AdaCore Technologies for

FACE™ SOFTWARE DEVELOPERS

Benjamin M. Brosgol & Dudrey Smith

AdaCore Technologies for

FACE™ Software Developers

Supporting the Future Airborne Capability Environment™

Benjamin M. Brosgol & Dudrey Smith

Version 1.0
March 2021

The AdaCore Technologies Series

The AdaCore Technologies Series is a collection of books targeted to software developers in critical domains. Each book explains how the Ada and SPARK programming languages, together with AdaCore's products, can reduce system life cycle costs and facilitate conformance with applicable software certification standards. Current titles in the series:

- *AdaCore Technologies for CENELEC EN 50128:2011*
  by Jean-Louis Boulanger & Quentin Ochem
- *AdaCore Technologies for DO-178C / ED-12C*
  by Frédéric Pothon & Quentin Ochem
- *AdaCore Technologies for Cyber Security*
  by Roderick Chapman & Yannick Moy
- *AdaCore Technologies for FACE™ Software Developers*
  by Benjamin M. Brosgol & Dudrey Smith

AdaCore has also prepared a variety of texts introducing Ada and SPARK to developers familiar with other language technologies, including Embedded C, MISRA C, C++, and Java.

All are available for download at https://www.adacore.com/books. For printed copies, please contact info@adacore.com.

About the Authors

Dr. Brosgol and Dr. Smith are AdaCore representatives in The Open Group's FACE™ Consortium.

Benjamin M. Brosgol

Dr. 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-assurance systems with a focus on Ada and safety certification (DO-178B/C). Ben is Vice Chair of the FACE Consortium's Technical Working Group, and an active member of the Operating Systems Segment Subcommittee and the Enterprise Architecture Subteam 25 (EA-25) .

Dudrey Smith

Dr. Smith is a senior embedded system development consultant at AdaCore. He has been involved with military and commercial embedded system/software development and certification for more than 40 years, with major leadership roles at companies including Lear Siegler, Smiths Aerospace, and General Electric Aviation Systems. Dudrey is a longstanding contributor to both the Business and Technical Working Groups of the FACE Consortium, and an active member of the Operating Systems Segment and Conformance Verification Subcommittees.

Foreword

AdaCore is a Principal Member of The Open Group's Future Airborne Capability Environment™ (FACE) Consortium. We have been actively involved with the FACE effort since 2012, participating in and contributing to both the Technical and Business Working Groups.

Our objective in the FACE Consortium is to help FACE software suppliers meet assurance requirements for reliability, safety, and security while realizing the portability and reusability benefits that come from FACE conformance. Among the languages that are called out in the FACE Technical Standard – C, C++, Ada and Java – the one that best promotes high assurance coupled with code portability is Ada.

As a software tool provider to the Aerospace and Defense community, we offer products that enable and encourage FACE software suppliers to use Ada for their applications. More specifically, AdaCore's FACE related products include:

- GNAT Pro Ada development environments targeted to RTOSes supplied by FACE Consortium members, such as Wind River's VxWorks 653 and Lynx Software Technologies' LynxOS-178;
- Efficient run-time libraries that are distributed with our cross-compilation environments and have been deployed in avionics systems certified at the highest Design Assurance Levels (DALs) of software standards such as DO-178B/C; and
- Static analysis tools including the formal methods-based and CWE-compatible SPARK Pro toolsuite, the CWE-compatible CodePeer deep static analyzer for Ada, and the GNATcheck coding standard verifier for Ada.

This book summarizes AdaCore's technologies and shows how they can help avionics suppliers develop and verify high-assurance FACE conformant software. The discussion is based on Edition 3.1 of the FACE Technical Standard and applies also to earlier versions.

Benjamin M. Brosgol
AdaCore
Bedford, Massachusetts
March 2021

Dudrey Smith
AdaCore and DS Consulting
Grand Rapids, Michigan
March 2021

info@adacore.com
www.adacore.com

Table of Contents

About the Authors iii

Foreword iv

Table of Contents v

1 Introduction 7

2 The FACE Approach 9

2.1 The FACE Reference Architecture 9

2.2 The OSS and Its Interface 11

2.3 Profiles 13

2.4 Capability Sets 13

2.5 Conformance Verification 14

3 Programming Languages for High-Assurance FACE Software 17

3.1 Ada 17

3.1.1 Ada language overview 17
3.1.2 Ada language background 18
3.1.3 Scalar ranges 19
3.1.4 Contract-based programming 19
3.1.5 Programming in the large 20
3.1.6 Generic templates 21
3.1.7 Object-Oriented Programming (OOP) 21
3.1.8 Concurrent programming 21
3.1.9 Systems programming 22
3.1.10 Real-time programming 22
3.1.11 High-integrity systems 22

3.2 SPARK 23

3.2.1 SPARK Basics 23
3.2.2 SPARK and the Ada Safety capability sets 25
3.2.3 Ease of Adoption: Levels of Adoption of Formal Methods 25
3.2.4 Hybrid Verification 26

4 Tools for FACE Software Development 28

4.1 AdaCore Tools and the Software Life Cycle 28

4.2 QGen Toolsuite for Model-Based Engineering 29

4.3 Static Verification: SPARK Pro 30

4.3.1 Powerful Static Verification 30
4.3.2 Minimal Run-Time Footprint 30
4.3.3 SPARK Pro Support for FACE Conformance 31
4.3.4 CWE Compatibility 31
4.4 GNAT Pro Ada Development Environments 32
4.4.1 GNAT Pro Enterprise 32
4.4.2 GNAT Pro Assurance 33
4.4.3 GNAT Pro Integrated Development Environments (IDEs) 33
4.4.4 GNAT Pro Support for FACE Conformance 35
4.5 GNAT Pro Ada Tools for Static Analysis 35
4.5.1 GNATcheck 35
4.5.2 GNATmetric 36
4.5.3 GNATstack 36
4.5.4 Time and Space Analysis 38
4.5.5 Semantic Analysis Tools—Libadalang 38
4.6 Static Verification: CodePeer 38
4.6.1 Early Error Detection 39
4.6.2 CWE Compatibility 39
4.7 Dynamic Analysis Tools 40
4.7.1 GNATtest 40
4.7.2 GNATemulator 40
4.7.3 GNATcoverage 41
4.8 Support and Expertise 41
5 Abbreviations 43
6 References 45
6.1 Cited References 45
6.2 FACE Related Articles by AdaCore Authors 46
Index 47

1 Introduction

The term “software crisis” was coined in the late 1960s and in general refers to the difficulty in producing, within budget, software that meets its requirements. “Chronic condition” may be a more apt term than “crisis”, since the problem of costly but unreliable software-intensive systems persists to this day. Despite (or perhaps in some cases because of) advances in both hardware and software technology, software too often remains frustratingly expensive to develop, verify, maintain, and port from one platform to another.

The U.S. Department of Defense (DoD) has not been immune to this issue, and in 2010 a group was chartered to address a specific and significant contributing factor: software that depends on platform-specific features, such as Real-Time Operating System (RTOS)-functionality or peripheral device characteristics, and that is therefore expensive to update for reuse in other contexts.

Thus was born the Future Airborne Capability Environment™ and the FACE™ Consortium, under the auspices of The Open Group, comprising representatives from the government, defense contractors, and suppliers of platform infrastructure and third-party toolsuites such as RTOSes and integrated software development environments. The idea was to exploit recognized software engineering principles in defining a standard software architecture and associated data model, and to leverage common open standards such as POSIX [IEEE 2017], ARINC 653 [ARINC 2019], and IDL [OMG 2018], in order to lower the cost of reusing airborne software components across multiple programs and platforms. The FACE approach accordingly defines a set of requirements for component reuse, and conformance procedures for certifying that a component meets these requirements.

It's important to realize what the FACE approach is *not* intended to do: namely, to demonstrate that a software component satisfies its functional requirements, or that it meets the objectives of a specific airworthiness standard such as MIL-HDBK-516C [DoD 2014] or DO-178C [RTCA 2011]. These are obviously important but are outside the scope of the FACE effort, and suppliers of FACE conformant software will need to separately show compliance with such standards.

A software supplier developing a component that is designed for FACE conformance (known as a *Unit of Conformance*, or *UoC*) therefore needs to meet two challenges in practice:

- Demonstrating that the UoC is indeed FACE conformant (and more generally meets the underlying goal of code portability), and
- Showing that the UoC meets project-specific requirements beyond FACE conformance-- in particular, verifying functional correctness or other assurance properties such as safety and/or security.

This book explains how AdaCore’s tools and technologies help meet these challenges.

2 The FACE Approach

The Preface to the FACE Technical Standard [OG 2020] sets the context:

> *The FACE approach addresses the affordability initiatives of today's military aviation domain. The FACE approach is to develop a Technical Standard for a software COE [Common Operating Environment] designed to promote portability and create software product lines across the military aviation domain. Several components comprise the FACE approach to software portability and reuse:*
>
> - *Business processes to adjust procurement and incentivize industry*
> - *Technical practices to promote development of reusable software components*
> - *A software standard to promote the development of portable components between differing avionics architectures*
>
> *The FACE approach allows software-based "capabilities" to be developed as software components that are exposed to other software components through defined interfaces. It also provides for the reuse of software across different hardware configurations.*
>
> *Ultimately, the FACE key objectives are to reduce development costs, integration costs, and time-to-field for avionics capabilities.*

This chapter summarizes the principal elements of the FACE approach.

2.1 The FACE Reference Architecture

The FACE technical approach is embodied in the FACE Technical Standard and basically comprises a reference architecture, open interfaces, and a common data architecture to facilitate inter-component communication. The current version of the Technical Standard is Edition 3.1, and several older editions (2.0, 2.1, 2.1.1, and 3.0) are in use and supported.

The FACE reference architecture captures a longstanding principle of software engineering: a component should be designed with a clear separation between its *interface*, which is stable and defines the services that the component provides to other components, and its *implementation*, which may change as the system evolves. That separation, sometimes known as *encapsulation*, makes it easier to port the component, along with any client code that uses the services of that component, to a new platform. The implementation of the component might need to be adapted, but the interface and the client code can stay the same.

