Verifying High-Assurance FACE™ Components with Ada and SPARK

Combining Formal Methods and Testing

Air Force FACE™ TIM Paper by:

Benjamin M. Brosgol
AdaCore

September, 2019

Table of Contents

Executive Summary

For high-assurance FACE™ Units of Conformance – e.g., components that need to be certified / qualified at Design Assurance Levels A or B of a standard such as DO-178C [1] for software on commercial aircraft or a similar standard for military airborne systems– verifying that the code correctly implements its requirements can be a daunting challenge. This paper shows how a developer can address this challenge through *hybrid verification,* a technique that combines both traditional testing and mathematics-based formal methods. Taking advantage of advances in proof technology and hardware speed and capacity, formal methods are practical for real-world systems and can be introduced incrementally into a project alongside testing.

This paper explains hybrid verification in a FACE context. It uses the Ada language [2] and its formally analyzable SPARK subset [3] to illustrate the concepts, based on the Ada 95 Safety capability set definitions in Edition 3.0 of the FACE Technical Standard [4]. Work is currently in progress on defining Ada 2012 Safety capability sets, and the same verification principles will apply. Critical modules in the application can be written in SPARK, meeting the restrictions imposed by either the Ada 95 Safety-Base & Security or Safety-Extended capability set, and properties such as absence of run-time errors can be verified formally by the SPARK toolset. Less critical modules can be written using Ada features outside the SPARK subset but still within the relevant Safety capability set, and then verified through testing. Appropriate checking is performed at the interfaces between tested and formally verified modules. By combining formal methods and testing, a FACE component developer can achieve higher confidence that the software meets its requirements – including safety and security assurance – than through testing alone. Experience has shown that formal methods can be introduced into an organization's verification infrastructure by stages, without requiring developers to have previous knowledge of the technology and without increasing life cycle costs.

The general approach of hybrid verification can be applied in other contexts, for example using the C language and the Frama-C [5] technology, but Ada and SPARK bring the benefits of easier analyzability through strong typing and other semantic checks. Hybrid verification also applies in mixed-language situations (e.g., formally verified SPARK combined with traditionally tested C).

The paper is oriented towards software developers or project managers; no previous knowledge of Ada is required

Introduction

Software targeted to an Operating System that conforms to a FACE Safety or Security profile often has to meet a domain-specific standard such as DO-178C for airborne software on commercial aircraft or a similar military standard. Certification under such a standard comprises a range of activities associated with the various software life cycle processes, and a major effort is devoted to verification: demonstrating, with a degree of confidence commensurate with the criticality of the system, that the software meets its requirements. Verification combines review, analysis and testing. The highest Design Assurance Levels, DAL A and B in DO-178C, demand a correspondingly significant amount of effort, with an emphasis on requirements-based testing. Testing, however, can never be complete.

This paper describes a technique known as *hybrid verification*, which combines traditional testing with mathematics-based formal methods. Through formal methods the developer can prove relevant properties of critical software, such as correctness of information flow, absence of run-time errors, and even full functional correctness. Advances in proof technology and hardware power / capacity have made formal methods a practical approach, and one which can be introduced incrementally into an existing project. DO-333 [6], the Formal Methods supplement to DO-178C, shows how formal methods can be applied to reduce the verification effort during certification, in some cases by replacing a testing activity. Even when certification is not required, formal methods provide the benefit of increased confidence that the software has the required properties.

The SPARK technology – a formally analyzable subset of Ada 2012 together with its supporting toolset – illustrates how hybrid verification can work in practice. Developers can employ formal methods to analyze critical modules written in SPARK while using traditional testing to achieve an appropriate level of confidence in less critical modules that use Ada features outside the SPARK subset. The key to enabling this approach is Ada's contract-based programming features and, in particular, the ability to provide pre- and postconditions for subprograms. This facility is supported directly in Ada 2012 with special syntax and can be modeled in Ada 95 via pragmas. Formally verified components can invoke subprograms that have been verified through testing, and in the other direction a tested component can invoke a subprogram that has been formally verified. The relevant pre- and postcondition checks are performed at module boundaries, either statically or via testing. Hybrid verification can also be used when mixing formally analyzed SPARK code with modules written in C, such as low-level functions that might be needed for Operating System services.

