AdaCore Technologies for

SPACE SYSTEMS SOFTWARE

v 1.0

Supporting qualification for ECSS-E-ST-40C and ECSS-Q-ST-80C

Benjamin M. Brosgol &
Jean-Paul Blanquart

AdaCore Technologies for

Space Systems Software

Supporting Qualification for ECSS-E-ST-40C and ECSS-Q-ST-80C

Benjamin M. Brosgol &
Jean-Paul Blanquart

Version 1.0

November 2021

About the Authors

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. He was a Distinguished Reviewer during the original Ada development, a member of the language design team in the Ada 95 revision, and a member of the Expert Group for the Real-Time Specification for Java under the Java Community Process. He has published dozens of journal articles and delivered conference presentations on topics including the avionics software standard DO-178C, the Ada and SPARK languages, real-time software technologies, object-oriented methodologies, and the FACE™ (Future Airborne Capabilities Environment) approach to software portability.

Jean-Paul Blanquart

Dr. Blanquart is a recognized authority on computer-based systems safety and dependability, with a decades-long career that spans academic research (LAAS-CNRS, Toulouse, France) and the space industry (Airbus Defence and Space). He was a member of the ECSS Working Groups in charge of revision 1 of ECSS-Q-ST-80C and ECSS-Q-HB-80-03A (and also the dependability and safety standards ECSS-Q-ST-30C and ECSS-Q-ST-40C). He has been an active member of a French cross-domain Working Group on safety and safety standards since its creation in 2010. This Working Group gathers industrial safety experts and related tool providers from domains that include automotive, aviation, defense, nuclear, industrial processes, railway and space.

Foreword

Software development presents daunting challenges when the resulting system needs to operate reliably, safely, and securely while meeting hard real-time deadlines on a memory-limited target platform. Correct program execution can literally be a matter of life and death, but such is the reality facing developers of space software systems. A project's ability to produce high-assurance software in a cost-effective manner depends on two factors:

"Failure is not an option"
Gene Kranz (NASA) in the film *Apollo 13*

- Effective processes for managing the software and system life cycles, with well-defined activities for planning, controlling, and monitoring the work; and
- Effective technologies (programming languages, development and verification tools, version control systems, etc.) to support the software life cycle processes.

For safety-critical application domains, the first factor is typically anticipated by a regulatory authority in the form of certification / qualification standards. The ECSS software-related standards serve as an example, with the current set of documents based on decades of experience with space system development. In particular, the software engineering standard ECSS-E-ST-40C and the software product assurance standard ECSS-Q-ST-80C provide a framework in which software suppliers and customers can interact, with a clear statement and understanding of processes and responsibilities.

Technologies, and more specifically the choice of programming language(s) and supporting toolsuites, directly affect the ease or difficulty of developing, verifying, and maintaining quality software. The state of the art in software engineering has made large strides over the years, with programming language / methodology advances in areas such as modularization and encapsulation. Nevertheless, the key messages have stayed constant:

- The programming language should help prevent errors from being introduced in the first place; and
- If errors are present, the language rules should detect them early in the software life cycle, when defects are easiest and least expensive to correct.

These messages come through clearly in the Ada programming language, which was designed from the start to enforce sound software engineering principles, catching errors early and avoiding pitfalls such as buffer overrun that arise in other languages. Ada has evolved considerably since it first emerged in the mid-1980s, for example adding Object-Oriented Programming support in Ada 95, but each new version has kept true to the original design philosophy.

AdaCore's Ada-based tools have been helping developers design, develop and maintain high-assurance software for over 25 years, in domains that include space systems, commercial and military avionics, air traffic control, train systems, automotive, and medical devices. This book summarizes AdaCore's language and tool technologies and shows how they can help space software suppliers meet the requirements in ECSS-E-ST-40C and ECSS-Q-ST-80C. With effective processes as established by these standards, and effective technologies as supplied by AdaCore, software suppliers will be well equipped to meet the challenges of space software development.

Benjamin M. Brosgol
AdaCore
Bedford, Massachusetts USA
November 2021

Jean-Paul Blanquart
Airbus Defence and Space
Toulouse, France
November 2021

info@adacore.com
www.adacore.com

Table of Contents

About the Authors iii

Foreword v

Table of Contents vii

1 Introduction 11

1.1 ECSS-E-ST-40C: Space engineering / Software 13

1.2 ECSS-Q-ST-80C: Space product assurance / Software product assurance 22

1.3 ECSS Handbooks 24

2 Programming Languages for Space Software 27

2.1 Ada 27

2.1.1 Ada language overview 28
2.1.2 Ada language background 30
2.1.3 Scalar ranges 31
2.1.4 Contract-based programming 32
2.1.5 Programming in the large 34
2.1.6 Generic templates 34
2.1.7 Object-Oriented Programming (OOP) 35
2.1.8 Concurrent programming 36
2.1.9 Systems programming 36
2.1.10 Real-time programming 36
2.1.11 High-integrity systems 37
2.1.12 Enforcing a coding standard 38
2.1.13 Ada and the ECSS Standards 39

2.2 SPARK 40

2.2.1 SPARK Basics 40
2.2.2 Ease of Adoption: Levels of Adoption of Formal Methods 44

2.2.3 Hybrid Verification....46
2.2.4 SPARK and the ECSS Standards....48

3 Tools for Space Software Development....49

3.1 AdaCore Tools and the Software Life Cycle....49

3.2 QGen Toolsuite for Model-Based Engineering....51

3.2.1 QGen Capabilities....52
3.2.2 QGen and the ECSS standards....53

3.3 Static Verification: SPARK Pro....53

3.3.1 Powerful Static Verification....54
3.3.2 Minimal Run-Time Footprint....55
3.3.3 CWE Compatibility....55
3.3.4 SPARK Pro and the ECSS Standards....57

3.4 GNAT Pro Ada Development Environments....58

3.4.1 GNAT Pro Enterprise....59
3.4.2 GNAT Pro Assurance....60
3.4.3 GNAT Pro Integrated Development Environments (IDEs)....61
3.4.4 GNAT Pro and the ECSS Standards....64

3.5 GNAT Pro Ada Tools for Static Analysis and Target Emulation 66

3.5.1 GNATcheck....66
3.5.2 GNATmetric....67
3.5.3 GNATstack....68
3.5.4 Time and Space Analysis....70
3.5.5 Semantic Analysis Tools—Libadalang....71
3.5.6 GNATemulator....71
3.5.7 GNAT Pro Ada Tools and the ECSS Standards....72

3.6 Static Verification: CodePeer....73

3.6.1 Early Error Detection....73
3.6.2 CWE Compatibility....74
3.6.3 CodePeer and the ECSS Standards....75

3.7 GNAT Dynamic Analysis Suite ... 76

3.7.1 GNATtest ... 76
3.7.2 GNATcoverage ... 77
3.7.3 GNAT Dynamic Analysis Suite and the ECSS Standards.. 77

3.8 Support and Expertise ... 78

4 Compliance with ECSS-E-ST-40C ... 81

4.1 Software requirements and architecture engineering process {§5.4} ... 81

4.1.1 Software architecture design {§5.4.3} ... 81

4.2 Software design and implementation engineering process {§5.5} ... 85

4.2.1 Design of software items {§5.5.2} ... 85
4.2.2 Coding and testing {§5.5.3} ... 88
4.2.3 Integration {§5.5.4} ... 89

4.3 Software validation process {§5.6} ... 90

4.3.1 Validation activities with respect to the technical specification {§5.6.3} ... 90
4.3.2 Validation activities with respect to the requirements baseline {§5.6.4} ... 90

4.4 Software delivery and acceptance process {§5.7} ... 91

4.4.1 Software acceptance {§5.7.3} ... 91

4.5 Software verification process {§5.8} ... 92

4.5.1 Verification activities {§5.8.3} ... 92

4.6 Software operation process {§5.9} ... 96

4.6.1 Process implementation {§5.9.2} ... 96
4.6.2 Software operation support {§5.9.4} ... 97

4.7 Software maintenance process {§5.10} ... 97

4.7.1 Process implementation {§5.10.2} ... 97

4.7.2 Modification implementation {§5.10.4}..........................98

5 Compliance with ECSS-Q-ST-80C.. 99

5.1 Software product assurance programme implementation {§5} 99

5.1.1 Software product assurance programme management {§5.2} 99

5.1.2 Tools and supporting environment {§5.6}.......................99

5.2 Software process assurance {§6}... 101

5.2.1 Requirements applicable to all software engineering processes {§6.2} 101

5.2.2 Requirements applicable to individual software engineering processes or activities {§6.3}....................................103

5.3 Software product quality assurance {§7}.............................. 107

5.3.1 Product quality objectives and metrication {§7.1}........107

5.3.2 Product quality requirements {§7.2}............................107

6 Abbreviations 109

7 References... 111

Index 117

1 Introduction

Software for space applications must meet unique and formidable requirements. Hard real-time deadlines, a constrained target execution environment with limited storage capacity, and distributed functionality between ground and on-board systems are some of the challenges, with little margin for error. The software needs to work correctly from the outset, without safety or security defects, and the source code needs to be amenable to maintenance over the system's lifetime (which may extend over decades) as requirements evolve.

To provide a common approach to addressing these challenges, the European Cooperation for Space Standardization (ECSS) was formed in the mid-1990s in a joint effort conducted by the European Space Agency (ESA), individual national space organizations, and industrial partners. As stated in [KG 95]: "The European Cooperation for Space Standardization (ECSS) is an initiative established to develop a coherent, single set of user-friendly standards for use in all European space activities." The resulting set of standards, available from the ECSS web portal [ECSS 20xx], addresses space activities as a whole and complement the relevant country-specific standards.

The ECSS standards specify requirements that must be satisfied (although project-specific tailoring is allowed) and fall into three categories: *space engineering* (the -E series), *project assurance* (the -Q series), and *project management* (the -M series). This book focuses on two specific standards – ECSS-E-ST-40C (Space engineering / Software) and ECSS-Q-ST-80C[1] (Space product

[1] All references to ECSS-Q-ST-80C in this book relate to the ECSS-Q-ST-80C-Rev1 edition.

assurance / Software product assurance) – and shows how the Ada and SPARK languages, together with AdaCore's product and services offerings, can help space software suppliers comply with these standards.

AdaCore has a long and successful history supporting developers of space software, and the company has proven experience and expertise in qualification under ECSS-E-ST-40C and ECSS-Q-ST-80C. Examples include:

- The ZFP (Zero Footprint) minimal run-time library for Ada on LEON2 ELF, qualified at criticality category B, for the aerospace company AVIO [Ad 2019a].
- The Ravenscar SFP (Small Footprint) QUAL run-time library for Ada on LEON2 and LEON3 boards, pre-qualified at criticality category B, for ESA [Ad 2019b].

The remainder of this chapter summarizes the ECSS-E-ST-40C and ECSS-Q-ST-80C standards.

