LEARN.
ADACORE.COM

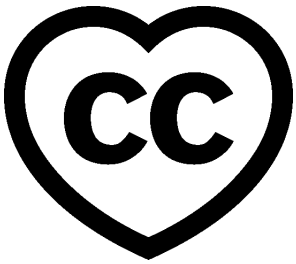
AdaCore Technologies for Space Systems Software

Version 2.1

Benjamin M. Brosgol
and Jean-Paul Blanquart

Mar 28, 2026

CONTENTS



[1] http://creativecommons.org/licenses/by-sa/4.0

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/ED - 12C, 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

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

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:

- 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 such as are found in the civil aviation and nuclear industries. In the space domain the ECSS software-related standards play a similar role, 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 since the mid 1990s, in domains that include space systems, commercial and military avionics, air traffic control, train systems, automotive, and medical devices. This document 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

Foreword to V2.1

In the years since the initial version of this document was published, both ECSS-E-ST-40C and ECSS-Q-ST-80C have been revised, and AdaCore's products have evolved to meet the growing demands for, and challenges to, high assurance in mission-critical real-time software. This new edition reflects the current (2025) versions of ECSS-E-ST-40C, ECSS-Q-ST-80C, and AdaCore's offerings. Among other updates and enhancements to the company's products, the static analysis tools supplementing the GNAT Pro development environment have been integrated into a cohesive toolset (the *GNAT Static Analysis Suite*). The dynamic analysis tools have likewise been consolidated, and the resulting *GNAT Dynamic Analysis Suite* has introduced a fuzzing tool — *GNATfuzz* — which exercises the software with invalid input and checks for failsafe behavior.

As editor of this revised edition, I would like to thank my AdaCore colleagues Olivier Appéré and Vasiliy Fofanov, as well as co-author Jean-Paul Blanquart, for their detailed and helpful reviews and suggestions.

For up-to-date information on AdaCore support for developing space software, please visit[43].

Benjamin M. Brosgol, AdaCore
Bedford, Massachusetts USA
December 2025

[43] AdaCore. AdaCore + Space. URL: https://www.adacore.com/industries/space.

CHAPTER

ONE

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[2]:

> *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[3], 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),
- *Space product assurance* (the "–Q" series), and
- *Space project management* (the "–M" series).

This document focuses on two specific standards:

- ECSS-E-ST-40C Rev. 1 (Space engineering / Software)[4], and
- ECSS-Q-ST-80C Rev. 2 (Space product assurance / Software product assurance)[5]

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. Unless noted otherwise, all references to ECSS-E-ST-40C and ECSS-Q-ST-80C in this document relate to these cited editions of the 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:

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

[3] European Cooperation for Space Standardization. ECSS Web Portal. URL: https://www.ecss.nl.

[4] *ECSS-E-ST-40C Rev.1 - Software*. ECSS, April 2025. URL: https://ecss.nl/standard/ecss-e-st-40c-rev-1-software-30-april-2025/.

[5] *ECSS-Q-ST-80C Rev.2 - Software product assurance*. ECSS, April 2025. URL: https://ecss.nl/standard/ecss-q-st-80c-rev-2-software-product-assurance-30-april-2025/.

- The ZFP (Zero Footprint) minimal run-time library for Ada on LEON2 ELF, qualified at criticality category B, for the aerospace company AVIO[6].
- The Ravenscar SFP (Small Footprint) QUAL run-time library for Ada on LEON2 and LEON3 boards, prequalified at criticality category B, for ESA[7].

An important update in the 2025 versions of ECSS-E-ST-40C and ECSS-Q-ST-80C is the explicit attention paid to security issues. Memory-safe languages like Ada and SPARK, and formal analysis tools such as SPARK Pro, help reduce the effort in demonstrating security properties in space software.

The remainder of this chapter summarizes the ECSS-E-ST-40C and ECSS-Q-ST-80C standards, and the subsequent chapters have the following content:

- *Programming Languages for Space Software* (page 15) describes the Ada and SPARK programming languages and relates their software engineering support to the relevant sections / requirements in the two standards.
- Analogously, *Tools for Space Software Development* (page 29) 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, *Compliance with ECSS-E-ST-40C* (page 49) 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.
- *Compliance with ECSS-Q-ST-80C* (page 63) does likewise for the requirements in ECSS-Q-ST-80C.
- For ease of reference, the *Abbreviations* (page 69) chapter contains a table of acronyms and initialisms used in this document, and bibliography lists the various resources cited.

Although this document is focused on specific ECSS standards, the *Programming Languages for Space Software* (page 15) and *Tools for Space Software Development* (page 29) chapters explain how the Ada and SPARK languages / technologies and AdaCore's products benefit software development in general for large-scale safety-critical systems. These chapters may thus be applicable to software that has to comply with regulatory standards in other domains.

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

As stated in ECSS-E-ST-40C (Page 5, 4, p. 11):

> "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 defines the following space system software engineering processes:

- Software-related systems requirements process (§ 4.2.2)

[6] AdaCore. Press Release: AVIO Selects AdaCore's GNAT Pro Assurance Toolsuite for European Space Agency Program. January 8, 2019. URL: https://www.adacore.com/press/avio-selects-adacores-gnat-pro-assurance-toolsuite-for-european-space-agency-program.

[7] AdaCore. Press Release: European Space Agency Selects AdaCore's Qualified-Multitasking Solution for Spacecraft Software Development. September 24, 2019. URL: https://www.adacore.com/press/european-space-agency-selects-adacores-qualified-multitasking-solution-for-spacecraft-software-development.

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" (Page 5, 4, p. 27).

- Software management process (§ 4.2.3)

  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" (Page 5, 4, pp. 27, 28). It covers the joint review process, interface management, and technical budget and margin management.

- Software requirements and architecture engineering process (§ 4.2.4)

  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 (§ 4.2.5)

  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 (§ 4.2.6)

  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" (Page 5, 4, p. 29).

- Software delivery and acceptance process (§ 4.2.7)

  This process "prepares the software product for delivery and testing in its operational environment" (Page 5, 4, p. 29).

- Software verification process (§ 4.2.8)

  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." (Page 5, 4, p. 30)

- Software operation process (§ 4.2.9)

  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." (Page 5, 4, p. 30)

- Software maintenance process (§ 4.2.10)

  This process "covers software product modification to code or associated documentation for correcting an error, a problem or implementing an improvement or adaptation." (Page 5, 4, p. 31)

- Software security process (§ 4.2.11)

  This process "is supported by a software security analysis that is systematically maintained at different points in the lifecycle of the software.... The software security analysis is used to ensure that security risks are properly addressed.... It is also used to assess and drive the design, implementation and operation of secure software." (Page 5, 4, p. 32)

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).

The tables below, derived from Table A-1 in Annex A of ECSS-E-ST-40C, show the association between files, DRL items, and reviews. Cells with "E" indicate requirements from ECSS-E-ST-40C, and cells with "Q" are the contributions from ECSS-Q-ST-80C.

Table 1: **Relationship between RB (Requirements Baseline), DRL items, and reviews**

DRL Item	SRR	PDR	CDR	QR	AR	ORR
Software System Specification	E					
Interface requirements document (IRD)	E					
Safety and dependability analysis results for lower level suppliers	E Q					

Table 2: **Relationship between TS (Technical Specification), DRL items, and reviews**

DRL Item	SRR	PDR	CDR	QR	AR	ORR
Software requirements specification (SRS)		E				
Software interface control document (ICD)		E	E			

Table 3: **Relationship between DDF (Design Definition File), DRL items, and reviews**

DRL Item	SRR	PDR	CDR	QR	AR	ORR
Software design document (SDD)		E	E			
Software configuration file (SCF)		E	E	E Q	E	E Q
Software release document (SRelD)				E	E	
Software user manual (SUM)			E	E	E	
Software source code and media labels			E			
Software product and media labels				E	E	E
Training material				E		

Table 4: **Relationship between DJF (Design Justification File), DRL items, and reviews**

DRL Item	SRR	PDR	CDR	QR	AR	ORR
Software verification plan (SVerP)		E				
Software validation plan (SValP)		E				
Independent software verification and validation plan	E Q	E				
Software integration test plan (SITP)		E	E			
Software unit test plan (SUTP)			E			
Software validation specification (SVS) with respect to TS			E			
Software validation specification (SVS) with respect to RB				E	E	
Acceptance test plan				E	E	
Acceptance test report			E			
Installation report			E			
Software verification report (SVR)	E	E	E	E	E	E Q
Independent software verification and validation report		E Q	E Q	E Q	E Q	E
Software reuse file (SRF)	E Q	E	E			
Software problems reports and nonconformance reports	E Q	E Q	E Q	E Q	E Q	E Q
Joint review reports	E	E	E	E	E	
Justification of selection of operational ground equipment and support services	E Q	E Q				