The SPARK Ada subset is an excellent match for the restrictions in both the Ada 95 Safety-Extended and Safety-Base & Security capability sets. This is not a surprise, as both SPARK and the Safety capability sets have the goal of restricting or excluding features that could be problematic (or are not needed) in safety-critical software. Work is in progress (in the FACE Technical Working Group's Operating System Segment Subcommittee) on defining analogous Safety capability sets for Ada 2012, compatible with the Ada 95 sets: i.e., a program obeying the Ada 95 Safety capability set restrictions will also obey the Ada 2012 restrictions. As a result, SPARK is likewise expected to be an excellent match for the Ada 2012 Safety capability sets. A legal SPARK program satisfies many of the restrictions in both Ada 95 Safety sets. Features allowed by SPARK, but prohibited from a Safety capability set, can be detected and prevented through a combination of the standard Ada pragma Restrictions coupled with an automated static analysis tool. The SPARK toolset can formally prove a variety of relevant properties based on the FACE component's assurance requirements. As a

result, the SPARK technology in general, and its support for hybrid verification in particular, are especially useful to developers of safety- or security-critical FACE components.

The SPARK Technology

The SPARK language

SPARK is an extensive subset of Ada 2012. It includes as much of the Ada language as is possible /practical to analyze formally, while eliminating sources of undefined and implementation-dependent behavior. SPARK includes Ada's program structure support (packages, generics, child libraries), most data types, contract-based programming (subprogram pre- and postconditions, scalar ranges, type/subtype predicates), Object-Oriented Programming, and the Ravenscar subset [2, Section D.13] of the tasking features.

Principal exclusions are side effects in functions and expressions, problematic aliasing of names, the *goto* statement, exception handling, and most tasking features. Access types (pointers) have traditionally been excluded from SPARK but support for a safe and formally analyzable subset of the Ada access type features is being added.

Figure 1 shows an example of SPARK code.

The "with" constructs, known as "aspects", here define the Decrement procedure's contracts:

- Global: the only access to non-local data is to read the value of N
- Depends: the value of X on return depends only on N and the value of X on entry

```
N : Positive := 100; -- N constrained to 1 .. Integer'Last

procedure Decrement (X : in out Integer)
   with Global => (Input =>N),
        Depends => (X => (X, N)),
        Pre     => X >= Integer'First + N,
        Post    => X = X'Old – N;

procedure Decrement (X : in out Integer) is
begin
   X := X-N;
end Decrement;
```

Figure 1: SPARK Example with Contracts

- Pre: a Boolean condition that the procedure assumes on entry
- Post: a Boolean condition that the subprogram guarantees on return

In this example the SPARK tool can verify the Global and Depends contracts and can also prove several dynamic properties: no run-time errors will occur during execution of the Decrement procedure, and, if the Pre contract is met when the procedure is invoked then the Post contract will be satisfied on return.

Although SPARK (and the SPARK proof tools) work with Ada 2012 syntax, a SPARK program can also be expressed in Ada 95, with contracts captured as pragmas. A usage scenario, if a FACE component developer is applying the SPARK technology in an otherwise all-Ada 95 context, would be to write the SPARK contracts as pragmas, limiting expressions to those permitted in Ada 95 (thus no quantification). SPARK's "ghost variable" facility – the use of temporary variables that are only used for proofs and do not exist at run time – is useful here.

SPARK and the Ada Safety capability sets

The Ada 95 Safety capability sets (and indeed the capability sets for the other programming languages supported by the FACE Technical Standard) are not intended as style-oriented coding rules, but they do prohibit features with complex run-time semantics and also restrict functionality that could be problematic in a safety-critical system. These goals are consistent with SPARK, and in fact a SPARK program will automatically comply with many of the restrictions imposed by the Ada 95 Safety capability sets. For example, the tasking features in SPARK are those that are allowed by the Ravenscar profile, which is also the tasking subset permitted in the Ada Safety capability sets. In those cases where SPARK permits a feature that is outside the Ada capability sets, the FACE component developer has several techniques to detect and eliminate the excluded feature:

- pragma Restrictions

  This standard Ada pragma allows the user to specify language features that the compiler will reject. The pragma can prohibit dependence on all of the run-time packages excluded from the Ada Safety capability sets (Ada.Wide_Text_IO, etc.) and also prohibit or restrict the usage of exceptions and allocators.

- Static analysis tool (code standard enforcer)

  The other restrictions in the Ada Safety capability sets are Ada semantic features that can be detected by an automated tool such as GNATcheck [7]. The FACE component developer can configure this rule-based and tailorable tool to flag Ada capability set violations such as usage of specific prohibited types or subprograms defined in otherwise-permitted packages.