Chapter 2 describes the Ada and SPARK programming languages and relates their software engineering support to the relevant sections / requirements in the two standards.

Analogously, Chapter 3 presents AdaCore's various software development and verification toolsuites and relates their functionality to the relevant sections / requirements in the two standards.

In the other direction, Chapter 4 surveys the individual requirements in ECSS-E-ST-40C and shows how a large number of them can be met by a software supplier through Ada, SPARK, and/or specific AdaCore products.

Chapter 5 does likewise for the requirements in ECSS-Q-ST-80C.

Chapters 6 contains a list of abbreviations/acronyms.

Chapter 7 is a bibliography.

Although this book is focused on specific ECSS standards, Chapters 2 and 3 explain how the Ada and SPARK languages / technologies benefit space software development in general and thus may also be applicable to software that has to comply with other regulatory standards.

1.1 ECSS-E-ST-40C: Space engineering / Software

As stated in ECSS-E-ST-40C [ECSS 2009], p. 12:

> *This Standard covers all aspects of space software engineering including requirements definition, design, production, verification and validation, transfer, operations and maintenance.*
>
> *It defines the scope of the space software engineering processes and its interfaces with management and product assurance, which are addressed in the Management (-M) and Product assurance (-Q) branches of the ECSS System, and explains how they apply in the software engineering processes.*

ECSS-E-ST-40C specifies the requirements for the following software engineering processes:

- Software related systems requirements process

  This process links the system and software levels and "establishes the functional and the performance requirements baseline (including the interface requirement specification) (RB) of the software development" [ECSS 2009], p. 26].

- Software management process

This process “tailors the M standards for software-specific issues” and produces “a software development plan including the life cycle description, activities description, milestones and outputs, the techniques to be used, and the risks identification” [ECSS 2009, pp. 26, 27]. It covers the joint review process, interface management, and technical budget and margin management.

- Software requirements and architecture engineering process

This process comprises software requirements analysis (based on system requirements) and a resulting software architecture design. Activities associated with the latter include selection of a design method, selection of a computational model for real-time software, description of software behavior, development and documentation of the software interfaces, and definition of methods and tools for software intended for reuse.

- Software design and implementation engineering process

This process covers the detailed design of the software items (including an analysis of the dynamic model showing how issues such as storage leakage and corrupted shared data are avoided), coding, testing, and integration.

- Software validation process

Software validation entails “software product testing against both the technical specification and the requirements baseline” and “confirm[ing] that the technical specification and the requirements baseline functions and performances are correctly and completely implemented in the final product" [ECSS 2009, p. 28].

- Software delivery and acceptance process

This process "prepares the software product for delivery and testing in its operational environment" [ECSS 2009, p. 28].

- Software verification process

Software verification "confirm[s] that adequate specifications and inputs exist for every activity and that the outputs of the activities are correct and consistent with the specifications and inputs. This process is concurrent with all the previous processes." [ECSS 2009, p. 28]

- Software operation process

This process involves the activities needed to ensure that the software remains operational for its users; these include "mainly the helpdesk and the link between the users, the developers or maintainers, and the customer" [ECSS 2009, p. 29]

- Software maintenance process

This process comprises the relevant activities "when the software product undergoes any modifications to code or associated documentation as a result of correcting an error, a problem or implementing an improvement or adaptation" [ECSS 2009, p. 30]

The standard specifies the requirements associated with each of these processes and defines the expected output for each requirement. The expected output identifies three entities:

- the relevant destination file,
- the DRL (Document Requirements List) item(s) within that file where the requirement is addressed, and
- the review that will assess whether the requirement is met.

The files in question are the RB (Requirements Baseline), TS (Technical Specification), DDF (Design Definition File), DJF (Design Justification File), MGT (Management File), MF (Maintenance File), OP (Operational Plan), and PAF (Product Assurance File).

The reviews are the SRR (System Requirements Review), PDR (Preliminary Design Review), CDR (Critical Design Review), QR (Qualification Review), AR (Acceptance Review), and ORR (Operational Readiness Review).

Table 1-1, from Annex A of ECSS-E-ST-40C, shows the association between files, DRL items, and reviews. Shaded cells indicate requirements from ECSS-Q-ST-80C.

Table 1-1: ECSS-E-ST-40 and ECSS-Q-ST-80 Document requirements list (DRL)

File	DRL item	SRR	PDR	CDR	QR	AR	ORR
RB	Software system specification (SSS)	✓					
	Interface requirements document (IRD)	✓					

Table 1-1: ECSS-E-ST-40 and ECSS-Q-ST-80 Document requirements list (DRL)							
File	**DRL item**	**SRR**	**PDR**	**CDR**	**QR**	**AR**	**ORR**
	Safety and dependability analysis results for lower level suppliers	✓					
TS	Software requirements specification (SRS)		✓				
	Software interface control document (ICD)		✓	✓			
DDF	Software design document (SDD)		✓	✓			
	Software configuration file (SCF)		✓	✓	✓	✓	✓
	Software release document (SRelD)				✓	✓	
	Software user manual (SUM)			✓	✓	✓	
	Software source code and media labels			✓			
	Software product and media labels				✓	✓	✓
	Training material				✓		
DJF	Software verification plan (SVerP)		✓				
	Software validation plan (SValP)		✓				
	Independent software verification and validation plan	✓	✓				

Table 1-1: ECSS-E-ST-40 and ECSS-Q-ST-80 Document requirements list (DRL)							
File	**DRL item**	**SRR**	**PDR**	**CDR**	**QR**	**AR**	**ORR**
	Software integration test plan (SUITP)		✓	✓			
	Software unit test plan (SUITP)			✓			
	Software validation specification (SVS) with respect to TS			✓			
	Software validation specification (SVS) with respect to RB				✓	✓	
	Acceptance test plan				✓	✓	
	Software unit test report			✓			
	Software integration test report			✓			
	Software validation report with respect to TS			✓			
	Software validation report with respect to RB				✓	✓	
	Acceptance test report					✓	
	Installation report					✓	
	Software verification report (SVR)	✓	✓	✓	✓	✓	✓
	Independent software verification and validation report		✓	✓	✓	✓	✓

Table 1-1: ECSS-E-ST-40 and ECSS-Q-ST-80 Document requirements list (DRL)

File	DRL item	SRR	PDR	CDR	QR	AR	ORR
	Software reuse file (SRF)	✓	✓	✓			
	Software problems reports and non-conformance reports	✓	✓	✓	✓	✓	✓
	Joint review reports	✓	✓	✓	✓	✓	
	Justification of selection of operational ground equipment and support services	✓	✓				
MGT	Software development plan (SDP)	✓	✓				
	Software review plan (SRevP)	✓	✓				
	Software configuration management plan	✓	✓				
	Training plan	✓					
	Interface management procedures	✓					
	Identification of NRB SW and members	✓					
	Procurement data	✓	✓				
MF	Maintenance plan				✓	✓	✓
	Maintenance records				✓	✓	✓

Table 1-1: ECSS-E-ST-40 and ECSS-Q-ST-80 Document requirements list (DRL)							
File	**DRL item**	**SRR**	**PDR**	**CDR**	**QR**	**AR**	**ORR**
	SPR and NCR- Modification analysis report- Problem analysis report- Modification documentation- Baseline for change- Joint review reports						
	Migration plan and notification						
	Retirement plan and notification						
OP	Software operation support plan						✓
	Operational testing results						✓
	SPR and NCR- User's request record- Post operation review report						✓
PAF	Software product assurance plan (SPAP)	✓	✓	✓	✓	✓	✓
	Software product assurance requirements for suppliers	✓					
	Audit plan and schedule	✓					
	Review and inspection plans or procedures						

Table 1-1: ECSS-E-ST-40 and ECSS-Q-ST-80
Document requirements list (DRL)

File	DRL item	SRR	PDR	CDR	QR	AR	ORR
	Procedures and standards		✓				
	Modelling and design standards	✓	✓				
	Coding standards and description of tools		✓				
	Software problem reporting procedure		✓				
	Software depend-ability and safety analysis report-Criticality classific-ation of software components		✓	✓	✓	✓	
	Software product assurance report						
	Software product assurance mile-stone report (SPAMR)	✓	✓	✓	✓	✓	✓
	Statement of compliance with test plans and procedures			✓	✓	✓	✓
	Records of training and experience						
	(Preliminary) alert information						

Table 1-1: ECSS-E-ST-40 and ECSS-Q-ST-80 Document requirements list (DRL)							
File	DRL item	SRR	PDR	CDR	QR	AR	ORR
	Results of pre-award audits and assessments, and of procurement sources						
	Software process assessment plan						
	Software process assessment records						
	Review and inspection reports						
	Receiving inspection report	✓	✓	✓	✓		
	Input to product assurance plan for systems operation						✓

1.2 ECSS-Q-ST-80C: Space product assurance / Software product assurance

The ECSS-Q-ST-80C standard defines software product assurance requirements for the development and maintenance of space software systems, including non-deliverable software that affects the quality of the deliverable product. As stated in [ECSS 2017], p. 20:

> *The objectives of software product assurance are to provide adequate confidence to the customer and to the supplier that the developed or procured/reused software satisfies its requirements throughout the system lifetime. In particular, that the software is*

developed to perform properly and safely in its operational environment, meeting the quality objectives agreed for the project.

The requirements apply throughout the software lifecycle and cover a range of activities, including organizational responsibilities, process assessment, development environment selection, and product verification. The specific set of requirements that need to be met can be tailored based on several factors:

- Dependability and safety aspects, as determined by the software criticality category,
- Software development constraints, for example the type of development (database vs. real-time), or
- Product quality / business objectives as specified by the customer

ECSS-Q-ST-80C defines requirements in the following areas:

- Software product assurance programme implementation

This set of activities includes organizational aspects, product assurance management, risk management and critical item control, supplier selection and control, procurement, tools and supporting environment selection, and assessment and improvement process.

- Software process assurance

These activities comprise software life cycle management; requirements applicable to all software engineering processes (e.g., documentation, safety analysis, handling of critical software, configuration management, metrics, verification, reuse, and automatic code generation); and requirements applicable to individual software engineering processes or activities (e.g., requirements analysis, architecture and design, coding, testing and validation, delivery and acceptance, operations, and maintenance).

- Software product quality assurance

These activities comprise product quality objectives and metrication; product quality requirements; software intended for reuse; standard ground hardware and services for operational system; and firmware.

As with ECSS-E-ST-40C, the expected output for each requirement identifies the destination file, the DRL items within that file, and the review(s) that assess compliance with the requirement. Table 1-1 above includes this information for the requirements in ECSS-Q-ST-80C.

1.3 ECSS Handbooks

Supplementing the normative standards in the -E, -Q, and -M series, ECSS has published a set of handbooks offering additional support, guidance and practical discussion about the standards and their requirements. They indicate how a customer (the organization acquiring the space software or system) will likely interpret the standards and thus how they will expect the supplier to comply.