Table 5: **Relationship between MGT (Management File), DRL items, and reviews**

DRL Item	SRR	PDR	CDR	QR	AR	ORR
Software development plan (SDP)	E	E				
Software review plan (SRP)	E	E				
Software configuration management plan	E	E				
Training plan	E Q					
Interface management procedures	E					
Identification of NRB SW and members	E Q					
Procurement data	E Q	E Q				

Table 6: **Relationship between MF (Maintenance File), DRL items, and reviews**

DRL Item	SRR	PDR	CDR	QR	AR	ORR
Maintenance plan				E	E	E
Maintenance records	Q	Q	Q	E Q	E Q	E Q
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						

Table 7: **Relationship between OP (Operational Plan), DRL items, and reviews**

DRL Item	SRR	PDR	CDR	QR	AR	ORR
Software operation support plan						E
Operational testing results						E
SPR and NCR- User's request record- Post operation review report						E

Table 8: **Relationship between PAF (Product Assurance File), DRL items, and reviews**

DRL Item	SRR	PDR	CDR	QR	AR	ORR
Software product assurance plan (SPAP)	E Q	E Q	E Q	E Q	E Q	E Q
Software product assurance requirements for suppliers	E Q	Q	Q	Q	Q	Q
Audit plan and schedule	E Q	Q	Q	Q	Q	Q
Review and inspection plans or procedures	Q	Q	Q	Q	Q	Q
Procedures and standards	Q	E Q	Q	Q	Q	Q
Modelling and design standards	E Q	E Q	Q	Q	Q	Q
Coding standards and description of tools	Q	E Q	Q	Q	Q	Q
Software problem reporting procedure	Q	E Q	Q	Q	Q	Q
Software dependability and safety analysis report- Criticality classification of software components	Q	E Q	E Q	E Q	E Q	Q
Software product assurance report	Q	Q	Q	Q	Q	Q
Software product assurance milestone report (SPAMR)	E Q	E Q	E Q	E Q	E Q	E Q
Statement of compliance with test plans and procedures	Q	Q	E Q	E Q	E Q	E Q
Records of training and experience	Q	Q	Q	Q	Q	Q
(Preliminary) alert information	Q	Q	Q	Q	Q	Q
Results of preaward audits and assessments, and of procurement sources	Q	Q	Q	Q	Q	Q
Software process assessment plan	Q	Q	Q	Q	Q	Q
Software process assessment records	Q	Q	Q	Q	Q	Q
Review and inspection reports	Q	Q	Q	Q	Q	Q
Receiving inspection report	E Q	E Q	E Q	E Q	Q	Q
Input to product assurance plan for systems operation	Q	Q	Q	Q	Q	E Q

Table 9: **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)	E					
RB	Interface requirements document (IRD)	E					
RB	Safety and dependability analysis results for lower level suppliers	E Q					
TS	Software requirements specification (SRS)		E				
TS	Software interface control document (ICD)		E	E			
DDF	Software design document (SDD)		E	E			
DDF	Software configuration file (SCF)		E	E	E Q	E	E Q

continues on next page

Table 9 – continued from previous page

File	DRL Item	SRR	PDR	CDR	QR	AR	ORR
DDF	Software release document (SRelD)				E	E	
DDF	Software user manual (SUM)			E	E	E	
DDF	Software source code and media labels			E			
DDF	Software product and media labels				E	E	E
DDF	Training material				E		
DJF	Software verification plan (SVerP)		E				
DJF	Software validation plan (SValP)		E				
DJF	Independent software verification and validation plan	E Q	E				
DJF	Software integration test plan (SUITP)		E	E			
DJF	Software unit test plan (SUITP)			E			
DJF	Software validation specification (SVS) with respect to TS			E			
DJF	Software validation specification (SVS) with respect to RB				E	E	
DJF	Acceptance test plan				E	E	
DJF	Software unit test report			E			
DJF	Software integration test report			E			
DJF	Software validation report with respect to TS			E			
DJF	Software validation report with respect to RB				E	E	
DJF	Acceptance test report					E	
DJF	Installation report					E	
DJF	Software verification report (SVR)	E	E	E	E	E	E Q
DJF	Independent software verification and validation report		E Q	E Q	E Q	E Q	E
DJF	Software reuse file (SRF)	E Q	E	E			
DJF	Software problems reports and nonconformance reports	E Q	E Q	E Q	E Q	E Q	E Q
DJF	Joint review reports	E	E	E	E	E	
DJF	Justification of selection of operational ground equipment and support services	E Q	E Q				
MGT	Software development plan (SDP)	E	E				
MGT	Software review plan (SRevP)	E	E				
MGT	Software configuration management plan	E	E				
MGT	Training plan	E Q					
MGT	Interface management procedures	E					
MGT	Identification of NRB SW and members	E Q					
MGT	Procurement data	E Q	E Q				
MF	Maintenance plan				E	E	E
MF	Maintenance records	Q	Q	Q	E Q	E Q	E Q
MF	SPR and NCR- Modification analysis report- Problem analysis report- Modification documentation- Baseline for change - Joint review reports						
MF	Migration plan and notification						
MF	Retirement plan and notification						
OP	Software operation support plan						E
OP	Operational testing results						E
OP	SPR and NCR- User's request record- Post operation review report						E
PAF	Software product assurance plan (SPAP)	E Q	E Q	E Q	E Q	E Q	E Q

continues on next page

Table 9 – continued from previous page

File	DRL Item	SRR	PDR	CDR	QR	AR	ORR
PAF	Software product assurance requirements for suppliers	E Q	Q	Q	Q	Q	Q
PAF	Audit plan and schedule	E Q	Q	Q	Q	Q	Q
PAF	Review and inspection plans or procedures	Q	Q	Q	Q	Q	Q
PAF	Procedures and standards	Q	E Q	Q	Q	Q	Q
PAF	Modelling and design standards	E Q	E Q	Q	Q	Q	Q
PAF	Coding standards and description of tools	Q	E Q	Q	Q	Q	Q
PAF	Software problem reporting procedure	Q	E Q	Q	Q	Q	Q
PAF	Software dependability and safety analysis report- Criticality classification of software components	Q	E Q	E Q	E Q	E Q	Q
PAF	Software product assurance report	Q	Q	Q	Q	Q	Q
PAF	Software product assurance milestone report (SPAMR)	E Q	E Q	E Q	E Q	E Q	E Q
PAF	Statement of compliance with test plans and procedures	Q	Q	E Q	E Q	E Q	E Q
PAF	Records of training and experience	Q	Q	Q	Q	Q	Q
PAF	(Preliminary) alert information	Q	Q	Q	Q	Q	Q
PAF	Results of preaward audits and assessments, and of procurement sources	Q	Q	Q	Q	Q	Q
PAF	Software process assessment plan	Q	Q	Q	Q	Q	Q
PAF	Software process assessment records	Q	Q	Q	Q	Q	Q
PAF	Review and inspection reports	Q	Q	Q	Q	Q	Q
PAF	Receiving inspection report	E Q	E Q	E Q	E Q	Q	Q
PAF	Input to product assurance plan for systems operation	Q	Q	Q	Q	Q	E Q

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[Page 5, 5], 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's lifetime. In particular, that the software is developed to perform properly, securely, and safely in its operational environment, meeting the project's agreed quality objectives.*

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. The table above includes this information for the requirements in ECSS-Q-ST-80C.

The ECSS standards recognize that software systems (and different components of the same software system) may vary in their effects on system safety. The standards accordingly define several criticality categories, denoted A (most critical) to D, which correspond closely to the software levels in the airborne standard DO - 178C/ED - 12C.

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.

Several handbooks complement ECSS-E-ST-40C, including:

- ECSS-E-HB-40A (Software engineering handbook)[8],

  This document provides guidance, explanations, and examples on how to satisfy the ECSS-E-ST-40C requirements in practice.

- ECSS-E-HB-40-01A (Agile software development handbook)[9].

  This handbook shows how to reconcile agile development practices with the formal ECSS space software engineering processes.

- ECSS-E-HB-40-02A (Machine learning handbook)[10].

  This handbook provides guidelines on how to create reliable machine learning functions and perform the verification and validation considering the specifics of machine learning development practices.