For compliance with a certification standard such as DO-178C, the developer needs to define the language subset to which the application will be restricted, termed a "code standard". For a FACE component the code standard is the SPARK language as constrained by the additional Safety capability set restrictions. The Ada compiler and SPARK proof tools will enforce compliance with the SPARK language definition, and a combination of Ada's pragma Restrictions and an automated tool such as GNATcheck can enforce the additional FACE capability set restrictions.

The FACE Technical Standard, Edition 3.0, did not define Safety capability sets for Ada 2012, but, as noted above, work is in progress (a proposal has been drafted by the Operating Systems Subcommittee) to define such sets for inclusion in Edition 3.1, compatible with the Ada 95 Safety capability sets. As a result, the approach described in this section – use of pragma Restrictions and static analysis to detect violations – will be applicable to FACE components written in Ada 2012 once the new capability set definitions are incorporated in a subsequent edition of the FACE Technical Standard (possibly with additional restriction checks based on the new sets).

SPARK and software certification

The treatment of formal methods in the certification standards for airborne software has evolved in light of the benefits and continuing maturity of the underlying technology. In DO-178B formal methods were permitted as an "alternative method" which in principle could have been used to satisfy some of the verification objectives. However, there was not a common understanding of which activities formal methods could replace or ameliorate. In practice formal methods were therefore only considered as additional measures, augmenting the verification activities that DO-178B otherwise specified. As a result, formal methods tended to see very little usage for certification, at least as part of the certification evidence.

That situation is changing, and the DO-178C effort included the development of the DO-333 Formal Methods supplement to clarify the role that formal methods can take in verification. DO-333 defines a formal method as a "formal model combined with a formal analysis". A formal model needs to be precise and unambiguous and have a mathematically defined structure. Formal analysis applies to a formal model and needs to be *sound*: if it purports to detect whether a given property is true (e.g., that all variables are initialized before being read) then it must not report that the property is true when in fact there is an execution context where it is false.

DO-333 offers guidance (objectives, activities) on applying formal methods during various software life cycle processes. The SPARK language and toolset can be used to satisfy many of these objectives, most notably in the verification process. For example, since a subprogram's pre- and postconditions define the subprogram's low-level requirements, formal analysis showing that a subprogram's postcondition will be met if its precondition is also satisfied can replace low level requirements testing. As another example, proof of data initialization and absence of run-time errors can satisfy some of the source code accuracy and consistency objectives in DO-178C. However, absence of unintended functionality must still be shown, and additional activities – though at much less effort than requirements testing – need to be performed on the executable object code to demonstrate preservation of properties between the source code and the object code. A comprehensive discussion of how SPARK and formal analysis can be used in a DO-178C / DO-333 certification context may be found in [8].

Soundness is a necessary, but not sufficient condition for a formal analysis method to be practical. Other important usability considerations include precision (minimization of false positives), scalability (acceptable performance as a system's size increases), localizability (the ability to define specific code regions that are or are not subject to formal analysis), proof engine power, and user assistance (help, such as the furnishing of a counterexample, when a proof does not succeed). The SPARK toolset has been designed to meet these requirements. For example, it comprises multiple proof engines to maximize its analysis capabilities and can exploit multicore architectures for optimal performance.

Levels of Adoption of Formal Methods

Formal methods are not an "all or nothing" technique; it is possible and in fact advisable for an organization to introduce the methodology in a stepwise manner, with the ultimate level depending on the assurance requirements for the software. This approach is documented in [9], which details the levels of adoption, including the benefits and costs at each level, based on the practical experience of a major aerospace company in adopting formal methods incrementally; the development team did not have previous knowledge of formal methods. The levels are additive; all the checks at one level are also performed at the next higher level.

In the context of a FACE component, the analysis performed by the SPARK tools would need to be combined with verification of adherence to the Ada Safety capability set restrictions as explained above.

Stone level: Valid SPARK

As the first step, a project can implement as much of the code as is possible in the SPARK subset, run the SPARK analyzer on the codebase (or new code), and look at violations. For each violation, the developer can decide whether to convert the code to valid SPARK or exclude from analysis. The benefits include easier maintenance for the SPARK modules (no aliasing, no side effects in functions) and project experience with the basic usage of formal methods. The costs include the effort that may be required to convert the code to SPARK (especially if there is heavy use of pointers).

