halfpi32) and then
  (Arctan'Result <= C_Halfpi32);

function Arctan (X : T_Float32) return T_Float32
is (Num32.Arctan (X));
```

The postcondition is not proved by GNATprove. The exact behaviour of algorithmic functions depending of the implementation, this behaviour is acceptable.

Algorithmic functions and their contracts are preferably validated by intensive testing.

Results for “Single Variable”

Analysis duration: 118 seconds (0 h 1 mn 58 s)

Table 5. Subprograms in SPARK

Subprograms fully in SPARK	85
Bodies not in SPARK	0
Specifications not in SPARK	0
Bodies not yet in SPARK	0
Bodies not yet in SPARK	0

Table 6. Results

Features	Nb checks	Proved
discriminant_check	123	100%
Postcondition	30	100%
Precondition	115	100%
Total	268	100%

Results for “List Of Variables”

Analysis duration: 274 seconds (0 h 4 mn 34 s)

Table 7. Subprograms in SPARK

Subprograms fully in SPARK	140
Bodies not in SPARK	0
Specifications not in SPARK	0
Bodies not yet in SPARK	0
Bodies not yet in SPARK	0

Table 8. Results

Features	Nb checks	Proved
Assertion	85	100%
loop_invariant_initialization	2	100%
loop_invariant_preservation	2	100%
Postcondition	31	100%
Precondition	132	100%
Total	252	100%

Results for “Events”

Analysis duration: 371 seconds (0 h 6 mn 11 s)

Table 9. Subprograms in SPARK

Subprograms fully in SPARK	24
Bodies not in SPARK	0
Specifications not in SPARK	0
Bodies not yet in SPARK	0
Bodies not yet in SPARK	0

Table 10. Results

Features	Nb checks	Proved
Assertion	27	100%
discriminant_check	104	100%
loop_invariant_initialization	1	100%
loop_invariant_preservation	1	100%
overflow_check	5	100%
Postcondition	17	100%
Precondition	57	100%
range_check	1	100%
Total	213	100%

Results for “Expressions”

Analysis duration: 1992 seconds (0 h 33 mn 12 s)

Table 11. Subprograms in SPARK

Subprograms fully in SPARK	331
Bodies not in SPARK	0
Specifications not in SPARK	0
Bodies not yet in SPARK	0
Bodies not yet in SPARK	0

Table 12. Results

Features	Nb checks	Proved
Assertion	385	100%
discriminant_check	767	100%
loop_invariant_initialization	2	100%
loop_invariant_preservation	2	100%
overflow_check	2	100%
Postcondition	97	100%
Precondition	413	100%
range_check	2	100%
Total	1670	100%

Results for “Parameters”

Analysis duration: 279 seconds (0 h 4 mn 39 s)

Table 13. Subprograms in SPARK

Subprograms fully in SPARK	62
Bodies not in SPARK	0
Specifications not in SPARK	0
Bodies not yet in SPARK	0
Bodies not yet in SPARK	0

Table 14. Results

Features	Nb checks	Proved
Assertion	2	100%
discriminant_check	11	100%
index_check	6	66%
loop_invariant_initialization	1	100%
loop_invariant_preservation	1	100%
Postcondition	2	100%
Precondition	1	100%
range_check	7	100%
Total	31	93%

GNATprove is not yet able to verify the index of an array which dimension is defined by a type discriminant

```
subtype R is Integer range 1 .. 100;
type T_Array is array (R range <>) of Boolean;
type T_Record (L : R) is record
    A : T_Array (1 .. L);
  end record;