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

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

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

[10] *ECSS-E-HB-40-02A: Space engineering / Machine learning handbook*. ECSS. Noordwijk, The Netherlands; 15 November 2024.

- ECSS-Q-HB-80-01A (Reuse of existing software)[11]

  This handbook offers guidance on software reuse (including software tools) and also presents a Tool Qualification Level (TQL) concept based on DO - 178C/ED - 12C[12], DO - 330/ED - 215[13], and ISO 26262[14].

- ECSS-Q-HB-80-03A Rev. 1 (Software dependability and safety)[15]

  This handbook 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)[16]

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

[11] *ECSS-Q-HB-80-01A: Space Product assurance / Reuse of existing software*. ECSS. Noordwijk, The Netherlands; 5 December 2011.
[12] *DO-178C/ED-12C: Software Considerations in Airborne Systems and Equipment Certification*. RTCA and EUROCAE, December 2011.
[13] *DO-330/ED-215: Software Tool Qualification Considerations*. RTCA and EUROCAE, December 2011.
[14] *ISO 26262:2018 Road vehicles – Functional safety*. International Organization for Standardization, 2018.
[15] *ECSS-Q-HB-80-03A Rev.1: Space Product Assurance / Software dependability and safety*. ECSS. Noordwijk, The Netherlands; 30 November 2017.
[16] *ECSS-Q-HB-80-04A: Space product assurance / Software metrication programme definition and implementation*. ECSS. Noordwijk, The Netherlands; 30 March 2011.

CHAPTER

TWO

PROGRAMMING LANGUAGES FOR SPACE SOFTWARE

This chapter explains how space software developers can benefit from the Ada language and its formally analyzable SPARK subset ([17],[18]). Unless explicitly stated otherwise, the Ada discussion applies to the Ada 2012 version of the language standard with Technical Corrigendum 1 ([19],[20],[21]), and the SPARK language discussion applies to the 2014 version of the language as amended by[17]. Both Ada and SPARK support the development and verification of software at the highest criticality class levels.

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. With their standard features for preventing buffer overruns, pointer misuses and other vulnerabilities, Ada and SPARK both serve as memory-safe languages.

2.1.1 Ada language overview

Ada is multi-faceted. From one perspective it is a classical stack-based general-purpose language (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;

[17] AdaCore and Capgemini Engineering. *SPARK Reference Manual*. URL: https://docs.adacore.com/live/wave/spark2014/html/spark2014_rm/index.html.

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

[19] ISO/IEC JTC1/SC22. *Ada Reference Manual, Language and Standard Libraries*. International Organization for Standardization, 2016. ISO/IEC 8652:2012(E) with Technical Corrigendum 1. URL: https://www.adacore.com/documentation/ada-2012-reference-manual.

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

[21] John Barnes and Ben Brosgol. *Safe and Secure Software, an invitation to Ada 2012*. 2015. URL: https://www.adacore.com/books/safe-and-secure-software.

- flexible data composition facilities;
- strong type checking;
- traditional features for code modularization (“subprograms”);
- 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 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. The language's dynamic semantics 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, and other enhancements to increase the language's expressiveness were introduced in Ada 2022[22].

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 long been a language of choice in high-assurance / safety-critical / high-security domains, where software failures can have catastrophic consequences. Ada has a proven track record in space applications, military and commercial aircraft avionics, air traffic control, and railroad software, and it has also seen successful usage in other domains (such as automotive systems and medical devices).

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 also been the subject of significant research, especially in the area of real-time technologies. Throughout much of Ada's history, a worldwide group of experts from industry, academia, and government convened a biannual meeting — the *International Real-Time Ada Workshop* (*IRTAW*) — to focus on Ada's support for real-time software. The 1997 IRTAW in Ravenscar, UK, led directly to the definition of a subset of Ada's concurrency features with deterministic semantics — the so-called *Ravenscar Profile*[23]. This work broke new ground in supporting the use of concurrent programming in high-assurance software.

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, reflecting a 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.

Here's an example of an integer scalar range and an associated run-time check:

```
declare
   My_Score : Integer range 1..100;
   N        : Integer;
begin
   ...  -- Code that assigns an integer value to N
```

(continues on next page)

[22] ISO/IEC JTC1/SC22. *Ada Reference Manual, 2022 Edition, Language and Standard Libraries*. International Organization for Standardization, 2022. URL: https://www.adacore.com/documentation/ada-2022-reference-manual.

[23] Alan Burns, Brian Dobbing and Tullio Vardanega. Guide for the use of the Ada Ravenscar Profile in high integrity systems. Technical Report YCS-2003-348, University of York, January 2003. URL: https://www.open-std.org/jtc1/sc22/wg9/n424.pdf.

(continued from previous page)

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

The run-time check can be optimized out if the compiler can guarantee that the value of `N` is within the range 1 through 100 when control reaches the assignment to `My_Score`.

2.1.4 Contract-based programming

A feature introduced in 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; type contracts are used for invariants; and subtype contracts provide generalized constraints (predicates). Through contracts the developer can formalize the intended behavior of the application, and can verify this behavior by testing, static analysis or formal proof.

Here's a skeletal example that illustrates contract-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;

package body Table_Pkg is
   ... -- Implementation of Is_Full, Contains, Insert, Remove
end Table_Pkg;
```

A compiler option controls whether the pre- and postconditions are checked at run time. If checks are enabled, any pre- or postcondition failure — i.e., the contract's Boolean expression evaluating to **`False`** — raises the `Assertion_Error` exception.

Ada's type invariants and type / subtype predicates 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 below 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;
```

(continues on next page)

(continued from previous page)

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

Further information about type invariants and type / subtype predicates may be found in the *Design by contracts* chapter of the *Introduction to Ada* course of[24].

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 data structure (e.g., a stack, FIFO queue, or varying-length array) for an arbitrary element type T, where each element of the data structure is of type T. For safety and efficiency, compile-time checks should enforce type safety both within the parameterized module, and at each use (*instantiation*). Conceptually an instantiation can be regarded as an expansion of the parameterized module, with actual parameters replacing the formal parameters. However, the expansion is not at the lexical/syntactic level (source text) but rather at the semantic level (scope-resolved names). In addition, an implementation may be able to share a single copy of the code across multiple instantiations and thereby save code space.

Ada supplies this functionality through a facility known as *generics*, with type consistency enforced by the compiler both within the generic unit and at each instantiation. Ada generics are analogous to C++ templates but with more extensive compile-time checking: a data object within a generic unit can only be processed using operations that are known to be available for the object's type. (As a historical note, at the ACM SIGPLAN Conference on History of Programming Languages in 1993, the C++ designer Bjarne Stroustrup acknowledged the Ada design for generics as one of the inspirations for templates in C++[25].)

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

[24] AdaCore. Online training for Ada and SPARK. URL: https://learn.adacore.com/.

[25] Association for Computing Machinery (ACM). *HOPL-II: The Second ACM SIGPLAN Conference on History of Programming Languages*, 1993. URL: https://dl.acm.org/doi/proceedings/10.1145/154766.

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[Page 15, 20] or[26] 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. Protected objects provide the functionality of semaphores and condition variables but more clearly and reliably (e.g., avoiding subtle race conditions).

Ada supports asynchronous task interactions for timeouts, software event notifications, 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 multicore 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 Time and Space Analysis

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.

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

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.

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 in *GNATstack* (page 36).

2.1.12 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, DO - 178C/ED - 12C for avionics[27], and CENELEC EN 50128 for rail systems[28]. 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[29]. Key to Ada's support for high-assurance software is the language's memory safety; this is illustrated by a number of features, including:

- Strong typing

  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. (Low-level code sometimes needs to defeat the language's type checking; e.g., by treating *raw bits* as a value of a specific type. As a language that supports systems programming, Ada allows such usages but requires explicit syntax that makes the intent clear.)

- Array bounds checking

  A run-time check guarantees that an array index is within the bounds of the array. This prevents buffer overrun vulnerabilities that are common in C and C++. In many cases a compiler optimization can detect statically that the index is within bounds and thus eliminate any run-time code for the check.

- Prevention of null pointer dereferences

  As with array bounds, pointer dereferences are checked to make sure that the pointer is not null. Again, such checks can often be optimized out.

- Prevention of dangling references

  A scope accessibility check ensures that a pointer (known in Ada as an *access value*) cannot reference an object on the stack after exit/return from the scope (block or subprogram) in which the object is declared. Such checks prevent dangling references to stack objects (i.e., subprogram parameters and local variables) and are generally static, with no run-time overhead. (Since Ada does not require garbage collection of inaccessible dynamically allocated objects, it provides an `Unchecked_Deallocation` facility for programmer-supplied deallocation. Uses of this feature require explicit syntax — a `with` clause at the beginning of the compilation unit. This makes clear that

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

[28] *EN 50128/A2: Railway applications – Communications, signalling and processing systems – Software for railway control and protection systems*. CENELEC, July 2020.