The handbooks associated with ECSS-E-ST-40C are ECSS-E-HB-40A (Software engineering handbook) [ECSS 2013], which provides some general discussion and explanations, and ECSS-E-HB-40-01A (Agile software development handbook) [ECSS 2020].

Several handbooks complement ECSS-Q-ST-80C, including:

- ECSS-Q-HB-80-01A (Reuse of existing software)

  This handbook [ECSS 2011b] offers guidance on software reuse (including software tools) and also discusses the notion of Tool Qualification Level (TQL) based on D0-178C [RTCA 2011] and ISO 26262 [ISO 2018].

- ECSS-Q-HB-80-03A Rev.1 (Software dependability and safety)

  This handbook [ECSS 2017] focuses on analysis techniques such as Failure Mode and Effects Analysis (FMEA) and their application to software; i.e., how to analyze what happens in case of failure due to software. It covers topics such as defensive programming and prevention of failure propagation.

- ECSS-Q-HB-80-04A (Software metrication program definition and implementation)

  This handbook [ECSS 2011a] offers recommendations on organizing and implementing a metrication program for space software projects.

A description of how AdaCore's technologies relate to the handbooks' guidance will be provided in a future version of this book.

Additionally, as the ECSS-E-ST-40 and ECSS-Q-ST-80 standards themselves evolve, this book will be updated to reflect the changes.

Benjamin M. Brosgol & Jean-Paul Blanquart

2 Programming Languages for Space Software

This chapter explains how space software developers can benefit from the Ada language and its formally analyzable SPARK subset. Unless explicitly stated otherwise, the Ada discussion applies to the current version of the language standard, Ada 2012.

2.1 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 / qualification against domain-specific standards. Although in principle almost any programming language can be used for software development, in practice the life-cycle costs for the high-assurance real-time software found in space systems are reduced when the chosen language has been explicitly designed for reliability, safety, security, and ease of maintenance of large, long-lived systems.

Ada helps meet high-assurance requirements through its support for sound software engineering principles, compile-time checks that guarantee type safety, and run-time checks for constraints such as array index bounds and scalar ranges. As will be explained below, the SPARK subset of Ada shares these benefits and adds an important advantage: the dynamic constraints are enforced through mathematics-based static analysis. This avoids run-time overhead for checks in production code while eliminating the risk that such a check could fail.

2.1.1 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 ("sub-programs");
- standard support for "programming in the large" and module reuse, including packages, Object-Oriented Programming, hierarchical package namespace ("child libraries"), and 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 / qualification requirements and/or small footprint constraints.

The language standard also includes:

- an extensive predefined environment with support for I/O, string handling, math functions, containers, 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).

Source code portability 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 environment.
- 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. Ada allows this to be defined in the program logic but separated from target-independent 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.

2.1.2 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 Ada 2012 (see [ACAA 2016], [Ba 2014], or [Ba 2015] for information about Ada).

A new version of the language standard is in progress and is expected to be completed in 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, space systems, air traffic control, 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.

2.1.3 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 raises an exception (run-time error).

The ability to specify range constraints makes the programmer's 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.

Figure 2-1 shows an example of an integer scalar range in the declaration of subtype `Test_Score`:

```
subtype Test_Score is Integer range 1..100;

My_Score : Test_Score
N        : Integer;
...
My_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
```

Figure 2-1: Integer Range in Ada

2.1.4 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 verify this behavior by testing, static analysis, or formal proof.

Figure 2-2 shows a skeletal example of 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;
```

Figure 2-2: Contract-Based Programing in Ada

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

Ada 2012 further allows type invariants and type / subtype predicates, which specify precisely what is and is not valid for any particular (sub)type, including composite types such as records and arrays. For example, the code fragment in Figure 2-3 specifies that field `Max_Angle` in the `Launching_Pad` structure below is the maximal angle allowed, given the distance D to the center of the launching pad and the height H of the rocket. The compiler will insert the necessary run-time checks when a `Launching_Pad` object is created, to verify this predicate as well as the constraints on the individual fields:

```
subtype Meter  is Float range 0.0 .. 200.0;
subtype Radian is Float range 0.0 .. 2.0 * Pi;