function G (X : T_Record) return Boolean is
  (for all I in X.A'Range => X.A (I));
```

In function *G*, the index check *X.A (I)* is not proved even if *I* is defined in the range of *X.A*. An improvement of GNATprove is in progress in order to deal with such case.

Results for "Functional Unit"

Analysis duration: 2921 seconds (0 h 48 mn 41 s)

Table 15. Subprograms in SPARK

Subprograms fully in SPARK	76
Bodies not in SPARK	0
Specifications not in SPARK	0
Bodies not yet in SPARK	13
Bodies not yet in SPARK	13

The origins of subprograms not yet in SPARK are the following:

- class wide types (5 subprograms)
- tagged type (17 subprograms)

These origins are related to Object Oriented Programming. The analysis of Object Oriented software is foreseen but has not yet been implemented.

Table 16. Results

Features	Nb checks	Proved
Assertion	2	100%
discriminant_check	58	100%
index_check	26	15%
Loop_invariant_initialization	1	100%
Loop_invariant_preservation	1	100%
Postcondition	3	66%
Precondition	1	100%
range_check	14	71%
Total	106	74%

Most of the non proved VCs are due to index of an array which dimension is defined by a type discriminant (see "Parameters" section). The proof of the postconditions is not possible before the other proofs.

Results for "Automated Procedure"

Analysis duration: 7803 seconds (2 h 10 mn 3 s)

Table 17. Subprograms in SPARK

Subprograms fully in SPARK	192
Bodies not in SPARK	28
Specifications not in SPARK	15
Bodies not yet in SPARK	3
Bodies not yet in SPARK	3

The origins of subprograms not in SPARK are the use of accesses. Accesses are used in the case study to store objects in a table. This kind of design can not be proved by GNATprove.

The origins of subprograms not yet in SPARK are the following:

- class wide types (20 subprograms)
- tagged type (22 subprograms)

As before, this is due to Object Oriented programming.

Table 18. Results

Features	Nb checks	Proved
Assertion	6	50%
discriminant_check	158	99%
index_check	68	25%
loop_invariant_initialization	2	50%
loop_invariant_preservation	2	100%
Postcondition	7	57%
Precondition	13	92%
range_check	28	53%
Total	284	74%

A part of the non proved VCs are due to index of an array which dimension is defined by a type discriminant (see "Parameters" section). The remaining non proved VCs are due to a too complex subprogram. This subprogram shall be split in several smaller subprograms to be proved.

Results for "On Board Control Procedure"

Analysis duration: 13705 seconds (3 h 48 mn 25 s)

Table 19. Subprograms in SPARK

Subprograms fully in SPARK	547
Bodies not in SPARK	447
Specifications not in SPARK	30
Bodies not yet in SPARK	13
Bodies not yet in SPARK	13

The origins of subprograms not in SPARK are the following:

- access (61 subprograms)

- unchecked conversion (377 subprograms)

Accesses are used with Object (see "Automated Procedure" section). The unchecked conversions are used in a library allowing reading external inputs. All the concerned subprograms are very small and shall be validated by intensive testing because there are out of the perimeter of HiLite and of SPARK.

The origins of subprograms not yet in SPARK are the following:

- attribute (10 subprograms)
- class wide types (23 subprograms)
- tagged type (25 subprograms)

As before, this is due to Object Oriented programming.

Table 20. Results

Features	Nb checks	Proved
Assertion	418	99%
discriminant_check	1113	99%
index_check	82	23%
loop_invariant_initialization	5	80%
loop_invariant_preservation	5	100%
overflow_check	7	100%
Postcondition	148	87%
Precondition	637	98%
range_check	39	61%
Total	2454	95%

A part of the non proved VCs are due to index of an array which dimension is defined by a type discriminant (see "Parameters" section). The remaining non proved VCs are due to too complex subprograms. These subprograms shall be split in several smaller subprograms to be proved.

4. CONCLUSION

Despite the important results achieved by academic researchers in the last decade, formal proof techniques did not, up to now, break through in the software industry in general and the space domain in particular. The reasons for this failure were numerous: techniques not natively supported by the programming languages (necessity to define contracts in specific comments), contracts not executable, tools not powerful enough or too hard to be used by non expert.

The Hi-Lite project has suppressed these current limitations of formal proof. The space community has to take benefit from these advances by using these newly available tools and by adapting accordingly the ECSS.

5. BIBLIOGRAPHY

[1] Formal Model Driven Engineering for Space Onboard Software - ERTS 2012 - Eric Conquet, François-Xavier Dormoy, Iulia Dragomir, Susanne Graf, David Lesens, Piotr Nienaltowski, Iulian Ober

[2] SPARK - The Proven Approach to High Integrity Software - 2012 - John Barnes with Altran Praxis

[3] Integrating Formal Program Validation with Testing - ERTS 2012 - Cyrille Comar, Johannes Kanig and Yannick Moy