[29] *Common Criteria for Information Technology Security Evaluation (ISO/IEC 15408)*. Common Criteria Development Board, 2022. URL: https://www.commoncriteriaportal.org/.

special attention needs to be paid to verification, since erroneous deallocations can lead to dangling references.)

The evolution of Ada has seen a continued increase in support for safety-critical and high-security applications. Ada 2005 standardized the Ravenscar Profile[Page 17, 23], 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 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.

The most recent version of the standard, Ada 2022, has added several contract-based programming constructs inspired by SPARK (`Contract_Cases`, `Global`, and `Depends` aspects) and, more generally, has enhanced the language's expressiveness. For example, Ada 2022 has introduced some new syntax in its concurrency support and has defined the Jorvik tasking profile, which is more inclusive than Ravenscar.

2.1.13 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 constructs that the compiler will reject. Sample restrictions include dependence on particular packages or language facilities, or usage of features requiring unwanted implementation techniques (e.g. run-time support that is overly complex and difficult to certify).

- 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 *GNATcheck* (page 42)) that is included with the GNAT Pro Ada Static Analysis Suite. 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.

2.1.14 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. Please see *Compliance with ECSS-E-ST-40C* (page 49) and *Compliance with ECSS-Q-ST-80C* (page 63) to learn how specific clauses in these standards relate to individual Ada features. In summary, the use of Ada can help the software supplier meet the requirements in 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
    - §5.9 Software operation process
        * §5.9.4 Software operation support
    - §5.10 Software maintenance process
        * §5.10.4 Modification implementation
    - §5.11 Software security process
        * § 5.11.2 Process implementation
        * § 5.11.3 Software security analysis
        * § 5.11.5 Security activities in the software life cycle
    - Annex U - Software code verification
- ECSS-Q-ST-80C
    - §6.2 Requirements applicable to all software engineering processes
        * §6.2.3 Handling of critical software
        * §6.2.9 Software security
        * §6.2.10 Handling of security sensitive 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 (Page 15, 17, Page 15, 18) is a software development technology (programming language and verification toolset) specifically oriented around applications demanding an ultra-low defect level, for example where safety and/or security are key requirements. SPARK Pro is the commercial-grade offering of the SPARK technology developed by AdaCore, Capgemini Engineering (formerly Altran), and Inria. As will be described in *Static Verification: SPARK Pro* (page 30), 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 systems, and cross-domain solutions. SPARK 2014 is the most recent version of the technology.

The SPARK language has been stable over the years, with periodic enhancements. The 2014 version of SPARK represented a major revision, incorporating contract-based programming syntax from Ada 2012, and subsequent upgrades included support for pointers (access types) based on the Rust ownership model.

The SPARK language is a large subset of Ada 2012 and is memory safe. 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, pointer functionality that may be unsafe, and most tasking features.

A legal SPARK program has unambiguous semantics: its effect is precisely defined and does not depend on the implementation. This property helps ensure 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. Moreover, the SPARK tools achieve soundness while keeping *false positives* manageable.

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 (embodying mathematics-based formal methods) 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.

Here is an example of SPARK code:

```
package Example is
   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;
end Example;

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

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

User experience has shown that the language and the SPARK Pro toolset do not require a steep learning curve. Training material such as AdaCore's online AdaLearn course for SPARK[Page 19, 24] can quickly bring developers up to speed; users are assumed to be experts in their own application domain such as space technology and do not need to be familiar with formal methods or the proof approaches implemented by the toolset. In effect, SPARK Pro is an advanced static analysis tool that will detect many logic errors and security vulnerabilities very early in the software life cycle. It can be smoothly integrated into an organization's existing development and verification methodology and infrastructure.

SPARK uses the standard Ada 2012 contract syntax, which both simplifies the learning process and also allows new paradigms of software verification. Programmers familiar with writing executable contracts for run-time assertion checking can use the same approach but with additional flexibility: the contracts can be verified either dynamically through classical run-time testing methods or statically (i.e., pre-compilation and pre-test) using automated tools.

2.2.3 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[30], 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.

2.2.3.1 Stone level: Valid SPARK

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

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

2.2.3.2 Bronze level: Initialization and correct data flow

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

2.2.3.3 Silver level: Absence of run-time errors (AORTE)

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.

2.2.3.4 Gold level: Proof of key integrity properties

At the Gold level, the proof tool will verify properties such as correctness 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.

2.2.3.5 Platinum level: Full functional correctness

At the Platinum level, the algorithmic code is proved to satisfy its formally specified functional requirements. This is still a challenge in practice for realistic programs but may be appropriate for small critical modules, especially for security-critical systems at high Evaluation Assurance Levels where formal methods can provide the needed confidence.

2.2.4 Hybrid Verification

Hybrid verification combines formal methods with traditional testing. A typical scenario is an in-progress project that is based on testing but which 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 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; here is an example.

```
function getascii return Interfaces.C.unsigned_char
   with Post => getascii'Result in 0..127;

pragma Import (C, getascii);
-- Interfaces.C.unsigned_char is a modular (unsigned)
-- integer type, typically ranging from 0 through 255