type Launching_Pad is
   record
      D, H      : Meter;
      Max_Angle : Radian;
   end record
with
   Predicate => Arctan (H, D) <= Max_Angle;
```

Figure 2-3: Type Predicate in Ada

2.1.5 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," which provides a hierarchical and extensible namespace for library units and thus eases the design and maintenance 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.

2.1.6 Generic templates

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

2.1.7 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 Object Oriented 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 hard real-time constraints and that can 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.

2.1.8 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.

2.1.9 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 object facility.

2.1.10 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 in which a task runs until blocked or preempted, and Ada 2005 introduced several others including Earliest Deadline First.

2.1.11 High-integrity systems

With its emphasis on sound software engineering principles, Ada supports the development of safety-critical and other high-integrity applications, including those that need to be certified/qualified against software standards such as ECSS-E-ST-40C and ECSS-Q-ST-80C for space systems, RTCA DO-178C for avionics, and CENELEC EN 50128 for rail systems. Similarly, Ada (and its SPARK subset) can help developers produce high Evaluation Assurance Level (EAL) code that meets security standards such as the Common Criteria [CCRA 20xx]. 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, so 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++.

The evolution of Ada has seen a 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 safety certification practical. Ada 2012 introduced contract-based programming facilities, allowing the programmer to specify preconditions

[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.

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.

2.1.12 Enforcing a coding standard

Ada is a large language, suitable for general-purpose programming, but the full language may be inappropriate in a safety- or security-critical application. The generality and flexibility of some features – especially those with complex run-time semantics – complicate analysis and could interfere with traceability / certification requirements or impose too large a memory footprint. A project will then need to define and enforce a coding standard that prohibits problematic features. Several techniques are available:

- **pragma** Restrictions

  This standard Ada pragma allows the user to specify language features that the compiler will reject. Sample restrictions include dependence on specific packages or on language features such as exceptions.

- **pragma** Profile

  This standard Ada pragma provides a common name for a collection of related `Restrictions` pragmas. The predefined **`pragma`** `Profile(Ravenscar)` is a shorthand for the various restrictions that comprise the Ravenscar tasking subset.

- Static analysis tool (coding standard enforcer)

  Other restrictions on Ada features can be detected by AdaCore's automated GNATcheck tool (see section 3.5.1) that is included with GNAT Pro Ada. The developer can configure this rule-based and tailorable tool to flag violations of the project's coding standard, such as usage of specific prohibited types or subprograms defined in otherwise-permitted packages.

2.1.13 Ada and the ECSS Standards

ECSS-E-ST-40C covers software engineering practice, and ECSS-Q-ST-80C covers software product assurance, and both of these areas are "sweet spots" that match Ada's strengths. Chapters 4 and 5 below relate specific clauses in these standards to individual features of the language; in summary, the use of Ada can help the software supplier meet the requirements from the following sections:

- ECSS-E-ST-40C
  - §5.4 Software requirements and architecture engineering process
    - §5.4.3 Software architecture design
  - §5.5 Software design and implementation engineering process
    - §5.5.2 Design of software items
  - §5.8 Software verification process
    - §5.8.3 Verification activities
  - §5.10 Software maintenance process
    - §5.10.4 Modification implementation
- ECSS-Q-ST-80C
  - §6.2 Requirements applicable to all software engineering processes
    - §6.2.3 Handling of critical software
  - §6.3 Requirements applicable to individual software engineering processes or activities
    - §6.3.4 Coding
  - §7.2 Product quality requirements

- §7.2.2 Design and related documentation

In brief, Ada is an internationally standardized language combining object-oriented programming features, well-engineered concurrency facilities, real-time predictability and performance, and built-in reliability through both compile-time and run-time checks. With its support for producing software that is correct, maintainable, and efficient, the language is especially well suited for writing space applications.

2.2 SPARK

2.2.1 SPARK Basics

SPARK[3] ([MC 2015], [AA 2021]) is a software development technology (programming language and verification toolset) specifically oriented around applications demanding an ultra-low defect level, in particular where safety and/or security are key requirements. SPARK Pro is the commercial-grade offering of the SPARK technology developed by AdaCore, Capgemini Engineering[4], and Inria. As will be described 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 wide variety of applications, including civil and military avionics, space satellite control, air traffic management / control, railway signaling, cryptographic software, medical devices, automotive

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

[4] Formerly Altran

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 *unambiguous* and *formal* semantics, which permit 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.

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 others excluded from SPARK analysis (SPARK and non-SPARK code can also 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 using automated proof 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 2-4 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 2-4: 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.

SPARK (and the SPARK proof tools) work with Ada 2012 syntax, but a SPARK program can also be expressed in Ada 95, with contracts captured as pragmas.

2.2.2 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.

2.2.2.1 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).

2.2.2.2 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.

2.2.2.3 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 at least 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.

2.2.2.4 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.

2.2.2.5 *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 security-critical systems at high Evaluation Assurance Levels where formal methods can provide the needed confidence.

2.2.3 Hybrid Verification

A typical scenario for hybrid verification is an in-progress project that is using traditional testing and has high-assurance requirements that can best be demonstrated 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

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

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

Figure 2-5: SPARK Code Invoking a Tested C Function

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; Figure 2-5 shows an example.

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

  The postcondition of the called subprogram can be assumed to be valid at the point following the return, although the validity needs to be established by testing. In the example shown in Figure 2-5, 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.

2.2.4 SPARK and the ECSS Standards

The qualities that make Ada an appropriate choice for space software also apply to SPARK (see section 2.1.13 above), and indeed the static determination that the code is free from run-time errors (“Silver” level of SPARK adoption) can significantly reduce the effort in showing that the software meets its requirements. Additionally, SPARK directly addresses several criteria in ECSS-Q-ST-80C requirement 6.2.3.2a: “use of a ‘safe subset’ of programming language” and “use of formal design language for formal proof”.

3 Tools for Space Software Development

This chapter explains how suppliers of space software can benefit from AdaCore's products. The advantages stem in general from reduced life cycle costs for developing and verifying high-assurance software. More specifically, in connection with space software qualification, several tools can help to show that an application complies with the requirements in ECSS-E-ST-40C and ECSS-Q-ST-80C.

3.1 AdaCore Tools and the Software Life Cycle

The software life cycle is often depicted as a "V" diagram, and Figure 3-1 shows how AdaCore's major products fit into the various stages. Although the stages are rarely performed as a single sequential 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.

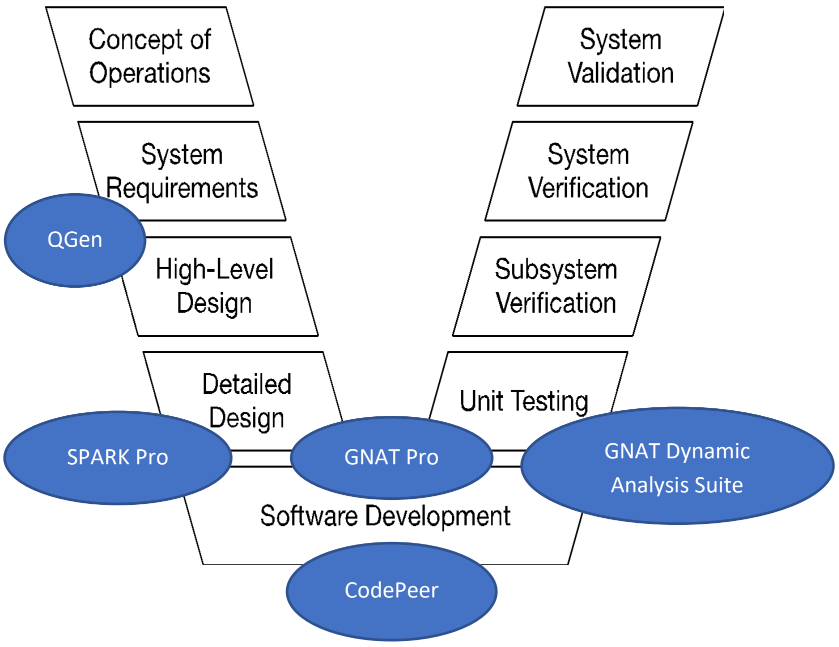

Figure 3-1: AdaCore Toolsuites and the "V" Software Life Cycle

In summary:

- The *QGen* model-based engineering environment (see page 51) applies during the Systems Requirements and High-Level Design phases. It is based around a qualifiable code generator from a safe subset of the Simulink® and Stateflow® modeling languages.
- The *SPARK Pro* static analysis toolsuite (see page 53) 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 58) 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 analysis tools, and a variety of supplemental libraries (including some for which qualification material is available with respect to ECSS-E-ST-40C and ECSS-Q-ST-80C.)
- The *CodePeer* advanced Ada static analysis tool (see page 73) 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.
- *The GNAT Dynamic Analysis Suite* (see page 76) applies during Software Development and Unit Testing. One of these tools, *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 space system software.

3.2 QGen Toolsuite for Model-Based Engineering

QGen is a qualifiable and tunable code generation and model verification toolsuite for a safe subset of the Simulink® and Stateflow® modeling languages. 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 ECSS-E-ST-40C and ECSS-Q-ST-80C, DO-178C, ISO 26262, or EN 50128.

3.2.1 QGen Capabilities

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 to test both the generated code and any hand-written code, on the host or the final target. It allows the user to perform a back-to-back comparison against expected values for a single Simulink® 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, corresponding roughly to a "development tool" in earlier editions of the DO-178 standard), with qualification anticipated in mid 2022. QGen at TQL-1 can reduce the effort needed to demonstrate compliance with some of the DO-178C objectives for the generated source code, 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 it is coupled with tests that verify the precise match between model simulation results and the run-time semantics of the generated target code.

A technology such as QGen can increase developer productivity by automating the translation of system requirements into source code. The MISRA-C and SPARK code generated by QGen can be augmented with developer-supplied code as needed.

3.2.2 QGen and the ECSS standards

Model-based engineering can be useful in space software development, and QGen can help meet a number of requirements in ECSS-E-ST-40C and ECSS-Q-ST-80C. Details are provided in chapters 5 and 6; in summary, these are the relevant sections of the two standards:

- ECSS-E-ST-40C
    - §5.4 Software requirements and architecture engineering process
        - §5.4.3 Software architecture design
    - §5.5 Software design and implementation engineering process
        - §5.5.2 Design of software items
    - §5.8 Software verification process
        - §5.8.3 Verification activities
- ECSS-Q-ST-80C
    - §6.2 Requirements applicable to all software engineering processes
        - §6.2.8 Automatic code generation

3.3 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.

3.3.1 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 44), 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.

3.3.2 Minimal Run-Time Footprint

Developers of systems with security requirements are generally advised to “minimize the trusted computing base”, making it as small as possible so that high-assurance verification is feasible. However, adhering to this principle may be difficult if a Commercial Off-the-Shelf (COTS) library or operating system is used: 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 Light run-time library[5]. In a Bare-Metal / light run-time 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 undergo evaluation by a national technical authority or regulator.

SPARK code can also run with a specialized run-time library on top of a real-time operating system (RTOS), 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.

3.3.3 CWE Compatibility

SPARK Pro detects a number of dangerous software errors in The MITRE Corporation’s Common Weakness Enumeration, and

[5] This library supersedes the Zero FootPrint (ZFP) run-time library from earlier GNAT Pro releases

the tool has been certified by the MITRE Corporation as a “CWE-Compatible” product [Mi 20xx].

Table 3-1lists the CWE weaknesses detected by SPARK Pro:

Table 3-1: SPARK Pro and the CWE

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 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

Table 3-1: SPARK Pro and the CWE

CWE Weakness	Description
CWE 821	Incorrect synchronization
CWE 822, 823, 824, 825	Pointer errors
CWE 835	Infinite loop

3.3.4 SPARK Pro and the ECSS Standards

SPARK Pro can help a space software supplier in various ways. At a general level, the technology supports the development of analyzable and portable code:

- The tool enforces a number of Ada restrictions that are appropriate for high-assurance software. 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 implement-ation 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.