The FACE software architecture is essentially an application of that principle to airborne systems. As shown in Figure 1, it comprises several layers, or *segments*, which abstract a system's platform-independent interfaces from its platform-dependent behavior:

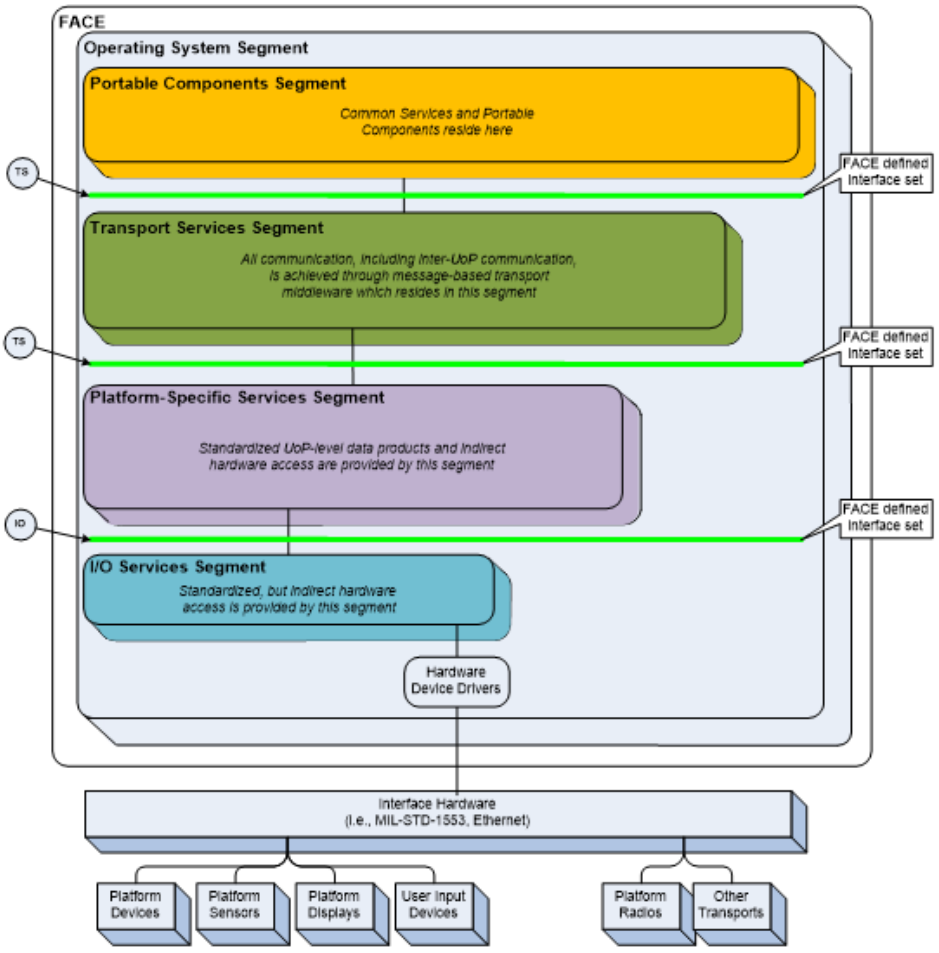

Figure 1: FACE Reference Architecture

- **Operating System Segment (OSS)**. The OSS is the software foundation for the other FACE segments and supplies services for partitioning, process/thread management, and memory management. It may also include functionality such as health monitoring, fault management, and life cycle management.
- **Input/Output Services Segment (IOSS)**. The IOSS provides a standard interface between PSSS UoCs and the IO devices supplied for a given platform.
- **Platform-Specific Services Segment (PSSS)**. The PSSS comprises device services, common services (e.g., logging, device protocol mediation), and graphics services. These supply a standard interface between PCS UoCs and the IOSS.
- **Transport Services Segment (TSS)**. The TSS supplies communication services for UoCs from other segments, including distribution, routing, state persistence, and data conversion.

- **Portable Components Segment (PCS)**. A PCS UoC supplies specific application functionality / "business logic", independent of the specifics of the underlying hardware and sensors and not tied to a specific data transport or OS implementation. A PCS UoC thus uses only the TSS Interface for data communication and only the OSS Interface for OS support.

A Unit of Portability (UoP) is a UoC in the PCS or PSSS.

Different parts of an airborne system that are developed independently may need to communicate, which raises several issues:

- **Multilanguage interoperability**. Different UoCs may be written in different languages; the FACE Technical Standard specifically calls out C, C++, Ada (both Ada 95 and Ada 2012), and Java. Data cannot, in general, be passed directly between components in different languages, since the formatting conventions may differ. The FACE Technical Standard addresses this issue by requiring PCS UoCs to communicate through TSS functions, and by expressing the TSS interfaces in a language-agnostic manner, through IDL. An IDL compiler will then generate appropriate specifications / header files in the UoC's programming language, which may be different for the sending and receiving UoC. The TSS function will perform the necessary data conversion.
- **Data view consistency**. Communicating components need to have the same interpretation of the data being sent: if one UoC is sending position data as WGS-84 longitude-latitude-altitude, but the receiving UoC is expecting earth-centered rectangular (XYZ) coordinates, then the mismatch needs to be identified early in the software life cycle. The FACE Technical Standard addresses this issue through data architecture requirements. The data architecture is expressed in the FACE Data Model Language, which is based on the Open Universal Domain Description Language (Open UDDL). The FACE Data Model Language "enforces a multi-level approach to modelling entities and their associations at the conceptual, logical, and platform levels, enabling gradual and varying degrees of abstraction" [OG 2020, §3.3.2]. The FACE conformance procedures include a Shared Data Model (SDM) expressed in the Face Data Model Language, and each PCS UoC using TSS interfaces needs to furnish a UoP Supplied Model (USM). The USM can be based on a Domain-Specific Data Model (DSDM) consistent with the FACE SDM.
- **Initialization / "loose coupling"**. The TSS defines a variety of data types and interfaces, and a UoC needs a handle on the implementation of the TSS interfaces that it uses. To avoid an unwanted coupling between the "client" UoC and a specific implementation of the TSS service that it needs, the FACE Technical Standard defines an IDL template for the *Injectable* interface, parameterized by interface type, which embodies the "dependency injection" software design pattern. This template (and thus each instance) defines a basic *Set_Reference* function, called during startup, to supply the reference to the implementation of the requested interface [OG 2020, §I.1].

2.2 The OSS and Its Interface

An application can use a variety of run-time services: memory management, concurrency control, exception handling, I/O, and so on. These have traditionally been a source of code non-portability, with

calls on platform-dependent or RTOS vendor-specific APIs. The FACE reference architecture's OS Segment (Figure 2) addresses this issue in several ways:

- **Standard industry APIs.** The FACE Technical Standard piggybacks on the widely used POSIX and ARINC 653 APIs for run-time functionality. These are very comprehensive APIs, and an application with safety or security requirements needs to restrict itself to a subset, or what FACE calls a "profile", which provides the necessary determinism and time and/or space partitioning. Profiles are described in more detail below.
- **Standard language syntax for run-time functionality.** The FACE Technical Standard reflects the reality that run-time services are sometimes invoked not through explicit function calls on an API but rather through programming language syntax. For example, an Ada application would typically create threads and manage inter-thread communication using the language's tasking features, which are compiled into calls on run-time library functions provided by the compiler vendor. Programming language run-time support libraries are thus considered to be part of the OSS when (as is typically the case) the interface to their services is through standard language syntax rather than FACE APIs.

The run-time library for a language like Ada, C++, or Java differs from the other OSS components in a critical way. Rather than being specified by an API invoked explicitly from UoC source code – which would be overly constraining, given the differences across compiler implementations – the interface to the run-time is defined by a set of standard language features (a so-called *capability set*). Portability of a UoC at the source code level is achieved not by calls on a POSIX or ARINC 653 API but rather through standard language syntax that every compiler will accept.

The implementation of a language's run-time library may or may not be implemented through calls on the FACE APIs. More generally the interface between an OSS component and the lower-level services needed in its implementation (the so-called "Bottom-side" interface) is not defined or constrained by the FACE Technical Standard.

The lowest layer of the Operating System Segment comprises the implementation of the various services. These are not defined by the FACE Technical Standard and may in turn make use of low-level functions that are platform dependent.

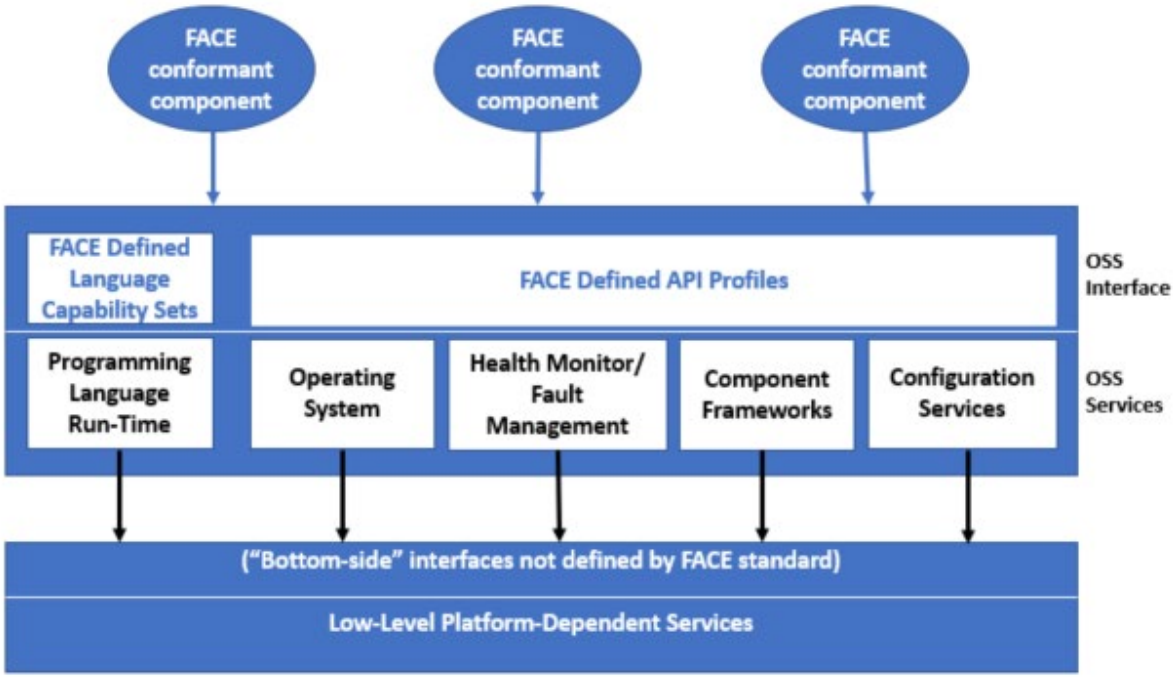

Figure 2: Operating System Segment and Its Interface

2.3 Profiles

FACE conformant components can be deployed in a variety of contexts with differing requirements for safety and/or security. The FACE Technical Standard therefore defines a set of *profiles* (API subsets) for the OSS interface. In increasing order of generality:

- *Security:* This is a minimal interface, designed to support applications with high security assurance requirements running in a single address space (e.g., a POSIX process or ARINC 653 partition). It guarantees real-time deterministic behavior and requires time and space partitioning.
- *Safety:* This consists of two sub-profiles, *Safety Base* and *Safety Extended*, which are aimed at systems with safety certification requirements. Both Safety Sub-profiles guarantee real-time deterministic behavior, require ARINC 653 support, and also require time and space partitioning. The Safety Base Sub-profile supports single POSIX process applications; the Safety-Extended Sub-profile includes optional support for multiple POSIX processes.
- *General-purpose:* In this profile real-time determinism is not guaranteed, space partitioning is required, and time partitioning is optional. The general-purpose profile is geared toward components at low levels of safety/security assurance. ARINC 653 support and multiple POSIX processes are optional.