procedure Example is
```

(continues on next page)

(continued from previous page)

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

2.2.5 SPARK and the ECSS Standards

In summary, the qualities that make Ada an appropriate choice for space software also apply to SPARK (see *Ada and the ECSS Standards* (page 22)). And specific to SPARK, 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 application is sufficiently safe and secure.

The use of SPARK can help the software supplier meet the requirements in the following sections of ECSS-E-ST-40C and ECSS-Q-ST-80C:

- 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
  - §5.11 Software security process
    - § 5.11.2 Process implementation
    - § 5.11.3 Software security analysis

        - § 5.11.5 Security activities in the software life cycle
    - Annex U Software code verification
- ECSS-Q-ST-80C
    - §6.2 Requirements applicable to all software engineering processes
        - §6.2.3 Handling of critical software
        - §6.2.9 Software security
        - §6.2.10 Handling of security sensitive 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

Note in particular that the SPARK language directly addresses several criteria in ECSS-Q-ST-80C requirement 6.2.3.2a:

- "use of a 'safe subset' of programming language"
- "use of formal design language for formal proof"

Further details on how SPARK and SPARK Pro can contribute to compliance with the ECSS standards are presented in *Compliance with ECSS-E-ST-40C* (page 49) and *Compliance with ECSS-Q-ST-80C* (page 63).

CHAPTER

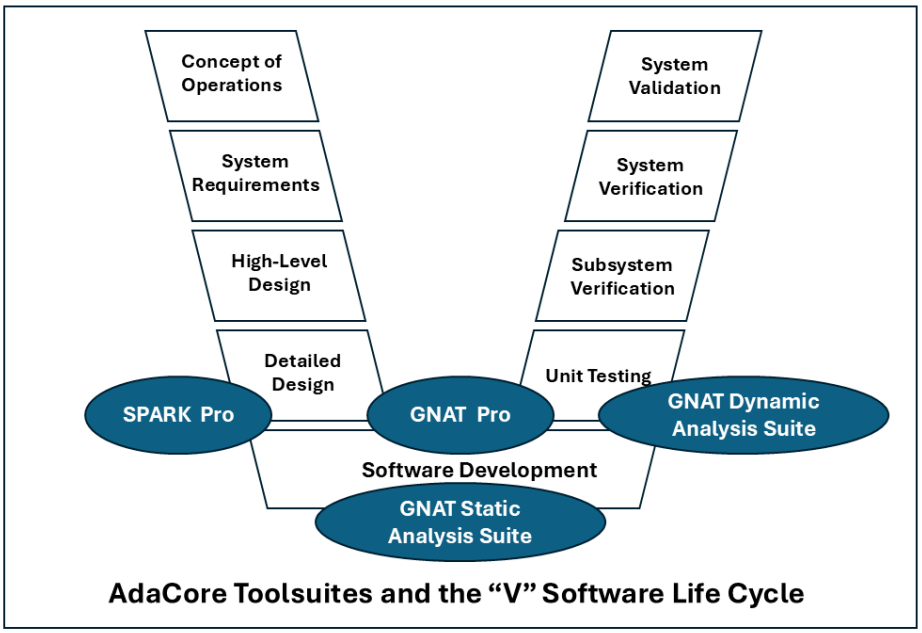
THREE

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 the figure below 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.



AdaCore Toolsuites and the "V" Software Life Cycle

As can be seen in the figure, AdaCore's toolsuites apply towards the bottom of the *V*. In summary:

- The SPARK Pro static analysis toolsuite (see *Static Verification: SPARK Pro* (page 30)) applies during Detailed Design and Software Development. It includes a proof tool that verifies properties ranging from correct information flows to functional correctness.

- The GNAT Pro development environment (see *GNAT Pro Development Environment* (page 33)) applies during Detailed Design, Software Development, and Unit Testing. It consists of gcc-based program build tools, an integrated and tailorable graphical user interface, accompanying tools, and a variety of supplemental libraries (including some for which qualification material is available for ECSS-E-ST-40C and ECSS-Q-ST-80C.)
- The GNAT Static Analysis Suite (see *Static Verification: GNAT Static Analysis Suite (GNAT SAS)* (page 40)) applies during Software Development. It contains a variety of tools for Ada, including a vulnerability detector that can be used retrospectively to detect issues in existing codebases and/or during new projects to prevent errors from being introduced.
- The GNAT Dynamic Analysis Suite (see *GNAT Dynamic Analysis Suite (GNAT DAS)* (page 44)) applies during Software Development and Unit Testing. One of these tools, GNATcoverage, supports code coverage and reporting at various levels of granularity for both Ada and C. It can be used during unit and integration testing.

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

3.2 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.

3.2.1 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 *Levels of Adoption of Formal Methods* (page 25)), the SPARK technology can be introduced incrementally into a project, based on the assurance requirements. Each level, from Bronze to Platinum, comes with associated benefits and costs.

3.2.2 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. 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.

3.2.3 CWE Compatibility

SPARK Pro detects a number of dangerous software errors in The MITRE Corporation's Common Weakness Enumeration (CWE), and the tool has been certified by the MITRE Corporation as a "CWE-Compatible" product[31].

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

Table 10: SPARK Pro and the CWE

CWE Weakness	Description
CWE 119, 120, 123, 126, 127, 129, 130,	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,	Buffer access errors
CWE 820	Missing synchronization
CWE 821	Incorrect synchronization
CWE 822, 823, 824,	Pointer errors
CWE 835	Infinite loop

3.2.4 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.

[31] The MITRE Corp. CWE-Compatible Products and Services. URL: https://cwe.mitre.org/compatible/compatible.html.

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 detailed discussion of this issue and its mitigation may be found in[32].) Such implementation dependencies are either prohibited in SPARK and thus detected by SPARK Pro, or else they do not affect the computed result. In either case the use of SPARK Pro eases the effort in porting the code from one environment to another.

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 *SPARK and the ECSS Standards* (page 27)) that relate to the SPARK language, together with the following:

- 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
  - §5.6 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
  - §5.10 Software maintenance process
    - §5.10.4 Modification implementation
  - §5.11 Software security process
    - §5.11.2 Process implementation
    - §5.11.3 Software security analysis
    - §5.11.5 Security activities in the software life cycle
  - Annex U - Software code verification
- 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
    - §6.2.9 Software security
    - §6.2.10 Handling of security sensitive software
  - §6.3 Requirements applicable to individual software engineering processes or activities
    - §6.3.4 Coding
  - §7.1 Product quality objectives and metrication
    - §7.1.3 Assurance activities for product quality requirements

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

  - §7.2 Product quality requirements
    - §7.2.3 Test and validation documentation

Details are provided in chapters *Compliance with ECSS-E-ST-40C* (page 49) and *Compliance with ECSS-Q-ST-80C* (page 63).

3.3 GNAT Pro Development Environment

This section summarizes the main features of the two editions of AdaCore's GNAT Pro language toolsuite, *Enterprise* and *Assurance*. These editions correspond to different levels of customer requirements and are available for Ada, C, C++, and Rust.

Based on the GNU GCC technology, GNAT Pro for Ada supports the Ada 83, Ada 95, Ada 2005, and Ada 2012 standards, as well as selected features of Ada 2022. It includes:

- several Integrated Development Environments (See *Integrated Development Environments (IDEs)* (page 37));
- a comprehensive toolsuite including a stack analysis tool (see *GNATstack* (page 36)) and a visual debugger;
- a library that enables customers to develop their own source analysis tools for project-specific needs (see *Libadalang* (page 35)); and
- other useful libraries and bindings.

For details on the tools and libraries supplied with GNAT Pro for Ada, see[33] and[34].

Other GNAT Pro products handle multiple versions of C (from C89 through C18), C++ (from C++98 through C++17), and Rust.

3.3.1 GNAT Pro Enterprise

GNAT Pro Enterprise is a development environment for producing critical software systems where reliability, efficiency, and maintainability are essential. Several features of GNAT Pro for Ada are noteworthy:

Run-Time Library Options

The product allows 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/or storage-constrained embedded applications. It supersedes the ZFP (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 and Jorvik tasking profiles. 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.

[33] AdaCore. GNAT User's Guide for Native Platforms. URL: https://docs.adacore.com/live/wave/gnat_ugn/html/gnat_ugn/gnat_ugn.html.
[34] AdaCore. GNAT User's Guide Supplement for Cross Platforms. URL: https://docs.adacore.com/live/wave/gnat_ugx/html/gnat_ugx/gnat_ugx.html.

Although limited in terms of dynamic Ada semantics, these predefined libraries fully support static Ada constructs such as private types, generic templates, and child units. Some dynamic semantics are also supported. For example, tagged types (at library level) and other Object-Oriented Programming features are supported, as is dynamic dispatching. The general use of dynamic dispatching at the application level can be prevented through pragma `Restrictions`.

Details on these libraries may be found in the "Predefined GNAT Pro Run-Times" chapter ofPage 33, 34.

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.

Run-Time Library Configurability

A traditional problem with predefined profiles is their inflexibility: if a feature outside a given profile is needed, then it is the developer's responsibility to address the certification issues deriving from its use. GNAT Pro for Ada accommodates this need by allowing the developer to define a profile for the specific set of features that are used. Typically this will be for features with run-time libraries that require associated certification materials. Thus the program will have a tailored run-time library supporting only those features that have been specified.

More generally, the configurable run-time capability allows specifying support for Ada's dynamic features in an à la carte fashion ranging from none at all to full Ada. The units included in the executable may be either a subset of the standard libraries provided with GNAT Pro, or specially tailored to the application. This latter capability is useful, for example, if one of the predefined profiles implements almost all the dynamic functionality needed in an existing system that has to meet new safety-critical requirements, and where the costs of adapting the application without the additional run-time support are considered prohibitive.

Enhanced Data Validity Checking

Improper or missing 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.

3.3.2 GNAT Pro Assurance

GNAT Pro Assurance extends GNAT Pro Enterprise with specialized support, including 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.

3.3.2.1 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.

3.3.2.2 Source to Object Traceability

Source-to-object traceability is required in standards such as DO - 178C/ED - 12C, 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.

3.3.2.3 Compliance with the ECSS Standards

Supplementing the support provided by GNAT Pro for Ada (see *GNAT Pro and the ECSS Standards* (page 39)), GNAT Pro Assurance helps compliance with the following requirements from ECSS-E-ST-40C and ECSS-Q-ST-80C:

- ECSS-E-ST-40C
  - §5.9 Software operation process
    - §5.9.2 Process implementation
    - §5.9.4 Software operation support