More specifically, using the SPARK Pro technology can help the supplier meet ECSS-E-ST-40C and ECSS-Q-ST-80C requirements in a number of areas. These comprise the ones mentioned earlier (see section 2.2.4) that relate to the SPARK language, together with the following:

- ECSS-E-ST-40C
  - §5.6 Software validation process
    - §5.6.3 Validation activities with respect to the technical specification
    - §5.6.4 Validation activities with respect to the requirements baseline
  - §5.8 Software verification process
    - §5.8.3 Verification activities
- ECSS-Q-ST-80C
  - §5.6 Tools and supporting environment
    - 5.6.2 Development environment selection
  - §6.2 Requirements applicable to all software engineering processes
    - §6.2.3 Handling of critical software
  - §7.2 Product quality requirements
    - §7.2.3 Test and validation documentation

Details are provided in chapters 4 and 5.

3.4 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.

3.4.1 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 upcoming 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 software:

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 Standard run-time, which is available for platforms that can support the full language capabilities, the product on some bare-metal or RTOS targets also includes restricted libraries that reduce the footprint and/or help simplify safety certification:

- The *Light Run-Time library* offers a minimal application footprint while retaining compatibility with the SPARK subset and verification tools. It supports a non-tasking Ada subset suitable for certification / qualification and storage-constrained embedded applications. It supersedes the ZFP

(Zero FootPrint) (“Zero FootPrint”) and Cert run-time libraries from previous GNAT Pro releases.
- The *Light-Tasking run-time library* augments the Light run-time library with support for the Ravenscar tasking profile. It supersedes the Ravenscar-Cert and Ravenscar-SFP libraries from previous GNAT Pro releases.
- The Embedded run-time library provides a subset of the Standard Ada run-time library suitable for target platforms lacking file I/O and networking support. It supersedes the Ravenscar-Full library from previous GNAT Pro releases.

Details on these libraries may be found in the “Predefined GNAT Pro Run-Times” chapter of [Ad 2021}

Adapted versions of the earlier ZFP and Ravenscar-Cert libraries have been qualified under ECSS-E-ST-40C and ECSS-Q-ST-80C at criticality category B.

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.

3.4.2 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.

3.4.2.1 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.

3.4.2.2 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.

3.4.3 GNAT Pro Integrated Development Environments (IDEs)

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

3.4.3.1 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. Components written in C and C++ can also be handled through the IDE's multi-language capabilities. 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 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 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.

3.4.3.2 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.

3.4.3.3 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 developer can use GNATdashboard with GNATcheck, to monitor progress on meeting the coding standard constraints.

3.4.4 GNAT Pro and the ECSS Standards

GNAT Pro can help meet a number of requirements in ECSS-E-ST-40C and ECSS-Q-ST-80C. Details are provided in chapters 4 and 5; in summary, these are the relevant sections of the two standards:

- ECSS-E-ST-40C
  - §5.4 Software requirements and architecture engineering process
    - §5.4.3 Software architecture design
  - §5.5 Software design and implementation engineering process
    - §5.5.3 Coding and testing
    - §5.5.4 Integration
  - §5.6 Software validation process
    - §5.6.2 Validation process implementation
  - §5.7 Software delivery and acceptance process
    - §5.7.3 Software acceptance
  - §5.8 Software verification process
    - §5.8.3 Verification activities

    - §5.10 Software maintenance process
        - §5.10.2 Process implementation
- ECSS-Q-ST-80C
    - §5.2 Software product assurance programme management
        - §5.2.7 Quality requirements and quality models
    - §5.6 Tools and supporting environments
        - §5.6.1 Methods and tools
        - §5.6.2 Development environment selection
    - §6.2 Requirements applicable to all software engineering processes
        - §6.2.3 Handling of critical software
    - §6.3 Requirements applicable to individual processes or activities
        - §6.3.4 Coding
        - §6.3.8 Maintenance
    - §7.1 Product quality objectives and metrication
        - §7.1.3 Assurance objectives for product quality requirements
        - §7.1.5 Basic metrics

AdaCore's ZFP (Zero Footprint) minimal run-time library (superseded by the Light run-time in current GNAT Pro releases) on LEON2 ELF has been qualified at criticality category B [Ad 2019b], and the Ravenscar SFP (Small Footprint) QUAL run-time library (superseded by the Light-Tasking run-time) on LEON2 and LEON3 boards have been qualified at criticality category B [Ad 2019b].

3.5 GNAT Pro Ada Tools for Static Analysis and Target Emulation

This section describes a number of GNAT Pro Ada utilities for static analysis of Ada source code (GNATcheck, GNATmetric, GNATstack, libadalang) and one tool that supports host-based target emulation (GNATemulator). These tools help in general during development and verification and, as will be noted below, some are particularly useful in demonstrating compliance with ECSS standards.

Several tools for testing and code coverage are available as a supplement to GNAT Pro (the GNAT Dynamic Analysis Suite) and are described below (see section 3.7).

3.5.1 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 the demonstration of a system’s conformance with software safety standards.

Space software developers can use GNATcheck to help demonstrate that their Ada code meets the restrictions imposed by a project-specific coding standard.

Key features include:

- An integrated Ada Restrictions mechanism for banning particular 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.
- A user-friendly interface for creating and applying 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.

3.5.2 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).

3.5.3 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. GNATstack will be useful to space software developers since a stack overflow in an application at a high-criticality category 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. Figure 3-2 shows an example:

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

Figure 3-2: Subprogram with Unbounded Stack Frame

- 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.

3.5.4 Time and Space Analysis

3.5.4.1 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.

3.5.4.2 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.

3.5.5 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

3.5.6 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 an 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.

3.5.7 GNAT Pro Ada Tools and the ECSS Standards

GNAT Pro can help meet a number of requirements in ECSS-E-ST-40C and ECSS-Q-ST-80C. Details are provided in chapters 4 and 5; in summary, these are the relevant sections of the two standards:

- ECSS-E-ST-40C
    - §5.8 Software verification process
        - §5.8.3 Verification activities
- ECSS-Q-ST-80C
    - §5.2 Software product assurance programme management
        - §5.2.7 Quality requirements and quality models
    - §5.6 Tools and supporting environments
        - §5.6.2 Development environment selection
    - §6.2 Requirements applicable to all software engineering processes
        - §6.2.3 Handling of critical software
    - §6.3 Requirements applicable to individual processes or activities
        - §6.3.4 Coding

- §7.1 Product quality objectives and metrication
  - §7.1.3 Assurance objectives for product quality requirements
  - §7.1.5 Basic metrics

3.6 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.

3.6.1 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.

3.6.2 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].

Table 3-2 lists the weaknesses detected by CodePeer:

Table 3-2: CodePeer and the CWE

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

Table 3-2: CodePeer and the CWE

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

3.6.3 CodePeer and the ECSS Standards

CodePeer can help meet a number of requirements in ECSS-E-ST-40C and ECSS-Q-ST-80C. Details are provided in chapters 4 and 5; in summary, these are the relevant sections of the two standards:

- ECSS-E-ST-40C
    - §5.5 Software design and implementation engineering process
        - §5.5.2 Design of software items
    - §5.6 Software validation process
        - §5.6.3 Validation activities with respect to the technical specification
        - §5.6.4 Validation activities with respect to the requirements baseline
    - §5.8 Software verification process
        - §5.8.3 Verification activities

- ECSS-Q-ST-80C
    - §5.6. Tools and supporting environment
        - 5.6.1 Methods and tools
        - §5.6.2 Development environment selection
    - §6.2 Requirements applicable to all software engineering processes
        - §6.2.3 Handling of critical software

3.7 GNAT Dynamic Analysis Suite

The GNAT Dynamic Analysis Suite comprises tools for test case generation, host-based processor emulation, and code coverage analysis and reporting. A fuzz testing tool is in progress, which will help developers identify security vulnerabilities by checking the software's behavior in the presence of non-valid inputs.

3.7.1 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 the consistency of class hierarchies.

3.7.2 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 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.

3.7.3 GNAT Dynamic Analysis Suite and the ECSS Standards

The GNAT Dynamic Analysis Suite can help meet a number of requirements in ECSS-E-ST-40C and ECSS-Q-ST-80C. Details are provided in chapters 4 and 5; in summary, these are the relevant sections of the two standards:

- ECSS-E-ST-40C
    - §5.5 Software design and implementation engineering process
        - §5.5.3 Coding and testing
        - §5.5.4 Integration
    - §5.6 Software validation process

        - §5.6.3 Validation activities with respect to the technical specification
        - §5.6.4 Validation activities with respect to the requirements baseline
    - §5.8 Software verification process
        - §5.8.3 Verification activities
- ECSS-Q-ST-80C
    - §5.6 Tools and supporting environment
        - §5.6.1 Methods and tools
    - §6.2 Requirements applicable to all software engineering processes
        - §6.2.3 Handling of critical software
    - §6.3 Requirements applicable to individual software engineering processes or activities
        - §6.3.5 Testing and validation
    - §7.1 Product quality objectives and metrication
        - §7.1.5 Basic metrics

3.8 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, domain-specific software certification / qualification standards, compilation technologies, embedded system technology, and static and dynamic verification. AdaCore's development engineers have extensive experience supporting customers in critical areas including commercial and military avionics, space, air traffic management/control, railway, and automotive. 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, 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.

4 Compliance with ECSS-E-ST-40C

The ECSS-E-ST-40C standard is concerned with software engineering – the principles and techniques underlying the production of code that is reliable, safe, secure, readable, maintainable, portable and efficient. These are the goals that drove the design of the Ada language (and its SPARK subset), whose features assist in designing a modular and robust system architecture and in preventing or detecting errors such as type mismatches or buffer overruns that can arise in other languages.

This chapter explains how Ada and SPARK, together with the relevant AdaCore development and verification tools, can help a space software supplier meet many of the requirements presented in ECSS-E-ST-40C. The section numbers in braces refer to the associated content in ECSS-E-ST-40C.

4.1 Software requirements and architecture engineering process {§5.4}

4.1.1 Software architecture design {§5.4.3}

4.1.1.1 Transformation of software requirements into a software architecture {§5.4.3.1}

- "The supplier shall transform the requirements for the software into an architecture that describes the top-level structure; identifies the software components, ensuring that all the requirements for the software item are allocated to the software components and later refined to facilitate detailed design; covers as a minimum hierarchy, dependency, interfaces and operational usage for the software components; documents the process, data and control aspects of the product; describes the architecture static decomposition into software elements such as packages, classes or units; describes the dynamic architecture, which involves the identification of active objects such as threads, tasks and processes; describes the software behavior." {§5.4.3.1a}

  - The Ada and SPARK languages directly support this requirement. Relevant features include packages, child libraries, subunits, private types, tasking, and object-oriented programming (tagged types). The GNATstub utility (included with GNAT Pro Ada) is useful here; it generates empty package bodies ("stubs") from a software design's top-level API (package specs).

4.1.1.2 Software design method {§5.4.3.2}

- "The supplier shall use a method (e.g., object oriented or functional) to produce the static and dynamic architecture including: software elements, their interfaces and; software elements relationships." {§5.4.3.2a}
  - Ada and SPARK are methodology agnostic and fully support both object-oriented and functional styles.