Bronze level: Initialization and correct data flow

This level entails performing flow analysis on the SPARK code to verify intended data usage. The benefits include assurance of no reads of uninitialized variables, no interference between parameters and global objects, no unintended access to global variables, and no race conditions on accesses to shared data. The costs include a conservative analysis of arrays (since indices may be computed at run time) and potential "false alarms" that need to be inspected.

Silver level: Absence of run-time errors

At the Silver level the SPARK proof tool performs flow analysis, locates all potential run-time checks (e.g., array indexing), and then attempts to prove that none will fail. If the proof succeeds, this brings all the benefits of the Bronze level plus the ability to safely compile the final executable without exception checks. Critical software should aim for this level. The cost is the additional effort needed to obtain provability. In some cases (if the programmer knows that an unprovable check will always succeed, for example because of hardware properties) it may be necessary to augment the code with pragmas to help the prover.

Gold level: Proof of key integrity properties

At the Gold level, the proof tool will verify properties such as maintenance of critical data invariants or safe transitions between program states. Subprogram pre- and postconditions and subtype predicates are especially useful here, as is "ghost" code that serves only for verification and is not part of the executable. A benefit is that the proofs can be used for safety case rationale, to replace certain kinds of testing. The cost is increased time for tool execution, and the possibility that some properties may be beyond the abilities of current provers.

Platinum level: Full functional correctness

At the Platinum level, the algorithmic code is proved to satisfy its formally specified functional requirements. This is still a challenge in practice for realistic programs but may be appropriate for small critical modules, especially for high-security systems.

Hybrid Verification

The typical scenario for hybrid verification is an in-progress project that is using traditional testing and that has high-assurance requirements that can best be met through formal methods. The new code will be in SPARK; and the adoption level depends on the experience of the project team (typically Stone at the start, then progressing to Bronze or Silver). The existing codebase may be in Ada or other languages. To maximize the precision of the SPARK analysis, the subprograms that the SPARK code will be invoking should have relevant pre- and postconditions expressing the subprograms' low-level requirements. If the non-SPARK code is not in Ada, then the pre- and postconditions should be included on the Ada subprogram specification corresponding to the imported function; see Figure 2 for an example.

```
function getascii return Interfaces.C.unsigned_char
with Post => getascii'Result in 0..127;
pragma Import (C, getascii);
-- Interfaces.C.unsigned_char is a modular (unsigned) integer type,
-- typically ranging from 0 through 255

procedure Example is
   N : Interfaces.C.unsigned_char range 0 .. 127;
begin
   N := getascii; -- SPARK can prove that no range check is needed

end Example;
```

Figure 2: SPARK Code Invoking a Tested C Function

The verification activity depends on whether the formally verified code invokes the tested code, or *vice versa*.

- The SPARK code calls a tested subprogram

  If the tested subprogram has a precondition, then at each call site the SPARK code is checked to see if the precondition is met. Any call that the proof tool cannot verify for compliance with the precondition needs to be inspected to see why the precondition cannot be proved. It could be a problem with the precondition, a problem at the call site, or a limitation of the prover.

  The postcondition of the called subprogram can be assumed to be valid at the point following the return, although the validity needs to be established by testing. In the example shown in Figure 2, testing would need to establish that the *getascii* function only returns a result in the range 0 through 127.

- The SPARK code is invoked from tested code

  In this situation testing would need to establish that, at each call, the precondition of the SPARK subprogram is met. Since the SPARK subprogram has been formally verified, at the point of return the subprogram's postcondition is known to be satisfied. Testing of the non-SPARK code can take advantage of this fact, thereby reducing the testing effort.

Hybrid verification can be performed within a single module; e.g., a package can specify different sections where SPARK analysis is or is not to be performed.

Conclusions

The FACE Ada 95 Safety capability sets are intended for applications that need to meet safety-and/or security requirements and that thus need to adhere to additional restrictions beyond those specified in the capability sets. The SPARK language, an Ada subset, was specifically designed to meet the needs of safety-critical and high-security software and is a natural candidate for implementing high-assurance FACE applications. In summary:

- The SPARK language is an excellent match for the FACE Ada 95 Safety capability sets and is likewise expected to be an excellent match for Ada 2012 Safety capability sets, when these are incorporated in a subsequent edition of the FACE Technical Standard. SPARK meets many of the restrictions in the Ada 95 sets, and the SPARK features outside the capability sets can be detected via standard Ada mechanisms (pragma Restrictions) or via a rule-based static analysis tool such as GNATcheck.
- The SPARK technology can be used for Ada 2012 and Ada 95 environments.
- Formal methods are a practical technique for real-world high-assurance systems, and can be introduced incrementally into an organization's verification infrastructure based on assurance requirements. For critical systems the Silver level (Absence of Run-Time Errors) is recommended.
- Formal methods can be combined with traditional testing ("hybrid verification"), realizing the benefits of both, and with smooth transitions in both directions: from testing to formal analysis for critical modules, from formal analysis to testing when proofs are not practical.
- When certification is required, for example under DO-178C, formal methods can be used for credit towards a variety of verification objectives and can eliminate or reduce the testing effort. Even if certification is not required, DO-178C and its Formal Methods supplement contain useful guidance for an organization responsible for fielding critical systems.