2.4 Capability Sets

The rationale underlying the provision of OSS profiles – higher assurance levels imply restrictions on generality – also applies to the programming language(s) used to code the UoC. The FACE Technical Standard thus defines corresponding sets of restrictions ("capability sets") for C, C++, Ada, and Java; in order of increasing generality:

- Safety-Base & Security
- Safety-Extended
- General-Purpose

The Security and Safety capability sets specify subsets of run-time functionality and also restrict other General-Purpose features that could be problematic at higher assurance levels. For each capability set, the functionality supported across the different languages is roughly comparable. For example, dynamic allocation is allowed but deallocation is forbidden[1] in the Safety Base & Security capability sets for C++, Ada 95, and Ada 2012.

Edition 3.1 of the FACE Technical Standard separately defines the capability sets for Ada 95 and Ada 2012. These are compatible in the sense that an Ada program satisfying the Ada 95 capability set restrictions will also satisfy the Ada 2012 restrictions. All Ada 2012 capability sets allow contract-based programming, a language feature for augmenting code contracts such as subprograms with program state assertions that can be verified either with run-time checks or, via appropriate tool support, through static analysis.

A software supplier producing a UoC that invokes OSS services needs to decide which profile / capability set the UoC is designed to target. The UoC is then prohibited from invoking APIs that are excluded by that profile and is analogously prohibited from using language features excluded by the corresponding capability set.

2.5 Conformance Verification

The FACE Conformance Program defines the processes to verify, certify, and provide formal recognition that registered software conforms to the FACE Technical Standard. The first step, verification, is a "technical evaluation of the software against the FACE Technical Standard … [and] is completed by an approved FACE VA [Verification Authority]" [OG 2016, §2.2].

The basis of the verification process is the Conformance Verification Matrix (CVM), a spreadsheet that enumerates each requirement in the FACE Technical Standard and specifies:

- the segment(s) to which the requirement applies,
- the verification method to be used (inspection or test),
- the conformance artifacts to be prepared, and
- any conditional requirements (e.g., the applicable profile(s) or programming language).

For example, Requirement 449 in the CVM for the FACE Technical Standard Edition 3.0 [OG 20xx] relates to the permitted usage of Ada 95 features:

[1] The ability to allocate but not deallocate dynamic memory leaves open the possibility of storage leakage, an obvious hazard / vulnerability. For airworthiness certification the UoC supplier would need to demonstrate how storage leakage would be prevented (e.g., by showing that no dynamic allocation occurs after system initialization), but this is outside the scope of the FACE requirements.

> *1. UoCs using an Ada Run-Time supplied by the OS shall be restricted to the Programming Language features for the selected capability set.*

This requirement applies to UoCs in the PSSS, PCS, TSS, and IOSS and is to be verified by Inspection. The conformance artifact is a "Fully Tested Requirement", i.e., evidence from the inspection (which could be the result of a static analysis tool). The conditional requirement is "Ada programming language".

More generally, verifying conformance entails testing and, in particular, running the Conformance Test Suite (CTS) as described in the Conformance Certification Guide [OG 2016, §2.4].

> *The FACE Conformance Program focuses on the portability of software, not the functionality. The FACE Conformance Program does not require the execution of the software. The UoC source code is compiled to a single target by the Software Supplier. The resulting object code is delivered to the VA and is analyzed for conformance through a linking process to ensure the compiled code will interface with other FACE UoCs. Only a linking of the object code is performed; the interfaces are not executed as part of this test.*

And more specifically [OG 2016, §3.2.3]:

> *The Test Suite is a program that evaluates a UoC against the FACE Technical Standard. This is performed in two ways:*
>
> - *An evaluation is performed by linking the UoC object code to sample applications that provide FACE conformant interfaces. Any failed linking is identified as non-conformance to the standard.*
> - *The UoC Data Model files are analyzed by the Test Suite against the FACE Technical Standard. Any failure in this analysis is identified as a non-conformance to the standard.*

The test cases thus are intended to demonstrate that, first, the UoC provides the services that are required, and, second, it does not use any interfaces that are prohibited in the profile / capability set that is targeted. For a UoC in the PCS, FACE conformance means that reusing the UoC in another platform with a FACE conformant RTOS may be as simple as recompiling the source code.

3 Programming Languages for High-Assurance FACE Software

This chapter explains how FACE software developers can benefit from the Ada language and its formally analyzable SPARK subset. Unless explicitly stated otherwise, the Ada discussion applies to both Ada 95 and Ada 2012.

3.1 Ada

The choice of programming language(s) is one of the fundamental decisions during software design. The source code is the artifact that is developed, verified, and maintained, and it is also the subject of much of the analysis / inspection required for certification against domain-specific standards. Although in principle almost any programming language could be used to develop high-assurance software, in practice software life-cycle costs are reduced when the chosen language has been explicitly designed for reliability, safety and security.

The FACE Technical Standard specifically cites four candidate languages – C, C++, Ada, and Java – and, of these, Ada best meets the needs of high-assurance systems. Ada is especially suitable for UoCs that need to conform to the security profile or one of the safety profiles, avoiding C and C++ vulnerabilities such as buffer overrun and integer overflow, and also avoiding Java's nondeterminism issues (garbage collection, implementation-dependent thread semantics).

Ada helps meet high-assurance requirements through its support for sound software engineering practice, compile-time checks that enforce type safety, and the enforcement of dynamic constraints such as array index bounds and scalar ranges. As will be explained below, the SPARK subset of Ada shares these benefits, with the added advantage that the dynamic constraints are enforced through mathematics-based static analysis, thereby avoiding run-time overhead.

3.1.1 Ada language overview

Ada is multi-faceted. From one perspective it is a classical stack-based general-purpose language (i.e., unlike languages like Java, it does not require garbage collection), and it is not tied to any specific development methodology. It offers:

- a simple syntax designed for human readability;
- structured control statements;
- flexible data composition facilities;
- strong type checking;
- traditional features for code modularization ("subprograms");
- standard support for "programming in the large" and module reuse (packages, Object-Oriented Programming, child libraries, generic templates);
- a mechanism for detecting and responding to exceptional run-time conditions ("exception handling"); and

- high-level concurrency support ("tasking") along with a deterministic subset (the *Ravenscar profile*) appropriate in applications that need to meet high-assurance certification requirements.

The language standard also includes:

- an extensive predefined environment with support for I/O, string handling, math functions, containers (Ada 2012), and more;
- a standard mechanism for interfacing with other programming languages (such as C and C++); and
- specialized needs annexes for functionality in several domains (Systems Programming, Real-Time Systems, Distributed Systems, Numerics, Information Systems, and High-Integrity Systems).

Portability is the driving force behind the FACE approach and was also a key goal for Ada. The challenge for a programming language is to define the semantics in a platform-independent manner but not sacrifice run-time efficiency. Ada achieves this in several ways.

- Ada provides a high-level model for concurrency (tasking), memory management, and exception handling, with standard semantics across all platforms that can be mapped to the most efficient services provided by the target system. This is entirely consistent with the treatment of language run-times in the FACE Reference Architecture.
- The developer can express the logical properties of a type (such as integer range, floating-point precision, and record fields/types) in a machine-independent fashion, which the compiler can then map to an efficient underlying representation.
- The physical representation of data structures (layout, alignment, and addresses) is sometimes specified by system requirements, and Ada allows this to be defined in the program logic but separated from target-dependent properties for ease of maintenance.
- Platform-specific characteristics such as machine word size are encapsulated in an API, so that references to these values are through a standard syntax. Likewise, Ada defines a standard type Address and associated operations, again facilitating the portability of low-level code.

3.1.2 Ada language background

Ada was designed for large, long-lived applications – and embedded systems in particular – where reliability, maintainability, and efficiency are essential. Under sponsorship of the U.S. Department of Defense, the language was originally developed in the early 1980s (this version is generally known as Ada 83) by a team led by Jean Ichbiah at CII-Honeywell-Bull in France. Ada was revised and enhanced in an upward-compatible fashion in the early 1990s, under the leadership of Tucker Taft from Intermetrics in the U.S. The resulting language, Ada 95, was the first internationally standardized object-oriented language. Under the auspices of the International Organization for Standardization (ISO), a further (minor) revision was completed as an amendment to the standard; this version of the language is known as Ada 2005. Additional features (including support for contract-based programming in the form of subprogram pre- and postconditions and type invariants) were added in the most recent version of the

language standard, Ada 2012 (see [ACAA 2016], [Ba 2014], or [Ba 2015] for information about Ada). A new version, termed “Ada 202x”, is in progress and is expected to be completed in 2021 or 2022.

The name “Ada” is not an acronym; it was chosen in honor of Augusta Ada Lovelace (1815-1852), a mathematician who is sometimes regarded as the world’s first programmer because of her work with Charles Babbage. She was also the daughter of the poet Lord Byron.

The Ada language has a long and continuing worldwide usage in high-assurance / safety-critical / high-security domains including military and commercial aircraft avionics, air traffic control, space applications, railroad systems, and other domains (such as automotive and medical) where software failures can have catastrophic consequences. With its embodiment of modern software engineering principles Ada is especially appropriate for teaching in both introductory and advanced computer science courses, and it has been the subject of significant university research, especially in the area of real-time technologies.

AdaCore has a long history and close connection with the Ada programming language. Company members worked on the original Ada 83 design and review and played key roles in the Ada 95 project as well as the subsequent revisions. The initial GNAT compiler was delivered at the time of the Ada 95 language’s standardization, thus guaranteeing that users would have a quality implementation for transitioning to Ada 95 from Ada 83 or other languages.

The following subsections provide additional detail on Ada language features.

3.1.3 Scalar ranges

Unlike languages based on C (such as C++, Java, and C#), Ada allows the programmer to simply and explicitly specify the range of values that are permitted for variables of scalar types (integer, floating-point, fixed-point, and enumeration types). The attempted assignment of an out-of-range value causes a run-time error. The ability to specify range constraints makes programmer intent explicit and makes it easier to detect a major source of coding and user input errors. It also provides useful information to static analysis tools and facilitates automated proofs of program properties.