4.1.1.3 Selection of a computational model for real-time software {§5.4.3.3}

- "The dynamic architecture design shall be described according to an analytical computational model." {§5.4.3.3a}

  - The Ada and SPARK tasking facility supports a stylistic idiom that is amenable to Rate Monotonic Analysis, allowing static verification that real-time deadlines will be met.

4.1.1.4 Description of software behavior {§5.4.3.4}

- "The software design shall also describe the behaviour of the software, by means of description techniques using automata and scenarios." {§5.4.3.4a}
  - The QGen tool supports model-based development where the software behavior is captured by Simulink® / Stateflow® models.

4.1.1.5 Development and documentation of the software interfaces {§5.4.3.5}

- "The supplier shall develop and document a software preliminary design for the interfaces external to the software item and between the software components of the software item." {§5.4.3.5a}
  - The supplier can use the Ada / SPARK package facility to specify the interfaces, both external and internal. The contract-based programming features provide additional expressive power, allowing the specification of pre- and postconditions for the subprograms comprising an interface.

4.1.1.6 Definition of methods and tools for software intended for reuse {§5.4.3.6}

- "The supplier shall define procedures, methods and tools for reuse, and apply these to the software engineering processes to comply with the reusability

requirements for the software development.”
{§5.4.3.6a}

- Ada and SPARK facilitate reuse via the separate compilation semantics (which allows “bottom-up” development by reusing existing libraries) and the generic facility (which, for example, allows a module to be defined in a general and type-independent fashion and then instantiated with specific types as needed). The semantics for these features enforces safe reuse:
    - All checks that are performed within a single compilation unit are also enforced across separate compilation boundaries.
    - A post-compilation pre-link check detects and prevents “version skew” (building an executable where some compilation unit depends on an obsolescent version of another unit).
    - Unlike the situation with C++ templates, a type mismatch in an Ada generic instantiation is detected and prevented at compile time, ensuring consistency between the instantiation and the generic unit.

4.2 Software design and implementation engineering process {§5.5}

4.2.1 Design of software items {§5.5.2}

4.2.1.1 Detailed design of each software component {§5.5.2.1}

- "The supplier shall develop a detailed design for each component of the software and document it." {§5.5.2.1a}
    - Ada / SPARK features, including packages and child units, help meet this requirement. The contract-based programming feature (e.g., pre- and postconditions) allows the supplier to express low-level requirements as part of the software architecture, facilitating the low-level design of algorithms.
- "Each software component shall be refined into lower levels containing software units that can be coded, compiled, and tested." {§5.5.2.1b}
    - Relevant Ada / SPARK features include packages, child units, and subunits.

4.2.1.2 Development and documentation of the software interfaces detailed design {§5.5.2.2}

- "The supplier shall develop and document a detailed design for the interfaces external to the software items, between the software components, and between the software units, in order to allow coding without requiring further information." {§5.5.2.2a}
    - Ada / SPARK features, including packages and child units, help meet this requirement. The contract-based programming feature (e.g., pre- and postconditions) allows the supplier to express low-level requirements as part of the

interfaces, facilitating the implementation of algorithms.

4.2.1.3 Production of the detailed design model {§5.5.2.3}

- "The supplier shall produce the detailed design model of the software components defined during the software architectural design, including their static, dynamic and behavioural aspects." {§5.5.2.3a}
    - Ada / SPARK features such as packages, child units, and contract-based programming help meet this requirement. If model-based engineering is used, AdaCore's QGen tool supports the expression of the detailed design as Simulink® / Stateflow® models.

4.2.1.4 Software detail design method {§5.5.2.4}

- "The supplier shall use a design method (e.g. object oriented or functional method) to produce the detailed design including: software units, their interfaces, and; software units relationships." {§5.5.2.4a}
    - Ada and SPARK are methodology agnostic and fully support both object-oriented and functional styles.

4.2.1.5 Detailed design of real-time software {§5.5.2.5}

- "The dynamic design model shall be compatible with the computational model selected during the software architectural design model" {§5.5.2.5a}
    - The Ada / SPARK tasking model allows a straightforward mapping from the architectural design (where the system comprises a collection of tasks that interact via protected shared resources) to the detailed design.

- "The supplier shall document and justify all timing and synchronization mechanisms" {§5.5.2.5b}
    - The Ada / SPARK tasking model supplies the necessary timing and synchronization support.
- "The supplier shall document and justify all the design mutual exclusion mechanisms to manage access to the shared resources." {§5.5.2.5c}
    - The Ada / SPARK tasking model supplies the necessary mutual exclusion mechanisms (protected objects, pragma Atomic). The CodePeer static analysis tool can detect potential race conditions with respect to the usage of shared resources.
- "The supplier shall document and justify the use of dynamic allocation of resources." {§5.5.2.5d}
    - Ada has a general and flexible mechanism for dynamic memory management, including the ability of the programmer to specify the semantics of allocation and deallocation within a storage pool. This can be used, for example, to define a fragmentation-free strategy for memory management with constant time for allocation and deallocation. The latest version of SPARK includes a facility for safe pointers.
- "The supplier shall ensure protection against problems that can be induced by the use of dynamic allocation of resources, e.g. memory leaks." {§5.5.2.5e}
    - Ada includes a variety of mechanisms that assist in preventing dynamic memory management issues. The No_Standard_Allocators_After_Elaboration argument to pragma Restrictions produces a

run-time check that detects attempts to perform allocations from a standard storage pool after elaboration (initialization). Depending on the program structure, static analysis may be able to determine that this check will never fail.

4.2.1.6 *Utilization of description techniques for the software behaviour {§5.5.2.6}*

- "The behavioural design of the software units shall be described by means of techniques using automata and scenarios." {§5.5.2.6a}
  - The QGen tool supports model-based development where the software behavior is captured by Simulink® / Stateflow® models.

4.2.2 Coding and testing {§5.5.3}

4.2.2.1 *Development and documentation of the software units {§5.5.3.1}*

- "The supplier shall develop and document the following: the coding of each software unit; the build procedures to compile and link software units" {§5.5.3.1a}
  - The GNAT Pro project and gprbuild facility automate the build process and prevent "version skew".

4.2.2.2 *Software unit testing {§5.5.3.2}*

- "The supplier shall develop and document the test procedures and data for testing each software unit" {§5.5.3.2a}
  - AdaCore's GNATtest and GNATcoverage tools can assist in this process.

- "The supplier shall test each software unit ensuring that it satisfies its requirements and document the test results." {§5.5.3.2b}
    - AdaCore's GNATtest and GNATcoverage tools can assist in this process.
- "The unit test shall exercise: code using boundaries at n-1, n, n+1 including looping instructions **while**, **for** and tests that use comparisons; all the messages and error cases defined in the design document; the access of all global variables as specified in the design document; out of range values for input data, including values that can cause erroneous results in mathematical functions; the software at the limits of its requirements (stress testing)." {§5.5.3.2c}
    - AdaCore's GNATtest and GNATcoverage tools can assist in this process.

4.2.3 Integration {§5.5.4}

4.2.3.1 Software units and software component integration and testing {§5.5.4.2}

- "The supplier shall integrate the software units and software components, and test them, as the aggregates are developed, in accordance with the integration plan, ensuring each aggregate satisfies the requirements of the software item and that the software item is integrated at the conclusion of the integration activity." {§5.5.4.2a}
    - AdaCore's GNATtest and GNATcoverage tools can assist in this process, supplementing the GNAT Pro Ada compilation facilities.

4.3 Software validation process {§5.6}

4.3.1 Validation activities with respect to the technical specification {§5.6.3}

4.3.1.1 Development and documentation of a software validation specification with respect to the technical specification {§5.6.3.1}

- "The supplier shall develop and document, for each requirement of the software item in TS [Technical Specification] (including ICD [Interface Control Document]), a set of tests, test cases (inputs, outputs, test criteria) and test procedures …." {§5.6.3.1a}
    - AdaCore's GNATtest and GNATcoverage tools can assist in this process.
- "Validation shall be performed by test." {§5.6.3.1b}
    - AdaCore's GNATtest and GNATcoverage tools can assist in this process.
- "If it can be justified that validation by test cannot be performed, validation shall be performed by either analysis, inspection or review of design" {§5.6.3.1c}
    - The CodePeer and/or SPARK Pro static analysis tools may be able to show that a run-time check will always succeed and that no test case will trigger a failure.

4.3.2 Validation activities with respect to the requirements baseline {§5.6.4}

4.3.2.1 Development and documentation of a software validation specification with respect to the requirements baseline {§5.6.4.1}

- "The supplier shall develop and document, for each requirement of the software item in RB [Requirements

Baseline] (including IRD [Interface Requirements Document]), a set of tests, test cases (inputs, outputs, test criteria) and test procedures ….” {§5.6.4.1a}
    - AdaCore’s GNATtest and GNATcoverage tools can assist in this process.
- “Validation shall be performed by test.” {§5.6.4.1b}
    - AdaCore’s GNATtest and GNATcoverage tools can assist in this process.
- “If it can be justified that validation by test cannot be performed, validation shall be performed by either analysis, inspection or review of design” {§5.6.4.1c}
    - The CodePeer and/or SPARK Pro static analysis tools may be able to show that a run-time check will always succeed and that no test case will trigger a failure.

4.4 Software delivery and acceptance process {§5.7}

4.4.1 Software acceptance {§5.7.3}

4.4.1.1 Executable code generation and installation {§5.7.3.3}

- “The acceptance shall include generation of the executable code from configuration managed source code components and its installation on the target environment.” {§5.7.3.3a}
    - The GNAT Pro project and gprbuild facility can assist in the build and installation process.

4.5 Software verification process {§5.8}

4.5.1 Verification activities {§5.8.3}

4.5.1.1 Verification of code {§5.8.3.5}

- "The supplier shall verify the software code ensuring that: 1. the code is externally consistent with the requirements and design of the software item; 2. there is internal consistency between software units; 3. the code is traceable to design and requirements, testable, correct, and in conformity to software requirements and coding standards; 4. the code that is not traced to the units is justified; 5. the code implements proper events sequences, consistent interfaces, correct data and control flow, completeness, appropriate allocation of timing and sizing budgets, and error handling; 6. the code implements safety, security, and other critical requirements correctly as shown by appropriate methods; 7. the effects of run-time errors are controlled; 8. there are no memory leaks; 9. numerical protection mechanisms are implemented." {§5.8.3.5a}
    - Ada's strong typing and interface checks help meet criterion 2.
    - For criterion 3, static analysis tools such as CodePeer and SPARK Pro can help verify correctness, and the GNATcheck utility included with GNAT Pro can enforce conformance with a coding standard.
    - For criterion 5, QGen can help ensure proper events sequences (based on the Simulink® / Stateflow® model that is input), and Ada's strong typing and interface checks can help show consistent interfaces and correct data flow.

  - Static analysis tools such as CodePeer and SPARK Pro, as well as the standard semantic checks performed by the GNAT Pro compiler, can help meet criterion 6.
  - Ada's exception handling facility can help meet criterion 7.
  - Ada's user-defined memory management facilities can help meet criterion 8.
- "The supplier shall verify that the following code coverage is achieved:

Code coverage versus criticality category	A	B	C	D
Source code statement coverage	100%	100%	AM	AM
Source code decision coverage	100%	100%	AM	AM
Source code modified condition and decision coverage	100%	AM	AM	AM

Note: 'AM' means that the value is agreed with the customer and measured as per ECSS-Q-ST-80C clause 6.3.5.2." {§5.8.3.5b}
  - GNATcoverage can help meet this requirement.
- "Code coverage shall be measured by analysis of the results of the execution of tests." {§5.8.3.5c}
  - GNATcoverage can help meet this requirement.
- "In case the traceability between source code and object code cannot be verified (e.g. use of compiler

optimization), the supplier shall perform additional code coverage analysis on object code level as follows:

Code coverage VS. criticality category	A	B	C	D
Object code coverage	100%	N/A	N/A	N/A

Note: N/A means not applicable." {§5.8.3.5e}

- GNATcoverage can help meet this requirement.
- AdaCore can prepare an analysis of traceability between source and object code; the company has provided this to customers in connection with certification under the DO-178C standard for airborne software for the commercial aviation industry.

• "The supplier shall verify source code robustness (e.g. resource sharing, division by zero, pointers, run-time errors). AIM: use static analysis for the errors that are difficult to detect at run-time." {§5.8.3.5f}
  - Errors such as division by zero, null pointer dereferencing, array indices out of bounds, and many others are flagged at run-time by raising an exception. Effective practice is to keep these checks enabled during development and then, after verifying either statically or through sufficient testing that the run-time checks are not needed, disable the checks in the final code for maximal efficiency.
  - The CodePeer static analysis tool will detect such errors as well as many others, including

suspicious constructs that, although legitimate Ada, are likely logic errors.
  - The SPARK Pro tool will enforce the SPARK subset and can be used to demonstrate absence of run-time errors.
  - The GNATstack tool computes the potential maximum stack usage for each task in a program. Combining the result with a separate analysis showing the maximal depth of recursion, the developer can allocate sufficient stack space for program execution and prevent stack overflow.

4.5.1.2 Verification of software unit testing (plan and results) {§5.8.3.6}

- "The supplier shall verify the unit tests results…."{§5.8.3.6a}
  - The GNATtest and GNATcoverage tools contribute to meeting this requirement.

4.5.1.3 Schedulability analysis for real-time software {§5.8.3.11}

- "As part of the verification of the software requirements and architectural design, the supplier shall use an analytical model (or use modelling and simulation, if it can be demonstrated that no analytical model exists) to perform a schedulability analysis and prove that the design is feasible." {§5.8.3.11a}
  - The Ada tasking model, and the Ravenscar profile in particular, supports Rate-Monotonic Analysis.

    - The QGen tool supports model-based development (generating SPARK or MISRA-C code from Simulink® / Stateflow® models)

4.5.1.4 *Behaviour modelling verification {§5.8.3.13}*

- "As support to the verification of the software architectural design, the supplier shall verify the software behaviour using the behavioural view of the architecture produced in clause 5.4.3.4." {§5.8.3.13b}
    - The QGen tool can help meet this requirement if the software behavior is captured by Simulink® / Stateflow® models.
- "As support to the verification of the software detailed design, the supplier shall verify the software behaviour using the software behavioural design model produced in 5.5.2.3a. eo c., by means of the techniques defined in 5.5.2.6." {§5.8.3.13c}
    - The QGen tool can help meet this requirement if the software behavior is captured by Simulink® / Stateflow® models.

4.6 Software operation process {§5.9}

4.6.1 Process implementation {§5.9.2}

4.6.1.1 *Problem handling procedures definition {§5.9.2.3}*

- "The SOS [Software Operation Support] entry shall establish procedures for receiving, recording, resolving, tracking problems, and providing feedback." {§5.9.2.3a}
    - In the event that a product problem is due to a defect in an AdaCore tool (e.g., a code generation bug), AdaCore has a rigorous QA process for responding to and resolving such issues. The "sustained branch" service, which is

included with a GNAT Pro Assurance subscription, helps by ensuring that a specific version of the toolchain is maintained over the lifetime of the supplier's project.

4.6.2 Software operation support {§5.9.4}

4.6.2.1 Problem handling {§5.9.4.2}

- "Encountered problems shall be recorded and handled in accordance with the applicable procedures." {§5.9.4.2a}
    - As described above in connection with clause 5.9.2.3, AdaCore's QA process and the "sustained branch" service can help meet this requirement when an issue arises that is due to an AdaCore tool.

4.7 Software maintenance process {§5.10}

4.7.1 Process implementation {§5.10.2}

4.7.1.1 Long term maintenance for flight software {§5.10.2.2}

- "If the spacecraft lifetime goes after the expected obsolescence date of the software engineering environment, then the maintainer shall propose solutions to be able to produce and upload modifications to the spacecraft up to its end of life." {§5.10.2.2a}
    - AdaCore's "sustained branch" service, which is included with a GNAT Pro Assurance subscription, in effect means that the compilation environment will not become obsolescent.

4.7.2 Modification implementation {§5.10.4}

4.7.2.1 Invoking of software engineering processes for modification implementation {§5.10.4.3}

- "The maintainer shall apply the software engineering processes as specified in clauses 5.3 to 5.8 while implementing the modifications." {§5.10.4.3a}
    - The Ada language has specific features that support the design of modular, maintainable software with high cohesion and low coupling. These include encapsulation (private types, separation of specification from implementation), hierarchical child libraries, and object-oriented programming (tagged types). By exploiting these features, the developer can localize the impact of maintenance changes.

5 Compliance with ECSS-Q-ST-80C

The ECSS-Q-ST-80C standard defines software product assurance requirements for the development and maintenance of space software systems. This chapter explains how AdaCore's products can help a supplier meet many of these requirements. The section numbers in braces refer to the relevant content in ECSS-Q-ST-80C.

5.1 Software product assurance programme implementation {§5}

5.1.1 Software product assurance programme management {§5.2}

5.1.1.1 Quality requirements and quality models {§5.2.7}

- "Quality models shall be used to specify the software quality requirements" {§5.2.7.1a}
    - The GNAT Pro Ada utility GNATmetric can be used to show quality metrics related to the source code structure.
    - The GNAT Pro Ada utility GNATdashboard can be used as a general tool to display software quality data.

5.1.2 Tools and supporting environment {§5.6}

5.1.2.1 Methods and tools {§5.6.1}

- "Methods and tools to be used for all activities of the development cycle … shall be identified by the supplier and agreed by the customer" {§5.6.1.1a}

    - The GNAT Pro Ada environment and any supplemental tools that are selected (e.g., GNATcoverage, CodePeer, SPARK Pro, QGen) should be listed.

- "The choice of development methods and tools shall be justified by demonstrating through testing or documented assessment that … the tools and methods are appropriate for the functional and operational characteristics of the product, and … the tools are available (in an appropriate hardware environment) throughout the development and maintenance lifetime of the product" {§5.6.1.2a}
    - AdaCore can make available a variety of documentation showing that the selected tools are "appropriate for the functional and operational characteristics of the product", ranging from user manuals to qualification material relative to other high-assurance software standards such as DO-178C and EN 50128 [Cen 2011].
    - AdaCore's "sustained branch" service for its GNAT Pro Ada Assurance product (see section 3.4.2.1 above) can guarantee that the toolchain is maintained throughout the product lifetime.

5.1.2.2 Development environment selection {§5.6.2}

- "The software development environment shall be selected according to the following criteria: 1. availability; 2. compatibility; 3. performance; 4. maintenance; 5. durability and technical consistency with the operational environment; 6. the assessment of the product with respect to requirements, including the criticality category; 7. the available support

documentation; 8. the acceptance and warranty conditions; 9. the conditions of installation, preparation, training and use; 10. the maintenance conditions, including the possibilities of evolutions; 11. copyright and intellectual property rights constraints;
12. dependence on one specific supplier" {§5.6.2.1a}

  - AdaCore tools directly satisfy these criteria. The availability of qualification material for specific tools (CodePeer, GNATcheck, GNATprove, GNATstack) contributes to criterion 6, and the "sustained branch" service for GNAT Pro Ada Assurance supports criterion 7. All AdaCore tools come with source code and flexible licensing, mitigating criterion 12 (dependence on AdaCore as the supplier).

5.2 Software process assurance {§6}

5.2.1 Requirements applicable to all software engineering processes {§6.2}

5.2.1.1 Handling of critical software {§6.2.3}

- "The supplier shall define, justify and apply measures to assure the dependability and safety of critical software.... These measures can include: … use of a 'safe subset' of programming language; use of formal design language for formal proof; 100% code branch coverage at unit testing level; … removing deactivated code or showing through a combination of analysis and testing that the means by which such code can be inadvertently executed are prevented, isolated, or eliminated." {§6.2.3.2a}
  - Ada's pragma Restrictions and pragma Profile, together with the GNAT Pro Ada utility

    GNATcheck, can enforce a coding standard for Ada (in effect a 'safe subset'). See section 3.5.1 above.
  - SPARK Pro uses proof technology that can demonstrate a program's conformance with formally specified requirements.
  - GNATcoverage can report code coverage up to MC/DC at the source level and branch coverage at the object level.
  - CodePeer can detect unreachable code, including deactivated code.
- "Software containing deactivated code shall be verified specifically to ensure that the deactivated code cannot be activated or that its accidental activation cannot harm the operation of the system." {§6.2.6.5}
  - CodePeer can detect unreachable code, including deactivated code.

5.2.1.2 Automatic code generation {§6.2.8}

- "For the selection of tools for automatic code generation, the supplier shall evaluate the following aspects: … customization of the tools to comply with project standards; portability requirements for the generated code; …." {§6.2.8.1a}
  - The QGen tool can be customized to generate specific output code, and the source code that it produces in SPARK or MISRA C is portable.
- "The required level of verification and validation of the automatic generation tool shall be at least the same as the one required for the generated code, if the tool is used to skip verification or testing activities on the target code." {§6.2.8.3a}

  - QGen is due to achieve DO-178C qualification at level TQL-1 (the highest / most rigorous Tool Qualification Level) in mid 2022. Since an error in the tool could result in faulty code in the executable object code, achieving TQL-1 qualification basically entails demonstrating that the tool meets a set of objectives analogous to the ones that apply to software at level A, the highest assurance level (comparable to criticality category A).
  - With QGen qualified at TQL-1, the testing performed at the level of the Simulink® / Stateflow® models can be used to eliminate manual review of the generated source code, reduce or eliminate the need for a separate set of low-level requirements-based tests, and enable model coverage to substitute for low-level code coverage metrics.

5.2.2 Requirements applicable to individual software engineering processes or activities {§6.3}

5.2.2.1 Coding {§6.3.4}

- "Coding standards (including consistent naming conventions and adequate commentary rules) shall be specified and observed." {§6.3.4.1a}
  - Ada's Restrictions and Profile pragmas, together with AdaCore's GNATcheck tool (see section 3.5.1 above), can define and enforce a coding standard.
- "The tools to be used in implementing and checking conformance with coding standards shall be identified

in the product assurance plan before coding activities start.” {§6.3.4.2a}
    - GNATcheck is the relevant tool for this activity.
- “The supplier shall define measurements, criteria and tools to ensure that the software code meets the quality requirements.” {§6.3.4.6a}
    - The GNATmetric tool (see section 3.5.2 above) can be used to report quality traits related to the source code structure.
- “Synthesis of the code analysis results and corrective actions implemented shall be described in the software product assurance reports.” {§6.3.4.7a}
    - GNATdashboard (see section 3.4.3.3 above) can be used to synthesize and summarize code quality metrics.

5.2.2.2 Testing and validation {§6.3.5}

- “Testing shall be performed in accordance with a strategy for each testing level (i.e. unit, integration, validation against the technical specification, validation against the requirements baseline, acceptance), ….” {§6.3.5.1a}
    - AdaCore’s GNAT Dynamic Analysis Suite (GNATtest, GNATcoverage) can help meet this requirement.
- “Based on the criticality of the software, test coverage goals for each testing level shall be agreed between the customer and the supplier and their achievement monitored by metrics…” {§6.3.5.2a)
    - AdaCore’s GNAT Dynamic Analysis Suite (GNATtest, GNATcoverage) can help meet this requirement.

- “Test coverage shall be checked with respect to the stated goals.” {§6.3.5.5a}
    - AdaCore’s GNAT Dynamic Analysis Suite (GNATtest, GNATcoverage) can help meet this requirement.
- “The test coverage of configurable code shall be checked to ensure that the stated requirements are met in each tested configuration.” {§6.3.5.7a}
    - AdaCore’s GNAT Dynamic Analysis Suite (GNATtest, GNATcoverage) can help meet this requirement.
- “Test tool development or acquisition … shall be planned for in the overall project plan.” {§6.3.5.24a}
    - The tools in AdaCore’s GNAT Dynamic Analysis Suite (GNATtest, GNATcoverage) are candidates for consideration in test tool acquisition.