    - §5.9.5 User support
- ECSS-Q-ST-80C
  - §6.2 Requirements applicable to all software engineering processes
    - §6.2.6 Verification
  - §6.3 Requirements applicable to individual software engineering processes or activities
    - §6.3.9 Maintenance

3.3.3 Libadalang

Libadalang is a library included with GNAT Pro that gives applications access to the complete syntactic and semantic structure of an Ada compilation unit. This library is typically used by tools that need to perform some sort of static analysis on an Ada program.

AdaCore can assist customers in developing libadalang-based tools to meet their specific needs, as well as develop such tools upon request.

Typical libadalang applications include:

- Static analysis (property verification)
- Code instrumentation
- Design and document generation 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

3.3.4 GNATstack

GNATstack is a static analysis tool included with GNAT Pro that enables an Ada/C software developer to accurately predict the maximum size of the memory stack required for program execution.

GNATstack statically predicts the maximum stack space required by each task in an application. The computed bounds can be used to ensure that sufficient space is reserved, thus guaranteeing safe execution with respect to stack usage. The tool uses a conservative analysis to deal with complexities such as subprogram recursion, while avoiding unnecessarily pessimistic estimates.

This static stack analysis tool 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 analysis is based on information known at compile time. When the tool indicates that the result is accurate, the computed bound can never be exceeded.

On the other hand, there may be cases in which the results will not be accurate (the tool will report such situations) because of some missing information (such as the maximum depth of subprogram recursion, indirect calls, etc.). The user can assist the tool by specifying missing call graph and stack usage information.

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

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

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

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

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

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

3.3.4.1 Compliance with the ECSS Standards

The GNATstack tool can help meet several requirements in ECSS-E-ST-40C and ECSS-Q-ST-80C; details are provided in chapters *Compliance with ECSS-E-ST-40C* (page 49) and *Compliance with ECSS-Q-ST-80C* (page 63). 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
      - §5.8.3.5f Verification of code / source code robustness
  - Annex U - Software code verification
    - Verification check 12 (memory leaks)
- ECSS-Q-ST-80C
  - §5.6 Tools and supporting environment
    - §5.6.2 Development environment selection

3.3.5 GNAT Pro for Rust

The Rust language was designed for software that needs to meet stringent requirements for both assurance and performance: Rust is a memory-safe systems-programming language with software integrity guarantees (in both concurrent and sequential code) enforced by compile-time checks. The language is seeing growing use in domains such as automotive systems and is a viable choice for other high-assurance software.

AdaCore's GNAT Pro for Rust is a complete development environment for the Rust programming language, supporting both native builds and cross compilation to embedded targets. The product is not a fork of the Rust programming language or the Rust tools. Instead, GNAT Pro for Rust is a professionally supported build of a selected version of rustc and other core Rust development tools that offers stability for professional and high-integrity Rust projects. Critical fixes to GNAT Pro for Rust are upstreamed to the Rust community, and critical fixes made by the community to upstream Rust tools are backported as needed to the GNAT Pro for Rust code base. Additionally, the Assurance edition of GNAT Pro for Rust includes the *sustained branch* service (see *Sustained Branches* (page 34)) that strikes the balance between tool stability and project flexibility.

3.3.6 Integrated Development Environments (IDEs)

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

3.3.6.1 GNAT Studio

GNAT Studio is a powerful and simple-to-use IDE that streamlines software development from the initial coding stage through testing, debugging, system integration, and maintenance. It is designed to allow programmers to get the most out of GNAT Pro technology.

Functionality

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 a thorough understanding of a program at multiple levels. The IDE allows interfacing with third-party version control systems, easing both development and maintenance.

Robustness, Flexibility and Extensibility

Especially suited for large, complex systems, GNAT Studio can import existing projects from other Ada implementations while adhering to their file naming conventions and retaining the existing directory organization. Through the multi-language capabilities of GNAT Studio, components written in C and C++ can also be handled. The IDE 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.

Ease of Learning and Use

GNAT Studio is intuitive to new users thanks to its menu-driven interface with extensive online help (including documentation on all the menu selections) and *tool tips*. The Project Wizard makes it simple to get started, supplying default values for almost all of the project properties. For experienced users, it 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.

Support for 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 PC workstations.

3.3.6.2 VS Code Extensions for Ada and SPARK

AdaCore's extensions to Visual Studio Code (VS Code) enable Ada and SPARK development with a lightweight editor, as an alternative to the full GNAT Studio IDE. Functionality includes:

- Syntax highlighting for Ada and SPARK files
- Code navigation
- Error diagnostics (errors reported in the Problems pane)
- Build integration (execution of GNAT-based toolchains from within VS Code)
- Display of SPARK proof results (green/red annotations from GNATprove)
- Basic IntelliSense (completion and hover information for known symbols)

3.3.6.3 Eclipse Support – 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.

3.3.6.4 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, 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.

3.3.7 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 *Compliance with ECSS-E-ST-40C* (page 49) and *Compliance with ECSS-Q-ST-80C* (page 63). 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
    - §5.5.3 Coding and testing
    - §5.5.4 Integration
  - §5.7 Software delivery and acceptance process
    - §5.7.3 Software acceptance
  - §5.8 Software verification process
    - §5.8.3 Verification activities
  - §5.9 Software operation process
    - §5.9.2 Process implementation
  - §5.10 Software maintenance process
    - §5.10.2 Process implementation
    - §5.10.4 Modification implementation
  - §5.11 Software security process
    - §5.11.2 Process implementation
    - §5.11.3 Software security analysis
    - §5.11.5 Security analysis in the software lifecycle
    - Annex U - Source code verification
      - Verification check 12 (memory leaks)
- 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
    - §6.2.6 Verification
    - §6.2.9 Software security
  - §6.3 Requirements applicable to individual processes or activities
    - §6.3.4 Coding
    - §6.3.9 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, 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 (see[35]).

3.4 Static Verification: GNAT Static Analysis Suite (GNAT SAS)

GNAT SAS is a set of Ada static analysis tools that complement GNAT Pro for Ada. These tools can save time and effort in general during software development and verification, and they are useful in particular in supporting compliance with ECSS standards.

3.4.1 Defects and Vulnerability Analyzer

One of the main tools in GNAT SAS is an Ada source code analyzer that detects run-time and logic errors that can cause safety or security vulnerabilities in a codebase. This tool inspects the code for 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 during updates. It can also be used on new projects, helping to find errors early in the development life-cycle when they are least costly to repair. Using control-flow, data-flow, and other advanced static analysis techniques, this analysis tool detects errors that would otherwise only be found through labor-intensive debugging.

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

3.4.1.1 CWE Compatibility

The tool can detect a number of "Dangerous Software Errors" in the MITRE Corporation's Common Weakness Enumeration, and the tool has been certified (under its previous name, CodePeer) by The MITRE Corporation as a "CWE-Compatible" productPage 31, 31.

Here are the weaknesses that are detected:

[35] AdaCore. Press Release: European Space Agency Selects AdaCore's Qualified-Multitasking Solution for Spacecraft Software Development. September 24, 2019. URL: https://www.adacore.com/press/european-space-agency-selects-adacores-qualified-multitasking-solution-for-spacecraft-software-development.

Table 11: Defects and Vulnerability Analyzer and the CWE

CWE weakness	Description
CWE 120, 124, 125	Buffer overflow/underflow
CWE 136, 137	Variant record field violation, Use of incorrect type in inheritance hierarchy
CWE 190, 191	Numeric overflow/underflow
CWE 362, 366	Race condition
CWE 369	Division by zero
CWE 457	Use of uninitialized variable
CWE 476	Null pointer dereference
CWE 561	Dead (unreachable) code
CWE 563	Unused or redundant assignment
CWE 570	Expression is always false
CWE 571	Expression is always true
CWE 628	Incorrect arguments in call
CWE 667	Improper locking
CWE 682	Incorrect calculation
CWE 820	Missing synchronization
CWE 821	Incorrect synchronization
CWE 835	Infinite loop

3.4.1.2 Compliance with the ECSS Standards

The defects and vulnerability analyzer can help meet a number of requirements in ECSS-E-ST-40C and ECSS-Q-ST-80C; details are provided in chapters *Compliance with ECSS-E-ST-40C* (page 49) and *Compliance with ECSS-Q-ST-80C* (page 63). 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
  - §5.10 Software maintenance process
    - §5.10.4 Modification implementation
  - §5.11 Software security process
    - §5.11.2 Process implementation
    - §5.11.3 Software security analysis
    - §5.11.5 Software analysis in the software life cycle
  - Annex U - Software code verification
- 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
    * §6.2.6 Verification
    * §6.2.9 Software security