Here’s an example of an integer scalar range:

```
Score : Integer range 1..100;
N     : Integer;
...
Score := N;
-- A run-time check verifies that N is within the range 1 through 100, inclusive
-- If this check fails, a Constraint_Error exception is raised
```

3.1.4 Contract-based programming

Ada 2012 allows extending a subprogram specification or a type/subtype declaration with a contract (a Boolean assertion). Subprogram contracts take the form of *preconditions* and *postconditions*. Through

contracts the developer can formalize the intended behavior of the application, and can verify this behavior by testing, static analysis, or formal proof.

Here's a skeletal example that illustrates contact-based programming; a `Table` object is a fixed-length container for distinct `Float` values.

```
package Table_Pkg is
   type Table is private;  -- Encapsulated type

   function Is_Full  (T    : in Table) return Boolean;
   function Contains (T    : in Table;
                      Item : in Float) return Boolean;

   procedure Insert (T : in out Table; Item: in Float)
     with Pre  => not Is_Full(T) and not Contains(T, Item),
          Post => Contains(T, Item);

   procedure Remove (T : in out Table; Item: in Float);
     with Pre  => Contains(T, Item),
          Post => not Contains(T, Item);
   ...
private
   ... -- Full declaration of Table
end Table_Pkg;
```

A compiler option controls whether the pre- and post-conditions are checked at run time. If checks are enabled, a failure raises the `Assertion_Error` exception.

Ada 2012 goes further still, allowing type invariants and subtype predicates to specify *precisely* what is and isn't valid for any particular (sub)type, including composite types such as records and arrays. For example, one can easily specify that field `Max_A` in the `Launching_Pad` structure below is the maximal value of angle allowed given the distance D to the center of the launching pad and the height H of the rocket, with the guarantee that automatic run-time checks will be inserted by the compiler to verify this predicate as well as constraints on the individual fields:

```
type Launching_Pad is record
   D, H  : Length;
   Max_A : Angle;
end record
  with Predicate => Angle (Arctan (H, D)) <= Max_A;
```

3.1.5 Programming in the large

The original Ada 83 design introduced the *package* construct, a feature that supports encapsulation ("information hiding") and modularization, and which allows the developer to control the namespace that is accessible within a given compilation unit. Ada 95 introduced the concept of "child units," adding

considerable flexibility and easing the design of very large systems. Ada 2005 extended the language's modularization facilities by allowing mutual references between package specifications, thus making it easier to interface with languages such as Java.

3.1.6 Generic templates

A key to reusable components is a mechanism for parameterizing modules with respect to data types and other program entities, for example a stack package for an arbitrary element type. Ada meets this requirement through a facility known as "generics"; since the parameterization is done at compile time, run-time performance is not penalized. Ada generics are analogous to C++ templates, but with considerably more compile-time checking.

3.1.7 Object-Oriented Programming (OOP)

Ada 83 was object-based, allowing the partitioning of a system into modules (packages) corresponding to abstract data types or abstract objects. Full OOP support was not provided since, first, it seemed not to be required in the real-time domain that was Ada's primary target, and second, the apparent need for automatic garbage collection in an OO language would have interfered with predictable and efficient performance.

However, large real-time systems often have components such as graphical user interfaces (GUIs) that do not have real-time constraints and that could be most effectively developed using OOP features. In part for this reason, Ada 95 added comprehensive support for OOP, through its "tagged type" facility: classes, polymorphism, inheritance, and dynamic binding. These features do not require automatic garbage collection; instead, definitional features introduced by Ada 95 allow the developer to supply type-specific storage reclamation operations ("finalization"). Ada 2005 brought additional OOP features including Java-like interfaces and traditional `X.P(...)` notation for invoking operation `P(...)` on object `X`.

Ada is methodologically neutral and does not impose a "distributed overhead" for OOP. If an application does not need OOP, then the OOP features do not have to be used, and there is no run-time penalty.

See [Ba 2014] or [Ad 2016] for more details.

3.1.8 Concurrent programming

Ada supplies a structured, high-level facility for concurrency. The unit of concurrency is a program entity known as a "task." Tasks can communicate implicitly via shared data or explicitly via a synchronous control mechanism known as the *rendezvous*. A shared data item can be defined abstractly as a "protected object" (a feature introduced in Ada 95), with operations executed under mutual exclusion when invoked from multiple tasks. Asynchronous task interactions are also supported, specifically timeouts and task termination. Such asynchronous behavior is deferred during certain operations, to prevent the possibility of leaving shared data in an inconsistent state. Mechanisms designed to help take advantage of multi-core architectures were introduced in Ada 2012.

3.1.9 Systems programming

Both in the "core" language and the Systems Programming Annex, Ada supplies the necessary features for low-level / hardware-specific processing. For example, the programmer can specify the bit layout for fields in a record, define alignment and size properties, place data at specific machine addresses, and express specialized code sequences in assembly language. Interrupt handlers can also be written in Ada, using the protected type facility.

3.1.10 Real-time programming

Ada's tasking facility and the Real-Time Systems Annex support common idioms such as periodic or event-driven tasks, with features that can help avoid unbounded priority inversions. A protected object locking policy is defined that uses *priority ceilings*; this has an especially efficient implementation in Ada (mutexes are not required) since protected operations are not allowed to block. Ada 95 defined a standard task dispatching policy that basically requires tasks to run until blocked or preempted, and Ada 2005 introduced several others including Earliest Deadline First.

3.1.11 High-integrity systems

With its emphasis on sound software engineering principles Ada supports the development of high-integrity applications, including those that need to be certified against safety standards such as RTCA DO-178C for avionics, CENELEC EN 50128 for rail systems and security standards such as the Common Criteria. For example, strong typing means that data intended for one purpose will only be accessed via operations that are legal for that data item's type; errors such as treating pointers as integers (or vice versa) are prevented[2]. And Ada's array bounds checking prevents buffer overrun vulnerabilities that are common in C and C++.

However, the full language may be inappropriate in a safety- or security-critical application, since the generality and flexibility of some features – especially those with complex run-time semantics – complicate analysis and could interfere with traceability / certification requirements. Ada addresses this issue by supplying a compiler directive, `pragma Restrictions`, that allows constraining the language features to a well-defined subset (for example, excluding dynamic OOP facilities).

The evolution of Ada has seen the continued increase in support for safety-critical and high-security applications. Ada 2005 standardized the Ravenscar Profile [DB 2001], a collection of concurrency features that are powerful enough for real-time programming but simple enough to make certification practical. Ada 2012 introduced contract-based programming facilities, allowing the programmer to specify preconditions and/or postconditions for subprograms, and invariants for encapsulated (private) types. These can serve both for run-time checking and as input to static analysis tools.

In brief, Ada is an internationally standardized language combining object-oriented programming features, well-engineered concurrency facilities, real-time support, and built-in reliability through both compile-time and run-time checks. Ada addresses the real issues facing software developers today and,

[2] Low-level code sometimes needs to defeat the language's type checking (for example treating a pointer as an integer), and that is allowed in Ada but with explicit syntax that reveals the programmer intent.

with its support for source code portability, run-time efficiency, and program reliability, the language is especially well suited for writing FACE applications.

3.2 SPARK

3.2.1 SPARK Basics

SPARK[3] ([MC 2015], [AA 2021]) is a software development technology (programming language and verification toolset) specifically designed for engineering ultra-low defect level applications, for example where safety and/or security are key requirements. SPARK Pro is the commercial-grade offering of the SPARK technology developed by AdaCore, Altran, and Inria. As will be discussed below, the main component in the toolset is GNATprove, which performs formal verification on SPARK code.

SPARK has an extensive industrial track record. Since its inception in the late 1980s it has been used worldwide in a range of industrial applications such as civil and military avionics, air traffic management / control, railway signaling, cryptographic software, medical devices, automotive systems, and cross-domain solutions. SPARK 2014 is the most recent version of the technology.

The SPARK language is a large 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, safe pointers, contract-based programming (subprogram pre- and postconditions, scalar ranges, type/subtype predicates), Object-Oriented Programming, and the Ravenscar subset of the tasking features.

Principal exclusions are side effects in functions and expressions, problematic aliasing of names, exception handling, and most tasking features.

Two major design goals of SPARK are the provision of an *unambiguous* and *formal* semantics, which therefore permits the *soundness* of static verification: i.e., the absence of "false negatives". If the SPARK tools report that a program does not have a specific vulnerability, such as a reference to an uninitialized variable, then that conclusion can be trusted with mathematical certainty. Soundness builds confidence in the tools, provides evidence-based assurance, completely removes many classes of dangerous defects, and significantly simplifies subsequent verification effort (e.g., testing), owing to less rework, and in some cases can eliminate these activities entirely.

SPARK offers the flexibility of configuring the language on a per-project basis. Restrictions can be fine-tuned based on the relevant coding standards or run-time environments.

SPARK code can easily be combined with full Ada code or with C, so that new systems can be built on and reuse legacy codebases. Moreover, the same code base can have some sections in SPARK and

[3] Note that this language /.technology is totally unrelated to the Apache SPARK analytics framework, or the SPARC CPU Instruction Set Architecture.

others excluded from SPARK analysis (and indeed SPARK and non-SPARK code can be mixed in the same package or subprogram).

Software verification typically involves extensive testing, including unit tests and integration tests. Traditional testing methodologies are a major contributor to the high delivery costs for safety-critical software. Furthermore, testing can never be complete and thus may fail to detect errors. SPARK addresses this issue by allowing automated proof to be used to demonstrate program integrity properties up to functional correctness at the subprogram level, either in combination with or as a replacement for unit testing. In the high proportion of cases where proofs can be discharged automatically the cost of writing unit tests may be completely avoided. Moreover, verification by proofs covers all execution conditions and not just a sample.