- “Software containing deactivated code shall be validated specifically to ensure that the deactivated code cannot be activated or that its accidental activation cannot harm the operation of the system.” {§6.3.5.30a}
    - AdaCore’s GNAT Dynamic Analysis Suite (GNATtest, GNATcoverage) can help meet this requirement by detecting non-covered code that is intended (and can be categorized) as deactivated.
- “Software containing configurable code shall be validated specifically to ensure that unintended configuration cannot be activated at run time or included during code generation”. {§6.3.5.31a}
    - AdaCore’s GNAT Dynamic Analysis Suite (GNATtest, GNATcoverage) can help meet this

requirement by detecting configurable code that, because of incorrect configuration settings, was unintentionally included during code generation.

5.2.2.3 Maintenance {§6.3.8}

- "The maintenance plans and procedures shall include the following as a minimum: scope of maintenance; identification of the first version of the software product for which maintenance is to be done; support organization; maintenance life cycle; maintenance activities; quality measures to be applied during the maintenance; maintenance records and reports." {§6.3.8.4a}
    - The "sustained branch" service of AdaCore's GNAT Pro Assurance product can help meet this requirement.
- "Maintenance records shall be established for each software product…." {§6.3.8.7a}
    - AdaCore's ticket system, which is part of the standard support in all product subscriptions, provides an audit trail for problem reports / resolution.
    - The "sustained branch" service includes special maintenance accommodations for dealing with problems that relate to software safety.

5.3 Software product quality assurance {§7}

5.3.1 Product quality objectives and metrication {§7.1}

5.3.1.1 Assurance activities for product quality requirements {§7.1.3}

- "The supplier shall define assurance activities to ensure that the product meets the quality requirements as specified in the technical specification" {§7.1.3a}
    - Any of AdaCore's tools could potentially contribute to meeting this requirement, depending on the nature of the metrics that have been defined, and the GNATdashboard tool can serve to integrate the metrics in a meaningful way.

5.3.1.2 Basic metrics {§7.1.5}

- "The following basic products metrics shall be used: size (code); complexity (design, code); fault density and failure intensity; test coverage; number of failures." {§7.1.5a}
    - The GNATmetric, GNATtest, and GNATcoverage tools directly help to meet this requirement.

5.3.2 Product quality requirements {§7.2}

5.3.2.1 Design and related documentation {§7.2.2}

- "The software shall be designed to facilitate testing." {§7.2.2.2a}
    - The Ada language encourages and supports the use of sound software engineering principles such as modular design and structured programming, which makes the code easier to test.

- "Software with a long planned lifetime shall be designed with minimum dependency on the operating system and the hardware in order to aid portability." {§7.2.2.3a}
    - The Ada language has abstracted away the specifics of the underlying operating system and hardware through standard syntax and semantics for features such as concurrency, memory management, exception handling, and I/O. As a result, Ada programs can often be ported across different processor architectures and operating systems by simply recompiling, with minimal or no source code changes needed.

5.3.2.2 Test and validation documentation {§7.2.3}

- "For any requirements not covered by testing a verification report shall be drawn up documenting or referring to the verification activities performed." {§7.2.3.6a}
    - In many cases where verification cannot be achieved by testing, SPARK Pro may be able to provide convincing alternative verification evidence (for example, a robustness demonstration by proof that an out-of-range or otherwise non-valid input will never be passed to the unit being verified).

6 Abbreviations

Abbreviation	Expansion
API	Application Program Interface
AR	Acceptance Review
CDR	Critical Design Review
DDF	Design Definition File
DJF	Design Justification File
DRD	Document Requirements Definition
DRL	Document Requirements List
EAL	Evaluation Assurance Level
ECSS	European Cooperation for Space Standardization
ESA	European Space Agency
GCC	GNU Compiler Collection
GUI	Graphical User Interface
IDE	Integrated Development Environment
LSP	Language Server Protocol Liskov Substitution Principle
MF	Maintenance File
MGT	Management File
OP	Operational Plan
ORR	Operational Readiness Review
PAF	Product Assurance File
PDR	Preliminary Design Review
QR	Qualification Review
RB	Requirements Baseline
RTOS	Real-Time Operating Systems
SRR	System Requirements Review
TQL	Tool Qualification Level
TS	Technical Specification

7 References

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

[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`

[Ad 2019a] AdaCore (press release), *AVIO Selects AdaCore's GNAT Pro Assurance Toolsuite for European Space Agency Program*; January 8, 2019. `https://www.adacore.com/press/avio-selects-adacores-gnat-pro-assurance-toolsuite-for-european-space-agency-program`

[Ad 2019b]	AdaCore (press release), *European Space Agency Selects AdaCore's Qualified-Multitasking Solution for Spacecraft Software Development*; September 24, 2019; `https://www.adacore.com/press/european-space-agency-selects-adacores-qualified-multitasking-solution-for-spacecraft-software-development`
[Ad 2021]	AdaCore, *GNAT User's Guide Supplement for Cross Platforms*, Version 23.0W, October 2021; `https://docs.adacore.com/live/wave/gnat_ugx/html/gnat_ugx/gnat_ugx.html`
[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.
[Cen 2011]	CENELEC, *EN 50128:2011, Railway applications – Communications, signalling and processing systems – Software for railway control and protection systems*, 2011.

[CCRA 20xx] Common Criteria Recognition Arrangement, *Common Criteria Portal*, undated; `https://www.commoncriteriaportal.org/`

[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`

[ECSS 20xx] European Cooperation for Space Standardization, Web portal, undated; `https://www.ecss.nl`

[ECSS 2009] European Cooperation for Space Standardization, *ECSS-E-ST-40C: Space engineering / Software*; Noordwijk, The Netherlands; 6 March 2009.

[ECSS 2011a] European Cooperation for Space Standardization, *ECSS-HB-80-04A: Space product assurance / Software metrication programme definition and implementation;* Noordwijk, The Netherlands; 30 March 2011.

[ECSS 2011b] European Cooperation for Space Standardization, *ECSS-Q-HB-80-01A: Space Product assurance / Reuse of existing software;* Noordwijk, The Netherlands; 5 December 2011.

[ECSS 2013] European Cooperation for Space Standardization, *ECSS-E-HB-40A: Space engineering / Software engineering handbook;* Noordwijk, The Netherlands; 11 December 2013.

[ECSS 2017a] European Cooperation for Space Standardization, *ECSS-Q-ST-80C Rev. 1: Space product assurance / Software product assurance*; Noordwijk, The Netherlands; 15 February 2017.

[ECSS 2017b] European Cooperation for Space Standardization, *ECSS-HB-80-03A Rev.1: Space Product Assurance / Software dependability and safety;* Noordwijk, The Netherlands; 30 November 2017.

[ECSS 2020] European Cooperation for Space Standardization, *ECSS-HB-40-01A: Space engineering / Agile software development handbook;* Noordwijk, The Netherlands; 7 April 2020.

[ISO 2018] International Organization for Standardization, *ISO 26262:2018 Road vehicles – Functional safety*; 2018.

[JGMM 2002] M. Jones, E. Gomez, A. Mantineo, U.K. Mortensen; "Introducing ECSS Software-Engineering Standards within ESA – Practical approaches for space- and ground-segment software", in *ESA Bulletin 111*; August 2002; `http://www.esa.int/esapub/bulletin/bullet111/chapter21_bul111.pdf`

[KG 95] W. Kriedte and Y. El Gammal; "A New Approach to European Space Standards", in *ESA Bulletin 81*, February 1995; `https://www.esa.int/esapub/bulletin/bullet81/krie81.htm`

[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`

[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`

Index

Ada language **27**
- Abstract data types 35
- `Assertion_Error` exception 33
- Buffer overrun prevention ... 37
- Child units 34, 98
- Concurrent programming (tasks) 36
- Contract-based programming 30, 32, 37
- Dynamic memory management 87
- Encapsulation 98
- Generic templates.......... 34, 84
- High-integrity systems 37
- History and overview 30
- Memory management 93
- Object-Oriented Programming (OOP)........................ 35, 98
- Overview 28
- Portability 29, 108
- Postconditions 32
- **`pragma`** `Profile`38, 101, 103
- **`pragma`** `Restrictions`... 38, 70, 87, 101, 103
- Preconditions 32
- Programming in the large 34
- Real-Time Systems Annex 36
- Scalar ranges 31
- Separate compilation 84
- Support for ECSS standards . 39
- Systems Programming Annex 36
- Type / subtype predicates ... 33
- Type invariants.................... 33
- Usage 31

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.......... 78
- Tools and the software life cycle 49

Babbage, Charles 30
Buffer overrun 37
Byron, (Lord) George Gordon ... 30

C language 31
  Buffer overrun...................... 37
  Interfacing from Ada 29
C# language 31
C++ language....................... 31, 35
  Buffer overrun...................... 37
  Interfacing from Ada 29
Capgemini Engineering 40
Cert run-time library 60
CodePeer 51, 73, 90, 91, 92, 93, 94, 101, 102
  Common Weakness Enumeration (CWE) errors detected 55, 74
  Early error detection 73
  Support for ECSS Standards . 75
Coding standard
  Enforcement by GNATcheck 66
Common Criteria...................... 37
Common Weakness Enumeration (CWE) errors detected by CodePeer............................. 74
Common Weakness Enumeration (CWE) errors detected by SPARK Pro 55
CWE compatibility............... 55, 74

Data Validation 60
DO-178C...........37, 52, 61, 94, 100

Eclipse support.... See GNATbench
ECSS-E-HB-40-01A..................... 24
ECSS-E-HB-40A......................... 24
ECSS-E-ST-40C
  Software delivery and acceptance process 15
  Software design and implementation engineering process 14
  Software maintenance process 15
  Software management process 14
  Software operation process . 15
  Software related systems requirements process 13
  Software requirements and architecture engineering process 14
  Software validation process . 14
  Software verification process 15
ECSS-Q-HB-80-01A.................... 25
ECSS-Q-HB-80-03A Rev.1 25
ECSS-Q-HB-80-04A.................... 25
ECSS-Q-ST-80C
  Software process assurance. 23
  Software product assurance programme implementation 23
  Software product quality assurance 24
Embedded run-time library 60
EN 50128.................... 37, 52, 100
Encapsulation (software engineering principle) 34
European Cooperation for Space Standardization.................... 11
European Space Agency............ 11

Failure Mode and Effects Analysis (FMEA) 25
Fuzz testing 60, 76

GNAT Dynamic Analysis Suite .. 51, 76, 104, 105
  Support for ECSS Standards . 77
GNAT Pro Ada 51, 58, 100
  Support for ECSS Standards . 64

GNAT Pro Ada tools
    Support for ECSS Standards . 72
GNAT Pro Assurance 97, 100, 101, 106
    Sustained branch See Sustained branch
    Traceability analysis service See Traceability (Source to Object)
GNAT Pro Enterprise
    Run-Time Library options..... 59
GNAT Studio........................ 59, 61
GNATbench......................... 59, 63
GNATcheck......38, 66, 67, 92, 101, 102, 103, 104
    Coding standard enforcement 66
GNATcoverage ..51, 77, 88, 89, 90, 91, 93, 94, 102, 104, 105, 107
GNATdashboard...63, 99, 104, 107
GNATemulator.......................... 71
GNATmetric 67, 99, 104, 107
GNATprove 40, 53, 101
GNATstack 68, 95, 101
GNATstub................................. 82
GNATtest...76, 88, 89, 90, 91, 104, 105, 107
GNU GCC technology 59
gprbuild.............................. 88, 91

Hybrid verification 46

Ichbiah, Jean 30
Input Data
    validation 60
Inria... 40
Integrated Development
    Environments (IDEs)............. 61
ISO 26262................................. 52

Java language..................... 31, 35

Language Server Protocol (LSP) 63
LEON2 .... 12, 65
LEON3 .... 12, 65
Libadalang................................ 71
Light run-time library.... 55, 59, 65
Light-Tasking run-time library.. 60, 65
Liskov Substitution Principle (LSP) 76
Lovelace, Augusta Ada 30

Memory
    Fragmentation avoidance- ... 87
    Garbage collection 28, 35
    Storage leakage.................... 70
    Usage verification 68, 70
Metrication (software)
    ECSS-Q-HB-80-04A............... 25
    GNATmetric ... *See* GNATmetric
MISRA-C (generated by QGen) 51, 96
Model-based engineering..51, *See* QGen

Programming language (role in
    software design) 27

QEMU 71
QGen.....50, 51, 83, 86, 88, 92, 96, 102, 103
    Support for ECSS standards . 53
    TQL-1 qualification....... 52, 103
Qualification evidence (DO-178C) 101

Rapita Systems Ltd................... 70
Rate-Monotonic Analysis.......... 95

Ravenscar tasking profile ... 28, 37, 60, 95
Ravenscar-Cert run-time library 60
Reuse (software)
  Ada and safe reuse 84
  ECSS-Q-HB-80-01A 25

Simulink® 51
SPARK 40
  Absence of run-time exceptions 54
  Data and control flow analysis 54
  Generated by QGen 51
  Information flow analysis 54
  Static verification support 54
  Support for ECSS standards . 48
  Usage 40
SPARK adoption
  Bronze level 44
  Gold level 45
  Platinum level 46
  Silver level 45
  Stone level 44
SPARK Pro ...40, 50, 53, 90, 91, 92, 93, 95, 102, 108
  Support for ECSS Standards . 57
Specialized needs annexes (in Ada standard) 29
Stateflow® 51
Static Verification
  Soundness 41
Sustained branch ..61, 96, 97, 100, 101, 106

Taft, Tucker 30
Technical debt 64
Timing Verification 70
Tool Qualification Level (TQL) ... 52
Traceability (source to object code) 94
Traceability (Source to Object) . 61

Validation process (software)
  Definition in ECSS-E-ST-40C . 14
Verification process (software)
  Definition in ECSS-E-ST-40C . 15
Version skew (prevention in Ada) 84, 88

WCET Analysis 70
Workbench (WindRiver development environment) . 63

Zero Footprint (ZFP) run-time library 55, 60