  - §7.1 Product quality objectives and metrication
    * §7.1.3 Assurance activities for product quality requirements

3.4.2 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).

3.4.2.1 Compliance with the ECSS Standards

The GNATmetric tool can help meet a number of requirements in ECSS-E-ST-40C and ECSS-Q-ST-80C; details are provided in chapters *Compliance with ECSS-E-ST-40C* (page 49) and *Compliance with ECSS-Q-ST-80C* (page 63). Here is a summary:

- ECSS-E-ST-40C
  - §5.10 Software maintenance process
    * §5.10.4 Modification implementation
- ECSS-Q-ST-80C
  - §5.2 Software product assurance programme management
    * §5.2.7 Quality requirements and quality models
  - §6.3 Requirements applicable to individual software engineering processes or activities
    * §6.3.4 Coding
  - §7.1 Product quality objectives and metrication
    * §7.1.3 Assurance activities for product quality requirements
    * §7.1.5 Basic metrics

3.4.3 GNATcheck

GNATcheck is a coding standard verification tool that is extensible and rule-based. It allows developers to completely define a project-specific coding standard as a set of rules, for example a subset of permitted language features and/or code formatting and style conventions. It verifies a program's conformance with the resulting rules and thereby facilitates demonstration of a system's compliance with a certification standard's requirements on language subsetting.

GNATcheck provides:

- An integrated "Ada Restrictions" mechanism for banning specific features from an application. This can be used to restrict features such as tasking, exceptions, dynamic allocation, fixed- or floating point, input/output, and unchecked conversions.
- Restrictions specific to GNAT Pro, such as banning features that result in the generation of implicit loops or conditionals in the object code, or in the generation of elaboration code.

- Additional Ada semantic rules resulting from customer input, such as ordering of parameters, normalized naming of entities, and subprograms with multiple returns.
- An easy-to-use interface for creating and using a complete coding standard.
- Generation of project-wide reports, including evidence of the level of compliance 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, certain cases of infinite recursion, incorrect data alignment, and accidental hiding of names.
- Style checks that allow developers to control indentation, casing, comment style, and nesting level.

AdaCore's GNATformat tool[36], which formats Ada source code according to the GNAT coding style[37], can help avoid having code that violates GNATcheck rules. GNATformat is included in the GNAT Pro for Ada toolchain.

GNATcheck comes with a query language (LKQL, for Language Kit Query Language) that lets developers define their own checks for any in-house rules that need to be followed. GNATcheck can thus be customized to meet an organization's specific requirements, processes and procedures.

3.4.3.1 Compliance with the ECSS Standards

The GNATcheck tool can help meet a number of requirements in ECSS-E-ST-40C and ECSS-Q-ST-80C; details are provided in chapters *Compliance with ECSS-E-ST-40C* (page 49) and *Compliance with ECSS-Q-ST-80C* (page 63). 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
      - §5.8.3.4 Verification of the software detailed design
  - §5.10 Software maintenance process
    - §5.10.4 Modification implementation
  - §5.11 Software security process
    - §5.11.2 Process implementation
    - §5.11.3 Software security analysis
    - §5.11.5 Software analysis in the software life cycle
  - Annex U - Software code verification
    - Verification check 3 (infinite loops)
    - Verification check 4 (misuse of arithmetic and logical operators)
- 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

[36] AdaCore. GNATformat Documentation. URL: https://docs.adacore.com/live/wave/gnatformat/html/user-guide/.

[37] *GNAT Coding Style: A Guide for GNAT Developers*. AdaCore, 2025. URL: https://gcc.gnu.org/onlinedocs/gnat-style.pdf.

    - §6.2.3 Handling of critical software
- §6.3 Requirements applicable to individual software engineering processes or activities
    - §6.3.4 Coding
- §7.1 Product quality objectives and metrication
    - §7.1.3 Assurance activities for product quality requirements

3.5 GNAT Dynamic Analysis Suite (GNAT DAS)

3.5.1 GNATtest

The GNATtest tool helps create and maintain a complete unit testing infrastructure for complex projects. It captures the simple idea that each public subprogram (these are known as *visible* subprograms in Ada) should have at least one corresponding unit test. GNATtest takes a project file as input, and 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 be used to help verify tagged type substitutability (the Liskov Substitution Principle) that can be used to demonstrate consistency of class hierarchies.

Testing a private subprogram is outside the scope of GNATtest but can be implemented by defining the relevant testing code in a private child of the package that declares the private subprogram. Additionally, hybrid verification can help (see *Hybrid Verification* (page 26)): augmenting testing with the use of SPARK to formally prove relevant properties of the private subprogram.

3.5.2 GNATemulator

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

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

There are two basic types of emulators. The first can serve as a surrogate for the final hardware during development for a wide range of verification activities, particularly those that require time accuracy. However, they tend to be extremely costly, and are often very slow. The second, which includes GNATemulator, does not attempt to be a complete time-accurate target board simulator, and thus it cannot be used for all aspects of testing. But it does provide a very efficient and cost-effective way to execute the target code very early in the development and verification processes. GNATemulator thus offers a practical compromise between a native environment that lacks target emulation capability, and a cross configuration where the final target hardware might not be available soon enough or in sufficient quantity.

3.5.3 GNATcoverage

GNATcoverage is a code coverage analysis tool. Its results are computed 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: 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.

3.5.4 GNATfuzz

GNATfuzz is a fuzzing tool; i.e., a tool that automatically and repeatedly executes tests and generates new test cases at a very high frequency to detect faulty behavior of the system under test. Such anomalous behavior is captured by monitoring the system for triggered exceptions, failing built-in assertions, and signals such as SIGSEGV.

Fuzz testing has proven to be an effective mechanism for finding corner-case vulnerabilities that traditional human-driven verification mechanisms, such as unit and integration testing, can miss. Since such vulnerabilities can often lead to malicious exploitations, fuzzing technology can help meet security verification requirements.

However, fuzz-testing campaigns are complex and time-consuming to construct, execute and monitor. GNATfuzz simplifies the process by analyzing a code base and identifying subprograms that can act as fuzz-test entry points. GNATfuzz then automates the creation of test harnesses suitable for fuzzing. In addition, GNATfuzz will automate the building, executing and analyzing of fuzz-testing campaigns.

GNATfuzz can serve a useful role as part of the software development and verification life cycle processes. For example, by detecting anomalous behavior such as data corruption due to task or interrupt conflicts, GNATfuzz can help prevent defects from being introduced into the source code.

3.5.5 TGen

TGen is an experimental run-time library / marshalling technology that can be used by *GNATtest* (page 44) and/or *GNATfuzz* (page 45) to automate the production of test cases for Ada code. It performs type-specific low-level processing to generate test vectors for subprogram parameters, such as uniform value distribution for scalar types and analogous strategies for unconstrained arrays and record discriminants. A command-line argument specifies the number of test values to be generated, and these can then be used as input to test cases created by GNATtest.

TGen can also be used with GNATfuzz, to help start a fuzz-testing campaign when the user supplies an initial set of test cases where some may contain invalid data. GNATfuzz will utilize coverage-driven fuzzer mutations coupled with TGen to convert invalid test cases into valid ones. TGen represents test data values compactly, removing a large amount of memory padding that would otherwise be present for alignment of data components. With its space-efficient representation, TGen significantly increases the probability of a successful mutation that results in a new valid test case.

3.5.6 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 *Compliance with ECSS-E-ST-40C* (page 49) and *Compliance with ECSS-Q-ST-80C* (page 63). 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
  - §5.10 Software maintenance process
    - §5.10.4 Modification implementation
  - §5.11 Software security process
    - §5.11.2 Process implementation
    - §5.11.3 Software security analysis
    - §5.11.5 Security analysis in the software lifecycle
- 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
    - §6.2.9 Software security
  - §6.3 Requirements applicable to individual software engineering processes or activities
    - §6.3.5 Testing and validation
  - §7.1 Product quality objectives and metrication
    - §7.1.3 Assurance activities for product quality requirements
    - §7.1.5 Basic metrics
  - §7.2 Product quality requirements
    - §7.2.3 Test and validation documents

3.6 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 en-

gineers have extensive experience supporting customers in critical areas including space, commercial and military avionics, 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.

CHAPTER

FOUR

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 (and thus the GNAT Pro Ada and SPARK Pro toolsuites 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}
  - Ada and SPARK are appropriate target languages for tools that support such techniques.

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, and thus GNAT Pro for Ada and SPARK Pro, 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, and thus SPARK Pro and GNAT Pro for Ada, 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, and thus SPARK Pro and GNAT Pro for Ada, help meet this requirement.