Figure 3 shows an example of SPARK code.

```
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 3: SPARK Example with Contracts

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
- 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. 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.

3.2.2 SPARK and the Ada Safety capability sets

The Ada 95 and Ada 2012 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 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 (see page 35). The FACE UoC 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.

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.

3.2.3 Ease of Adoption: 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 [AT 2020], 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.

3.2.3.1 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 it 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).

3.2.3.2 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.

3.2.3.3 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.

3.2.3.4 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.

3.2.3.5 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.

3.2.4 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 4 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 4: 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 4, 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

  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.

4 Tools for FACE Software Development

This chapter explains how FACE software developers can benefit from AdaCore's products. The advantages stem from reduced life cycle costs for developing and verifying high-assurance airborne software; and more specifically in a FACE context, several tools also help in demonstrating that a UoC conforms with the FACE requirements.

4.1 AdaCore Tools and the Software Life Cycle

The software life cycle is often depicted as a "V" diagram, and Figure 5 shows how AdaCore's major products fit into the various stages. Although the stages are rarely performed as a single sequential

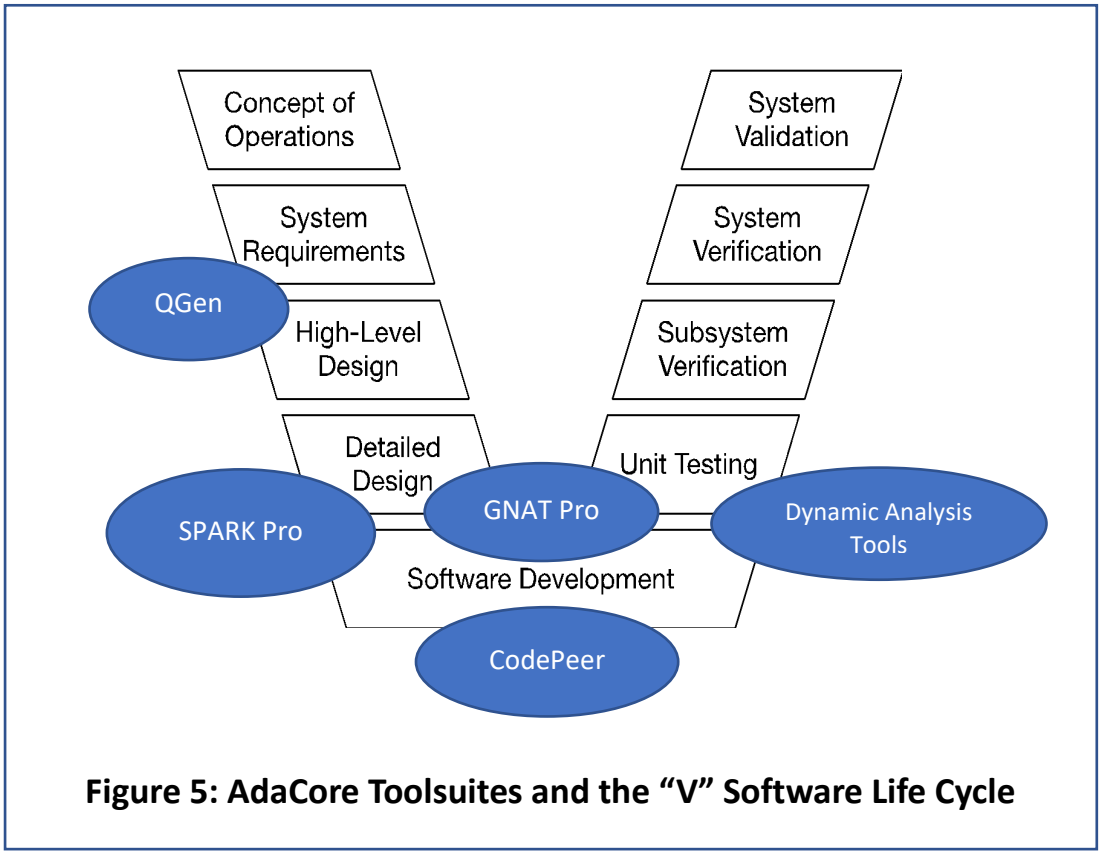

Figure 5: AdaCore Toolsuites and the "V" Software Life Cycle

process – the phases typically involve feedback / iteration, and requirements often evolve as a project unfolds – the "V" chart is useful in characterizing the various kinds of activities that occur.

In summary:

- The *QGen* model-based engineering environment (see page 29) applies during the Systems Requirements and High-Level Design phases. It is based around a qualifiable code generator from a safe subset of Simulink® and Stateflow® models.
- The *SPARK Pro* static analysis toolsuite (see page 30) applies during Detailed Design and Software Development. It includes a proof tool that verifies properties ranging from correct information flows to functional correctness.

- The *GNAT Pro* Ada development environment (see page 32) applies during Detailed Design, Software Development, and Unit Testing. It consists of gcc-based program build tools, an integrated and tailorable graphical user interface, a comprehensive set of static and dynamic analysis tools, and a variety of supplemental libraries.
- The *CodePeer* advanced Ada static analysis tool (see page 38) applies during Software Development. It can be used retrospectively to detect vulnerabilities in existing codebases, and/or during new projects to prevent errors from being introduced.
- *Dynamic analysis tools* (see page 40) apply during Software Development and Unit Testing. One such tool, *GNATcoverage*, supports code coverage and reporting at various levels of program construct granularity.

The following sections describe the tools in more detail and show how they can assist in developing and verifying FACE conformant software.

4.2 QGen Toolsuite for Model-Based Engineering

QGen is a qualifiable and tunable code generation and model verification toolsuite for a safe subset of Simulink® and Stateflow® models. The selected feature set ensures code generation that is appropriate for critical systems, leaving out features that might result in unpredictable behavior or potentially unsafe source code. The qualifiable QGen code generator translates control system models into source code in either the portable MISRA subset of C, or the SPARK subset of Ada. The generated code is suitable for formal analysis and for projects following software standards such as MIL-HDBK-516C, DO-178C, ISO 26262, or EN 50128.

The QGen tool suite additionally includes both static model verification and interactive model-level debugging of the generated code. The QGen model-level debugger provides a side-by-side view of the model and the generated code, allowing the developer to set breakpoints; to view, update and compare signal values; and to step through execution. The QGen debugger can be used for testing the generated code as well as any hand-written code, on the host or the final target. It allows the user to perform back-to-back comparison against expected values for a block or the model as a whole, while delving into the details of a particular subsystem whenever needed. By displaying the model together with the generated source code, the QGen debugger provides a productive bridge between control engineering and software engineering.

The QGen automatic code generator is being qualified in compliance with the DO-178C / ED-12C standard at Tool Qualification Level 1 (TQL-1), with qualification anticipated in early 2022. Some code generators rely on a separate verification tool to check their generated source code, but QGen at TQL-1 allows developers to use the generated code without any extra steps, streamlining the critical-system development and verification process. With QGen, the supported subset of the modeling language is clearly defined together with the expected structure of the generated code, and is coupled with tests that verify the precise match between model simulation results and the run-time semantics of the generated target code.

Although model-based development of control logic is outside the scope of the FACE Technical Standard, a toolsuite such as QGen can increase the productivity of FACE UoC developers by automating the translation of system requirements into source code. The MISRA-C and SPARK code generated by QGen meet the requirements of all of the FACE profiles and capability sets.

4.3 Static Verification: SPARK Pro

SPARK Pro is an advanced static analysis toolsuite for the SPARK subset of Ada, bringing mathematics-based confidence to the verification of critical code. Built around the GNATprove formal analysis and proof tool, SPARK Pro combines speed, flexibility, depth and soundness, while minimizing the generation of "false alarms". It can be used for new high-assurance code (including enhancements to or hardening of existing codebases at lower assurance levels, written in full Ada or other languages such as C) or projects where the existing high-assurance coding standard is sufficiently close to SPARK to ease transition.

4.3.1 Powerful Static Verification

The SPARK language supports a wide range of static verification techniques. At one end of the spectrum is basic data- and control-flow analysis, i.e., exhaustive detection of errors such as attempted reads of uninitialized variables, and ineffective assignments (where a variable is assigned a value that is never read). For more critical applications, dependency contracts can constrain the information flow allowed in an application. Violations of these contracts – potentially representing violations of safety or security policies – can then be detected even before the code is compiled.

In addition, SPARK supports mathematical proof and can thus provide high confidence that the software meets a range of assurance requirements: from the absence of run-time exceptions, to the enforcement of safety or security properties, to compliance with a formal specification of the program's required behavior.

As described earlier (see page 25), the SPARK technology can be introduced incrementally into a project, based on the assurance requirements. Each level, from Bronze to Platinum, comes with associated benefits and costs.

4.3.2 Minimal Run-Time Footprint

For the most secure systems (for example, embedded cryptographic devices), a developer has to worry about and justify the presence of *all* the code in a delivered system. Guidance talks of "minimizing the trusted computing base", which really means just making the delivered system as small as possible. There is also the problem of Commercial Off-the-Shelf (COTS) components: if a system uses a COTS library or operating system, then how are these to be evaluated or verified without the close (and probably expensive) cooperation of the COTS vendor?

For the most critical embedded systems, SPARK supports the so-called "Bare-Metal" development style, where SPARK code is running directly on a target processor with little or no COTS libraries or operating system at all. SPARK is also designed to be compatible with GNAT Pro's Zero FootPrint (ZFP) run-time library. In a Bare-Metal/ZFP development, every byte of object code can be traced to the application's

source code, and accounted for. This can be particularly useful for systems that must withstand evaluation by a national technical authority or regulator.

SPARK code can also run with a specialized run-time library on top of an RTOS such as Wind River's VxWorks 653 or Lynx Software Technologies' LynxOS-178, or with a full Ada run-time library and a commercial desktop operating system. The choice is left to the system designer, not imposed by the language.

4.3.3 SPARK Pro Support for FACE Conformance

The major value that SPARK Pro brings to a software project is the added assurance that comes from formal analysis, but SPARK Pro also offers some specific benefits in the context of FACE certification.

- The tool enforces a number of restrictions in the Ada 95 and Ada 2012 Safety capability sets. For example, the use of tasking constructs outside the Ravenscar subset will be flagged.
- The full Ada language has several implementation dependencies that can result in the same source program yielding different results when compiled by different compilers. For example, the evaluation order in expressions is not specified, and different orderings may produce different values if one of the terms has a side effect. (A complete discussion of this issue and its mitigation may be found in [Br 2021].) Such implementation dependencies are either prohibited in SPARK and thus detected by SPARK Pro, or else they do not affect the computed result. In either case the use of SPARK Pro eases the effort in porting the code from one environment to another, which is a major goal of the FACE approach.

4.3.4 CWE Compatibility

SPARK Pro detects a number of dangerous software errors in The MITRE Corporation's Common Weakness Enumeration, and the tool has been certified by the MITRE Corporation as a "CWE-Compatible" product [Mi 20xx].

The following table lists the CWE weaknesses detected by SPARK Pro:

CWE Weakness	Description
CWE 119, 120, 123, 124, 125, 126, 127, 129, 130, 131	Buffer overflow/underflow
CWE 136, 137	Variant record field violation, Use of incorrect type in inheritance hierarchy
CWE 188	Reliance on data layout
CWE 190, 191	Numeric overflow/underflow
CWE 193	Off-by-one error
CWE 194	Unexpected sign extension
CWE 197	Numeric truncation error
CWE 252, 253	Unchecked or incorrectly checked return value
CWE 366	Race Condition
CWE 369	Division by zero

CWE Weakness	Description
CWE 456, 457	Use of uninitialized variable
CWE 466, 468, 469	Pointer errors
CWE 476	Null pointer dereference
CWE 562	Return of stack variable address
CWE 563	Unused or redundant assignment
CWE 682	Range constraint violation
CWE 786, 787, 788, 805	Buffer access errors
CWE 820	Missing synchronization
CWE 821	Incorrect synchronization
CWE 822, 823, 824, 825	Pointer errors
CWE 835	Infinite loop

4.4 GNAT Pro Ada Development Environments

AdaCore's GNAT Pro language toolsuite comes in several editions. This section summarizes the main features of the Enterprise and Assurance editions, as well as the graphical Integrated Development Environments (IDEs) that accompany GNAT Pro.

4.4.1 GNAT Pro Enterprise

GNAT Pro Enterprise is a development environment for producing critical software systems where reliability, efficiency, and maintainability are essential. It is available for Ada, C, and C++.

Based on the GNU GCC technology, the GNAT Pro Enterprise product line supports all versions of the Ada language, from Ada 83 to Ada 2012, as well as features of the in-progress Ada 202x standard. Other editions of the GNAT Pro product handle multiple versions of C (from C89 through C18) and C++ (from C++98 through C++17). GNAT Pro Ada includes an Integrated Development Environment (GNAT Studio and/or GNATbench), a comprehensive toolsuite including a visual debugger, and a set of libraries and bindings.

GNAT Pro Enterprise offers several features that make it especially appropriate for the development of high-assurance FACE conformant systems:

Run-Time Library Options

GNAT Pro Ada Enterprise includes a variety of choices for the run-time library, based on the target platform. In addition to the full run-time, which is available for all platforms, the product on some platforms also includes restricted libraries that reduce the footprint and help simplify safety certification:

- The ZFP library ("Zero FootPrint") offers a minimal application footprint (rivalling that of C) while retaining compatibility with the SPARK subset and verification tools;
- The Cert library is a small-footprint runtime that implements basic exception handling and dynamic storage management for sequential applications;

- The Ravenscar-Cert library augments the Cert library with support for the Ravenscar tasking profile.

Especially relevant to FACE UoC developers:

- The Cert library implements both the Safety-Base / Security and the Safety-Extended capability sets for Ada 95 and Ada 2012 when POSIX pthreads are used rather than Ada tasking;
- Ravenscar-Cert implements these capability sets for applications that use Ada tasking; and
- The full Ada run-time implements the General-Purpose capability set.

AdaCore can supply evidence (results of running the ACATS tests) to demonstrate that the functionality required by the FACE Technical Standard is implemented.

Enhanced Data Validity Checking

Improper or absent data validity checking is a notorious source of security vulnerabilities in software systems. Ada has always offered range checks for scalar subtypes, but GNAT Pro goes further, offering enhanced validity checking that can protect a program against malicious or accidental memory corruption, failed I/O devices, and so on. This feature is particularly useful in combination with automatic *Fuzz Testing*, since it offers strong defense for invalid data *at the software boundary* of a system.

4.4.2 GNAT Pro Assurance

GNAT Pro Assurance extends GNAT Pro Enterprise with specialized support, such as bug fixes and "known problems" analyses, on a specific version of the toolchain. This product edition is especially suitable for applications with long-lived maintenance cycles or assurance requirements, since critical updates to the compiler or other product components may become necessary years after the initial release.

4.4.2.1 Sustained Branches

Unique to GNAT Pro Assurance is a service known as a "sustained branch": customized support and maintenance for a specific version of the product. A project on a sustained branch can monitor relevant known problems, analyze their impact, and if needed update to a newer version of the product on the same development branch (i.e., not incorporating changes introduced in later versions of the product).

Sustained branches are a practical solution to the problem of ensuring toolchain stability while allowing flexibility in case an upgrade is needed to correct a critical problem.

4.4.2.2 Source to Object Traceability

Source-to-object traceability is required in standards such as DO-178C, and a GNAT Pro compiler option can limit the use of language constructs that generate object code that is not directly traceable to the source code. As an add-on service, AdaCore can perform an analysis that demonstrates this traceability and justifies any remaining cases of non-traceable code.

4.4.3 GNAT Pro Integrated Development Environments (IDEs)

GNAT Pro includes several graphical IDEs for invoking the build tools and accompanying utilities and monitoring their outputs.

4.4.3.1 GNAT Studio

GNAT Studio (formerly named “GNAT Programming Studio” or “GPS”) is a powerful and simple-to-use IDE that streamlines software development from the initial coding stage through testing, debugging, system integration, and maintenance. GNAT Studio is designed to allow programmers to exploit the full capabilities of the GNAT Pro technology.

Tools

GNAT Studio’s extensive navigation and analysis tools can generate a variety of useful information including call graphs, source dependencies, project organization, and complexity metrics, giving the developer a thorough understanding of a program at multiple levels. It allows interfacing with third-party Version Control Systems, easing both development and maintenance.

Robust, Flexible and Extensible

Especially suited for large, complex systems, GNAT Studio can import existing projects from other Ada implementations while adhering to their file naming conventions and retaining the existing directory organization. Through the IDE’s multi-language capabilities, components written in C and C++ can also be handled. GNAT Studio is highly extensible; additional tools can be plugged in through a simple scripting approach. It is also tailorable, allowing various aspects of the program’s appearance to be customized in the editor.

Easy to Learn, Easy to Use

GNAT Studio is intuitive to new users, thanks to its menu-driven interface with extensive online help (including documentation of all the menu selections) and “tool tips”. The Project Wizard makes it simple to get started, supplying default values for almost all of the project properties. For experienced users, GNAT Studio offers the necessary level of control for advanced purposes; e.g., the ability to run command scripts. Anything that can be done on the command line is achievable through the menu interface.

Remote Programming

Integrated into GNAT Studio, Remote Programming provides a secure and efficient way for programmers to access any number of remote servers on a wide variety of platforms while taking advantage of the power and familiarity of their local laptop computers or workstations.

Support for Microsoft’s Language Server Protocol

GNAT Studio’s source navigation engine is implemented through support for Microsoft’s Language Server Protocol (LSP), and it includes a server for this protocol for the Ada and SPARK languages. A language server based on the LSP encapsulates the language-specific knowledge that clients (such as editing tools) can access via standard requests and through inter-process communication. An IDE that supports LSP can handle any language for which a language server is implemented, and in the other direction a language server can be reused in any IDE that supports LSP, such as Visual Studio Code.

4.4.3.2 GNATbench - GNATbench

GNATbench is an Ada development plug-in for Eclipse and Wind River’s Workbench environment. The Workbench integration supports Ada development on a variety of VxWorks real-time operating systems.

The Eclipse version is primarily for native applications, with some support for cross development. In both cases the Ada tools are tightly integrated.

GNATdashboard

GNATdashboard serves as a one-stop control panel for monitoring and improving the quality of Ada software. It integrates and aggregates the results of AdaCore's various static and dynamic analysis tools (GNATmetric, GNATcheck, GNATcoverage, CodePeer, and SPARK Pro, among others) within a common interface, helping quality assurance managers and project leaders understand or reduce their software's technical debt, and eliminating the need for manual input.

GNATdashboard fits naturally into a continuous integration environment, providing users with metrics on code complexity, code coverage, conformance to coding standards, and more. A FACE UoC developer can use GNATdashboard with GNATcheck, to monitor progress on meeting the Ada capability set constraints.

4.4.4 GNAT Pro Support for FACE Conformance

GNAT Pro Ada directly helps UoC developers demonstrate conformance with the FACE requirements. Support comes in several areas:

- Detection of features outside the targeted profile / capability set, via compiler-enforced standard pragmas and a GNAT Pro static analysis tool
  - `pragma Restrictions`
    This pragma, described earlier (see page 25), identifies language features that are to be prohibited in the program.
  - `pragma Profile (Ravenscar)`
    This pragma, introduced in Ada 2005, is equivalent to a collection of Restrictions pragmas that define the Ravenscar rules.
- Detection of features outside the targeted profile / capability set, via the GNATcheck static analysis tool (see page 35) supplied with GNAT Pro Ada
- Specialized run-time libraries

  The Cert and Ravenscar-Cert libraries (see page 32) implement the safety capability sets.

4.5 GNAT Pro Ada Tools for Static Analysis

This section describes a number of GNAT Pro Ada tools that perform static analysis of Ada source code. These tools help in general during the verification process, and, as will be noted below, some are particularly useful in demonstrating FACE conformance.

4.5.1 GNATcheck

GNATcheck is a coding standard verification tool that is extensible and rule-based. It allows developers to completely define a coding standard as a set of rules, for example a subset of permitted language features. It checks whether a source program satisfies the resulting rules and thereby facilitates demonstration of a system's conformance with certification standards.

FACE UoC developers can use GNATcheck as a key part of conformance verification, to help demonstrate that their Ada code meets the restrictions imposed by the applicable Capability Set (General-Purpose, Safety-Extended, Safety-Base & Security), for both Ada 95 and Ada 2012. Automating this step through GNATcheck is significantly more efficient and reliable than through manual code inspection.

Key features include:

- An integrated Ada Restrictions mechanism for banning specific features from an application. This can be used to restrict features such as tasking, exceptions, dynamic allocation, fixed- or floating point, input/output, and unchecked conversions.
- Restrictions specific to GNAT Pro, such as banning features that result in the generation of implicit loops or conditionals in the object code, or in the generation of elaboration code.
- Additional Ada semantic rules requested by customers, such as enforcing a specific ordering of parameters, normalizing entity names, and prohibiting subprograms from having multiple returns.
- User-friendly interface for creating and using a complete coding standard.
- Generation of project-wide reports, including evidence of the level of conformance with a given coding standard.
- Over 30 compile-time warnings from GNAT Pro that detect typical error situations, such as local variables being used before being initialized, incorrect assumptions about array lower bounds, infinite recursion, incorrect data alignment, and accidental hiding of names.
- Style checks that allow developers to control indentation, casing, comment style, and nesting level.

4.5.2 GNATmetric

GNATmetric is a static analysis tool that calculates a set of commonly used industry metrics, thus allowing developers to estimate code complexity and better understand the structure of the source program. This information also facilitates satisfying the requirements of certain software development frameworks and is useful in conjunction with GNATcheck (for example in reporting and limiting the maximum subprogram nesting depth).

4.5.3 GNATstack

GNATstack is a software analysis tool that enables Ada/C software development teams to accurately estimate the maximum size of the memory stack required for program execution. Although FACE conformance does not require such an analysis, GNATstack will be useful to FACE UoC developers since a stack overflow in a UoC for high-DAL software could lead to a catastrophic failure.

The GNATstack tool statically computes the maximum stack space required by each task in an application. The reported bounds can be used to reserve sufficient space, resulting in safe execution with respect to stack usage. The tool uses a conservative analysis based on compile-time analysis,

augmented with user-supplied input to deal with complexities such as subprogram recursion, while avoiding unnecessarily pessimistic estimates.

GNATstack exploits data generated by the compiler to compute worst-case stack requirements. It performs per-subprogram stack usage computation combined with control flow analysis.

GNATstack can analyze object-oriented applications, automatically determining maximum stack usage on code that uses dynamic dispatching in Ada. A dispatching call challenges static analysis because the identity of the subprogram being invoked is not known until run time. GNATstack solves this problem by statically determining the subset of potential targets (primitive operations) for every dispatching call. This significantly reduces the analysis effort and yields precise stack usage bounds on complex Ada code.

GNATstack's main output is the worst-case stack usage for every entry point, together with the paths that result in these stack sizes. The list of entry points can be automatically computed (all the tasks, including the environment task) or can be specified by the user (a list of entry points or all the subprograms matching a given regular expression).

GNATstack can also detect and display a list of potential problems when computing stack requirements:

- Indirect (including dispatching) calls. The tool will indicate the number of indirect calls made from any subprogram.
- External calls. The tool displays all the subprograms that are reachable from any entry point for which there is no stack or call graph information.
- Unbounded frames. The tool displays all the subprograms that are reachable from any entry point with an unbounded stack requirement. The required stack size depends on the arguments passed to the subprogram. For example:

```
procedure P (N : Integer) is
   S : String (1 .. N);
begin
   ...
end P;
```

- Cycles. The tool can detect all the cycles (i.e., potential recursion) in the call graph.

GNATstack allows the user to supply a text file with the missing information, such as the potential targets for indirect calls, the stack requirements for externals calls, and the maximal size for unbounded frames.

4.5.4 Time and Space Analysis

4.5.4.1 Timing Verification

Suitably subsetted, Ada (and SPARK) are amenable to the static analysis of timing behavior. This kind of analysis is relevant for real-time systems, where *worst-case execution time (WCET)* must be known in order to guarantee that timing deadlines will always be met. Timing analysis is also of interest for secure systems, where the issue might be to show that programs do not leak information via so-called *side-channels* based on the observation of *differences* in execution time.

AdaCore does not produce its own WCET tool, but there are several such tools on the market from partner companies, such as RapiTime from Rapita Systems Ltd.

4.5.4.2 Memory Usage Verification

Ada and SPARK can support the static analysis of worst-case memory consumption, so that a developer can show that a program will *never* run out of memory at execution time.

In both SPARK and Ada, users can specify pragma Restrictions with the standard arguments No_Allocators and No_Implicit_Heap_Allocations. This will completely prevent heap usage, thus reducing memory usage analysis to a worst-case computation of stack usage for each task in a system. Stack size analysis is implemented directly in AdaCore's GNATstack tool, as described above.

4.5.5 Semantic Analysis Tools—Libadalang

Libadalang is a reusable library that forms a high-performance semantic processing and transformation engine for Ada source code, with an API in Python as well as Ada. It is particularly suitable for writing *lightweight* and *project-specific* static analysis tools. Typical libadalang applications include:

- Static analysis (property verification)
- Code instrumentation
- Design documentation tools
- Metric testing or timing tools
- Dependency tree analysis tools
- Type dictionary generators
- Coding standard enforcement tools
- Language translators (e.g., to CORBA IDL)
- Quality assessment tools
- Source browsers and formatters
- Syntax directed editors

4.6 Static Verification: CodePeer

CodePeer is an Ada source code analyzer that detects run-time and logic errors that can cause safety and security vulnerabilities in a code base. CodePeer assesses potential bugs before program execution, serving as an automated peer reviewer. It can be used on existing codebases, thereby helping vulnerability analysis during a security assessment or system modernization, and when performing impact analysis when introducing changes. It can also be used on new projects, helping to find errors

efficiently and early in the development life-cycle. Using control-flow, data-flow, and other advanced static analysis techniques, CodePeer detects errors that would otherwise only be found through labor-intensive debugging.

CodePeer can be used from within the GNAT Pro development environment, or as part of a continuous integration regime. As a stand-alone tool, CodePeer can also be used with projects that do not use GNAT Pro for compilation.

4.6.1 Early Error Detection

CodePeer's advanced static error detection finds bugs in code by analyzing every line of code, considering every possible input and every path through the program. CodePeer can be used very early in the development life cycle, to identify problems when defects are much less costly to repair. It can also be used retrospectively on existing code bases, to detect latent vulnerabilities. For a FACE UoC developer, the main benefit from CodePeer will come from the tool's error detection capabilities.

4.6.2 CWE Compatibility

CodePeer can detect a number of "Dangerous Software Errors" in the MITRE Corporation's Common Weakness Enumeration, and the tool has been certified by The MITRE Corporation as a "CWE-Compatible" product [Mi 20xx].

The following table lists the weaknesses detected by CodePeer:

CWE weakness	Description
CWE 120, 124, 125, 126, 127, 129, 130, 131	Buffer overflow/underflow
CWE 136, 137	Variant record field violation, Use of incorrect type in inheritance hierarchy
CWE 190, 191	Numeric overflow/underflow
CWE 362, 366	Race condition
CWE 369	Division by zero
CWE 457	Use of uninitialized variable
CWE 476	Null pointer dereference
CWE 561	Dead (unreachable) code
CWE 563	Unused or redundant assignment
CWE 570	Expression is always false
CWE 571	Expression is always true
CWE 628	Incorrect arguments in call
CWE 667	Improper locking
CWE 682	Incorrect calculation

CWE weakness	Description
CWE 820	Missing synchronization
CWE 821	Incorrect synchronization
CWE 835	Infinite loop

4.7 Dynamic Analysis Tools

Although they do not play a direct role in demonstrating conformance with the FACE requirements, dynamic analysis tools are useful in general as part of the verification process. Their output can serve as evidence towards certification objectives in software standards such as DO-178C.

4.7.1 GNATtest

The GNATtest tool helps create and maintain a complete unit testing infrastructure for projects of any size / complexity. It is based on the concept that each visible subprogram should have at least one corresponding unit test. GNATtest produces two outputs:

- The complete harnessing code for executing all the unit tests under consideration. This code is generated completely automatically.
- A set of separate test stubs for each subprogram to be tested. These test stubs are to be completed by the user.

GNATtest handles Ada's Object-Oriented Programming features and can help verify tagged type substitutability (the Liskov Substitution Principle), or "LSP", which can be used to demonstrate consistency of class hierarchies.

4.7.2 GNATemulator

GNATemulator is an efficient and flexible tool that provides integrated, lightweight target emulation.

Based on the QEMU technology, a generic and open-source machine emulator and virtualizer, GNATemulator allows software developers to compile code directly for their target architecture and run it on their host platform, through an approach that translates from the target object code to native instructions on the host. This avoids the inconvenience and cost of managing an actual board, while offering an efficient testing environment compatible with the final hardware.

GNATemulator does not attempt to be a complete time-accurate target board simulator, and thus it cannot be used for all aspects of testing. But it does provide a very efficient and cost-effective way to execute the target code very early in the development and verification processes. GNATemulator thus offers a practical compromise between a native environment that lacks target emulation capability, and a cross configuration where the final target hardware might not be available soon enough or in sufficient quantity.

4.7.3 GNATcoverage

GNATcoverage is a dynamic analysis tool that analyzes and reports program coverage. It computes its results from trace files that show which program constructs have been exercised by a given test campaign. With source code instrumentation, the tool produces these files by executing an alternative version of the program, built from source code instrumented to populate coverage-related data structures. Through an option to GNATcoverage, the user can specify the granularity of the analysis by choosing any of the coverage criteria defined in DO-178C: statement coverage, decision coverage, or Modified Condition / Decision Coverage (MC/DC).

Source-based instrumentation brings several major benefits: efficiency of tool execution (much faster than alternative coverage strategies using binary traces and target emulation, especially on native platforms), compact-size source trace files independent of execution duration, and support for coverage of shared libraries.

4.8 Support and Expertise

Every AdaCore product subscription comes with front-line support provided directly by the product developers themselves, who have deep expertise in the Ada language, software certification standards in several domains, compilation technologies, embedded system technology, and static and dynamic verification. They have extensive experience supporting customers in domains such as commercial and military avionics, railway, space, energy, and air traffic management/control. Customers' questions (requests for guidance on feature usage, suggestions for technology enhancements, or defect reports) are handled efficiently and effectively.

Beyond this bundled support, AdaCore also provides Ada language and tool training as well as on-site consulting on topics such as how to best deploy the technology, and mentoring assistance on start-up issues. On-demand tool development or ports to new platforms are also available.

5 Abbreviations

Abbreviation	Expansion
API	Application Program Interface
COE	Common Operating Environment
COTS	Commercial Off-The Shelf
CTS	Conformance Test Suite
CVM	Conformance Verification Matrix
DAL	Design Assurance Level
DoD	Department of Defense
DSDM	Domain-Specific Data Model
FACE	Future Airborne Capability Environment
GCC	GNU Compiler Collection
GUI	Graphical User Interface
IDE	Integrated Development Environment
IDL	Interface Definition Language
IOSS	Input-Output Services Segment
ISO	International Organization for Standardization
LSP	Language Server Protocol, Liskov Substitution Principle
OSS	Operating System Segment
PCS	Portable Components Segment