With the increasing importance of reliability, safety and security in modern airborne systems, formal methods are moving into mainstream software development. SPARK and hybrid verification can be an enabling technology for FACE component developers, providing not simply the portability that is the FACE approach's cornerstone objective but also the high assurance that is becoming more and more critical.

References

(Please note that the links below are good at the time of writing but cannot be guaranteed for the future.)

[1] RTCA DO-178C/EUROCAE ED-12C, *Software Considerations in Airborne Systems and Equipment Certification,* December 2011

[2] *Ada Reference Manual ISO/IEC 8652:2012(E) with Technical Corrigendum 1, Language and Standard Libraries*
http://www.ada-auth.org/standards/rm12_w_tc1/html/RM-TTL.html

[3] AdaCore and Altran; *SPARK 2014 Reference Manual*;
http://docs.adacore.com/spark2014-docs/html/lrm/

[4] *FACE™ Technical Standard, Edition 3.0*.
https://publications.opengroup.org/c17c/

[5] *What is Frama-C*
https://frama-c.com/what_is.html/

[6] RTCA DO-333/EUROCAE ED-216, *Formal Methods Supplement to DO-178C and DO-278A*, December 2011.

[7] AdaCore; *GNATcheck Reference Manual*
http://docs.adacore.com/live/wave/asis/html/gnatcheck_rm/gnatcheck_rm.html

[8] F. Pothon and Q. Ochem, *AdaCore Technologies for DO-178C / ED-12C; Version 1.1*; March 2017.
https://www.adacore.com/books/do-178c-tech

[9] AdaCore and Thales. *Implementation Guidance for the Adoption of SPARK*; January 2017;
https://www.adacore.com/books/implementation-guidance-spark

About the Author

Dr. Benjamin Brosgol is a senior member of the technical staff at AdaCore. He has been involved with programming language design and implementation throughout his career, concentrating on languages and technologies for high-integrity systems, with a focus on Ada and safety certification (DO-178B/C).
Dr. Brosgol has been an active member of the Operating System Subcommittee of the FACE Technical Work Group since 2017. He has participated in the development of the IDL-to-Ada mapping, has been a major contributor to the definition of the Ada 2012 capability sets, and was the principal author of the paper *Ada Language Run-Times and the FACE™ Technical Standard: Achieving Application Portability and Reliability*, which was presented at the September 2018 FACE TIM in Huntsville. Dr. Brosgol holds a BA in Mathematics from Amherst College, and MS and PhD degrees in Applied Mathematics from Harvard University.

About The Open Group FACE™ Consortium

The Open Group Future Airborne Capability Environment (FACE™) Consortium, was formed as a government and industry partnership to define an open avionics environment for all military airborne platform types. Today, it is an aviation-focused professional group made up of industry suppliers, customers, academia, and users. The FACE Consortium provides a vendor-neutral forum for industry and government to work together to develop and consolidate the open standards, best practices, guidance documents, and business strategy necessary for acquisition of affordable software systems that promote innovation and rapid integration of portable capabilities across global defense programs.

Further information on FACE Consortium is available at www.opengroup.org/face.

About The Open Group

The Open Group is a global consortium that enables the achievement of business objectives through technology standards. Our diverse membership of more than 600 organizations includes customers, systems and solutions suppliers, tools vendors, integrators, academics, and consultants across multiple industries.

The mission of The Open Group is to drive the creation of Boundaryless Information Flow™ achieved by:

- Working with customers to capture, understand, and address current and emerging requirements, establish policies, and share best practices
- Working with suppliers, consortia, and standards bodies to develop consensus and facilitate interoperability, to evolve and integrate specifications and open source technologies
- Offering a comprehensive set of services to enhance the operational efficiency of consortia
- Developing and operating the industry's premier certification service and encouraging procurement of certified products

Further information on The Open Group is available at www.opengroup.org.