PSSS	Platform-Specific Services Segment
RTOS	Real-Time Operating Systems
SDM	Shared Data Model
TQL	Tool Qualification Level
TSS	Transport Services Segment
UoC	Unit of Conformance
UoP	Unit of Portability
USM	UoP-Supplied Model
VA	Verification Authority
WCET	Worst-Case Execution Time

6 References

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

6.1 Cited References

[AA 2021] AdaCore and Altran UK Ltd, *SPARK Reference Manual*, 2021
https://docs.adacore.com/live/wave/spark2014/html/spark2014_rm/index.html

[ACAA 2016] Ada Conformance Assessment Authority, *Consolidated Ada 2012 Language Reference Manual*;
http://www.ada-auth.org/standards/ada12_w_tc1.html

[Ad 2016] AdaCore, *High Integrity Object-Oriented Programming in Ada*, Version 1.4, 2016;
https://www.adacore.com/papers/high-integrity-oop-in-ada

[ARINC 2019] ARINC Industry Activities, *ARINC 653; Avionics Application Software Standard Interface, Parts 0, 1, 2, 3A, 4 and 5.*
https://www.aviation-ia.com/product-categories/600-series

[AT 2020] AdaCore and Thales, *Implementation Guidance for the Adoption of SPARK*, Release 1.2, July 24, 2020;
https://www.adacore.com/books/implementation-guidance-spark

[Ba 2014] John Barnes, *Programming in Ada 2012*, Cambridge University Press, 2014.

[Ba 2015] John Barnes, *Safe and Secure Software: An Invitation to Ada 2012*, AdaCore, 2015;
https://www.adacore.com/books/safe-and-secure-software

[Br 2021] Benjamin M. Brosgol, "Making Software FACE™ Conformant and Fully Portable: Coding Guidance for Ada", in *Military Embedded Systems*, March 2021

[DB 2001] Brian Dobbing and Alan Burns, *The Ravenscar Tasking Profile for High Integrity Real-Time Programs*.
http://www.sigada.org/conf/sigada2001/private/SIGAda2001-CDROM/SIGAda1998-Proceedings/dobbing.pdf

[DoD 2014] US Department of Defense, *Airworthiness Certification Criteria*, Department of Defense Handbook, MIL-HDBK-516C, 12 December 2014;
https://daytonaero.com/wp-content/uploads/MIL-HDBK-516C-from-ASSIST.pdf

[IEEE 2017] IEEE/Open Group 1003.1-2017 – IEEE Standard for Information Technology—*Portable Operating System Interface (POSIX™) Base Specifications*, Issue 7, 2017.
http://pubs.opengroup.org/onlinepubs/9699919799/
https://standards.ieee.org/findstds/standard/1003.1-2017.html

[MC 2015] John W. McCormick and Peter C. Chapin, *Building High Integrity Applications with SPARK*, Cambridge University Press, 2015

[Mi 20xx] The MITRE Corp., *CWE-Compatible Products and Services*, undated; https://cwe.mitre.org/compatible/compatible.html

[OG 2016] The Open Group, *FACE Conformance Certification Guide,* Version 1.0, August 2016; https://www.opengroup.org/face/docsandtools

[OG 2020] The Open Group, *FACE Technical Standard*, Edition 3.1, July 2020; https://www.opengroup.org/face/docsandtools

[OG 20xx] The Open Group, *FACE Conformance Verification Matrix*, Edition 3.0, undated; https://www.opengroup.org/face/docsandtools

[OMG 2018] Object Management Group, *Interface Definition Language (IDL)*, Version 4.2, March 2018; https://www.omg.org/spec/IDL

[RTCA 2011] RTCA SC-205 / EUROCAE WG-12, *Software Considerations in Airborne Systems and Equipment Certification*, DO-178C / ED-12C; 13 December 2011; https://my.rtca.org/NC__Product?id=a1B36000001IcjlEAC

6.2 FACE Related Articles by AdaCore Authors

Benjamin M. Brosgol and Dudrey Smith, “Towards Safety and Security in FACE™ Components: High Assurance with Portability”, in *Military Embedded Systems*, March 2018; https://mil-embedded.com/articles/toward-components-high-assurance-portability/

Benjamin M. Brosgol, Patrick Rogers and Dudrey Smith, “Ada Language Run-Times and the FACE™ Technical Standard: Achieving Application Portability and Reliability”, *Proceedings of the U.S. Army FACE™ Technical Interchange Meeting*, Huntsville AL, September 2018

Benjamin M. Brosgol, Patrick Rogers and Dudrey Smith, “Portable, Reliable and Efficient Concurrency: Ravenscar Ada Tasking and the FACE™ Safety Profiles”, in *Military Embedded Systems*, November-December 2018

Benjamin M. Brosgol, “Verifying High Assurance FACE Components with Ada and SPARK: Combining Formal Methods and Testing “, *Proceedings of the U.S. Air Force FACE™ Technical Interchange Meeting*, Dayton OH, September 2019

Benjamin M. Brosgol, “DO-178C Meets the FACE™ Technical Standard: High-Assurance and Reusability for Airborne Software”, in *Military Embedded Systems*, March 2020

Benjamin M. Brosgol, “Making Software FACE™ Conformant and Fully Portable: Coding Guidance for Ada”, in *Military Embedded Systems*, March 2021 (referenced as [Br 2021] earlier)

Index

ACATS Test Suite 34
Ada capability sets 14, 26
Ada language 17
  Abstract data types 21
  `Assertion_Error` exception 20
  Buffer overrun prevention 22
  Child units 21
  Concurrent programming (tasks) 22
  Contract-based programming 14, 19, 20, 23
  Generic templates 21
  History and overview 18
  Object-Oriented Programming (OOP) 21
  Postconditions 20
  `pragma Profile (Ravenscar)` 37
  `pragma Restrictions` 23, 26, 36, 39
  Preconditions 20
  Programming in the large 21
  Real-Time Systems Annex 22
  Scalar ranges 19
  Systems Programming Annex 22
  Usage 19
AdaCore
  CodePeer ..... See CodePeer
  GNAT Pro Assurance ..... See GNAT Pro Assurance
  GNAT Pro Enterprise ..... *See* GNAT Pro Enterprise
  GNAT Studio ..... See GNAT Studio
  GNATbench ..... *See* GNATbench
  GNATcheck ..... *See* GNATcheck
  GNATcoverage ..... *See* GNATcoverage
  GNATdashboard ..... See GNATdashboard
  GNATemulator ..... *See* GNAtemulator
  GNATmetric ..... *See* GNATmetric
  GNATprove ..... *See* GNATprove
  GNATstack ..... *See* GNATstack
  GNATtest ..... *See* GNATtest
  GPS ..... *See* GNAT Studio
  QGen ..... *See* QGen
  SPARK Pro ..... *See* SPARK Pro
  Support and expertise 43
Altran 23
ARINC 653 ..... 7, 12, 13

Babbage, Charles 19
Bottom-side interface 12
Buffer overrun 17, 22
Byron, (Lord) George Gordon 19

C language 11, 17, 19
  Buffer overrun 23
  Interfacing from Ada 18
C# language 19
C++ language 11, 17, 19, 21
  Buffer overrun 23
  Interfacing from Ada 18
Capability sets 12, 13, 25, 26, 32
Cert run-time library 34, 37
CodePeer 30, 40
  Common Weakness Enumeration (CWE) errors detected 32, 41
  Early error detection 40
Coding standard
  Enforcement by GNATcheck 37
Common Criteria 22
Common Weakness Enumeration (CWE) errors detected by CodePeer 41
Common Weakness Enumeration (CWE) errors detected by SPARK Pro 32
Conformance Verification Matrix (CVM) 14
CWE compatibility 32, 41
CWE-compatible tool
  CodePeer iv
  SPARK Pro iv

Data Validation 34
Dependency injection (software design pattern) 12
DO-178C ..... iv, 7, 22, 30, 35
Domain-Specific Data Model (DSDM) 11

Eclipse support ..... See GNATbench
EN 50128 ..... 22, 30
Encapsulation (software engineering principle) 9, 21

FACE Conformance 14
  Support by GNAT Pro 36
  Support by SPARK Pro 32
FACE Conformance Certification Guide 15
FACE Conformance Test Suite (CTS) 15
FACE Conformance Verification 14
FACE Consortium 7
FACE Data Model Language 11
FACE Reference Architecture 9
FACE Technical Standard 9, 12, 14, 17
Fuzz Testing 34

General-Purpose capability set 14
General-Purpose Profile 13
GNAT Pro Ada 33
GNAT Pro Assurance
  Sustained branch ..... See Sustained branch
  Traceability analysis service .. See Traceability (Source to Object)
GNAT Pro Enterprise
  Run-Time Library options 34
GNAT Studio 33, 35
GNATbench 33, 36
GNATcheck ..... iv, 26, 37
  Coding standard enforcement 37
GNATcoverage 30, 42
GNATdashboard 36
GNATemulator 42
GNATmetric 38
GNATprove 23, 31
GNATstack 38
GNATtest 41
GNU GCC technology 33

Hybrid verification 27

Ichbiah, Jean 18
IDL 7
Injectable interface 11
Input Data
  validation 34
Input/Output Services Segment (IOSS) 10
Inria 23
Integer overflow vulnerability 17
Integrated Development Environments (IDEs) 35
ISO 26262 30

Java language 11, 17, 19, 21

Language Server Protocol (LSP) 36
Libadalang 40
Liskov Substitution Principle (LSP) 42
Lovelace, Augusta Ada 19
Lynx Software Technologies
  LynxOS-178 RTOS iv

Memory
  Deallocation 14
  Dynamic allocation 14
  Garbage collection 17, 21
  Storage leakage 39
  Usage verification 39
MIL-HDBK-516C 7, 30
MISRA C (generated by QGen) 30
Model-Based Engineering 30

Open Universal Domain Description Language (Open UDDL) 11
Operating System Segment (OSS) 10, 12

Partitioning (in OSS profiles) 13
Platform-Specific Services Segment (PSSS) 10
Portability 18
Portable Components Segment (PCS) 11
POSIX 7, 12, 13
Profile 12, 13

QEMU 42
QGen 29, 30

Rapita Systems Ltd. 39
Ravenscar tasking profile 18, 23, 25, 34
Ravenscar-Cert run-time library 34, 37
Run-time library 12

Safety Base Sub-profile 13
Safety-Base & Security capability set 14
Safety-Extended capability set 14
Safety-Extended Sub-profile 13
Security Profile 13
Shared Data Model (SDM) 11
Simulink® 30
Software crisis 7
SPARK 23
  Absence of run-time exceptions 31
  Data and control flow analysis 31
  Generated by QGen 30
  Information flow analysis 31
  Static verification support 31
  Usage 23
SPARK adoption
  Bronze level 26
  Gold level 27
  Platinum level 27
  Silver level 27
  Stone level 26
SPARK Pro 23, 29, 31
Specialized needs annexes (in Ada standard) 18
Stateflow® 30
Static Verification
  Soundness 24
Storage leakage 14
Sustained branch 34

Taft, Tucker 18
Timing Verification 39
Tool Qualification Level (TQL) 30
Traceability (Source to Object) 35
Transport Services Segment (TSS) 11

Unit of Conformance (UoC) 7
Unit of Portability (UoP) 11
UoP Supplied Model (USM) 11
Vulnerability avoidance (through language choice) 17

WCET Analysis 39
Wind River
  VxWorks 653 RTOS iv
Workbench (WindRiver development environment) 36

Zero Footprint (ZFP) run-time library 32, 34