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; (*sic*) 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 types/objects, pragma Atomic). The protected type/object facility prevents certain kinds of race conditions: in state-based mutual exclusion, the state of an object cannot change between the time that a task evaluates the state condition and when it executes the code based on that state. Other race conditions can be detected by the Defects and Vulnerability Analyzer in the GNAT Static Analysis Suite.
- "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 by the GNAT Static Analysis Suite's Defect and vulnerability Analyzer 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}
  - Ada and SPARK are appropriate target languages for tools that support such techniques.

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 GNAT Dynamic Analysis Suite (see *GNAT Dynamic Analysis Suite (GNAT DAS)* (page 44)) 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 GNAT Dynamic Analysis Suite (see *GNAT Dynamic Analysis Suite (GNAT DAS)* (page 44)) 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 GNAT Dynamic Analysis Suite (see *GNAT Dynamic Analysis Suite (GNAT DAS)* (page 44)) 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 GNAT Dynamic Analysis Suite (see *GNAT Dynamic Analysis Suite (GNAT DAS)* (page 44)) can assist in this process, supplementing the GNAT Pro for Ada compilation facilities.

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

4.2.4.1 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 GNAT Pro Ada environment and GNAT Dynamic Analysis Suite (see *GNAT Dynamic Analysis Suite (GNAT DAS)* (page 44)) can support this process.
- "Validation shall be performed by test." {§5.6.3.1b}
  - AdaCore's GNAT Dynamic Analysis Suite (see *GNAT Dynamic Analysis Suite (GNAT DAS)* (page 44)) 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 Defects and Vulnerability Analyzer (see *Defects and Vulnerability Analyzer* (page 40)) in the GNAT Static Analysis Suite and/or SPARK Pro may be able to show that a run-time check will always succeed and that no test case will trigger a failure.

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

4.2.5.1 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 GNAT Dynamic Analysis Suite (see *GNAT Dynamic Analysis Suite (GNAT DAS)* (page 44)) can assist in this process.
- "Validation shall be performed by test." {§5.6.4.1b}
  - AdaCore's GNAT Dynamic Analysis Suite (see *GNAT Dynamic Analysis Suite (GNAT DAS)* (page 44)) 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 Defects and Vulnerability Analyzer (see *Defects and Vulnerability Analyzer* (page 40)) in the GNAT Static Analysis Suite and/or SPARK Pro may be able to show that a run-time check will always succeed and that no test case will trigger a failure.

4.3 Software delivery and acceptance process {§5.7}

4.3.1 Software acceptance {§5.7.3}

4.3.1.1 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.

4.4 Software verification process {§5.8}

4.4.1 Verification activities {§5.8.3}

4.4.1.1 Verification of the software detailed design {§5.8.3.4}

- "The supplier shall verify the software detailed design ensuring that: ... 5. testing is feasible, by assessing that: (a) controllability and observability features are identified and included in the detailed design in order to prepare the effective testing of the performance requirements; (b) computationally invariant properties and temporal properties are added within the design; (c) fault injection is possible. ... 7. the design is correct with respect to requirements and interfaces, including safety, security, and other critical requirements; 8. the design implements proper sequence of events, inputs, outputs, interfaces, logic flow, allocation of timing and sizing budgets, and error handling; ..." {§5.8.3.4a}
  - SPARK (and GNATprove) help meet criteria 5, 7, and 8.

4.4.1.2 Verification of code {§5.8.3.5}

- "The supplier shall verify the software code ensuring at least 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; 6. the code implements safety, security, and other critical requirements correctly as shown by appropriate methods; 7. the code is implemented in a way that it cannot result in runtime errors; 8. the effects of any residual runtime errors are controlled through error handling." {§5.8.3.5a}
  - SPARK (and GNATprove) help meet criterion 1.
  - Ada's strong typing and interface checks (and thus GNAT Pro for Ada) help meet criterion 2.
  - For criterion 3, the Defects and Vulnerability Analyzer (see *Defects and Vulnerability Analyzer* (page 40)) in the GNAT Static Analysis Suite and/or SPARK Pro can help verify correctness, and the GNATcheck utility included in the GNAT Static Analysis Suite (see *GNATcheck* (page 42)) can enforce conformance with a coding standard.
  - For criterion 5, Ada's strong typing and interface checks, as well as SPARK and GNATprove, can help show consistent interfaces and correct data flow.
  - The Defects and Vulnerability Analyzer (see *Defects and Vulnerability Analyzer* (page 40)) in the GNAT Static Analysis Suite, SPARK Pro, and the standard semantic checks performed by the GNAT Pro compiler can help meet criterion 6.
  - The GNAT Static Analysis Suite and SPARK / GNATprove can statically detect potential run-time errors and thereby help meet criterion 7.
  - Ada's exception handling facility 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	100.00%	100.00%	TBA	TBA
Source code decision coverage	100.00%	100.00%	TBA	TBA
Source code modified condition and decision coverage	100.00%	TBA	TBA	TBA

  Note: 'TBA' means that the value is to be agreed with the customer and measured as per ECSS-Q-ST-80C clause 6.3.5.2." {§5.8.3.5b}
  - The GNATcoverage tool (see *GNATcoverage* (page 45)) in the GNAT Dynamic Analysis Suite can help meet this requirement.
- "The supplier shall measure code coverage by analysis of the results of the execution of tests." {§5.8.3.5c}
  - The GNATcoverage tool (see *GNATcoverage* (page 45)) in the GNAT Dynamic Analysis Suite can help meet this requirement.
- "If it can be justified that the required percentage cannot be achieved by test execution, then analysis, inspection or review of design shall be applied to the non-covered code." {§5.8.3.5d}
  - The GNATcoverage tool (see *GNATcoverage* (page 45)) in the GNAT Dynamic Analysis Suite can help meet this requirement.

- "In case the traceability between source code and object code cannot be verified, the supplier shall perform additional code coverage analysis on object code level as follows:

Code coverage versus criticality category	A	B	C	D
Object code coverage	100.00%	N/A	N/A	N/A

Note: 'N/A' means not applicable.

Note: The use of some compiler optimization options can make the traceability between source code and object code not possible." {§5.8.3.5e}

  - The GNATcoverage tool (see *GNATcoverage* (page 45)) in the GNAT Dynamic Analysis Suite 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/ED-12C standard for airborne software for the commercial aviation industry.

- "The supplier shall verify source code robustness. 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 Defects and Vulnerability Analyzer (see *Defects and Vulnerability Analyzer* (page 40)) in the GNAT Static Analysis Suite will detect such errors as well as many others, including suspicious constructs that, although legitimate Ada, are likely logic errors.
  - SPARK Pro will enforce the SPARK subset and can be used to demonstrate absence of run-time errors.
  - The GNATstack tool in GNAT Pro 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.

4.4.1.3 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 to perform a schedulability analysis and prove that the design is feasible." {§5.8.3.11a}
  - The Ada Ravenscar profile restricts the tasking model to enable precise schedulability analysis, including Rate-Monotonic Analysis (RMA).

4.5 Software operation process {§5.9}

4.5.1 Process implementation {§5.9.2}

4.5.1.1 Problem handling procedures definition {§5.9.2.3}

- "The SOS [Software Operation Support] entity 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.

- "The SOS entity shall ensure that information regarding problems that can have an impact on security is protected." {§5.9.2.3b}
  - AdaCore's internal processes for maintaining sensitive data help to meet this criterion.

4.5.2 Software operation support {§5.9.4}

4.5.2.1 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 more specifically the GNAT Pro Assurance sustained branch service with its management of Known Problems, can help meet this requirement when an issue arises that is due to an AdaCore tool.

4.5.2.2 Vulnerabilities in operations {§5.9.4.3}

- "During operations, security vulnerabilities, threats and exploits shall be: 1. continuously monitored; 2. subject to further security analysis when evaluated relevant to the security of the system; 3. maintained for auditing purposes even when evaluated not relevant to the security of the system." {§5.9.4.3a}
  - The ability to express security-related requirements as contracts in the Ada source code, with run-time checks when needed, helps to meet criterion 1.

4.5.3 User support §5.9.5

4.5.3.1 Provisions of work-around solutions {§5.9.5.3}

- "If a reported problem has a temporary work-around solution before a permanent solution can be released, the SOS entity shall give to the originator of the problem report the option to use it." {§5.9.5.3a}
  - As part of the GNAT Pro Assurance sustained branch service, AdaCore can supply work-arounds to critical problems prior to releasing a permanent solution.

4.6 Software maintenance process {§5.10}

4.6.1 Process implementation {§5.10.2}

4.6.1.1 Long term maintenance for flight software {§5.10.2.2}

- "The maintainer shall propose solutions to be able to implement 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 receive support and not become obsolescent.

4.6.2 Modification implementation {§5.10.4}

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

- "The maintainer shall apply the software engineering processes specified in clauses 5.3 to 5.8 and 5.11 that are relevant to the scope of the modifications, using the tailoring applied during the development of the software." {§5.10.4.3a}
  - The Ada and SPARK languages have 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 and utilizing GNAT Pro for Ada and SPARK Pro, the developer can localize the impact of maintenance changes.
  - The GNAT Static and Dynamic Analysis Suites can ensure that any modifications meet the verification criteria applicable to the original software.

4.7 Software security process {§ 5.11}

4.7.1 Process implementation {§ 5.11.2}

- "A software security management plan shall be produced documenting: ... 7. the tools, methods and procedures to be used...." {§ 5.11.2a}
  - The Ada and SPARK languages, GNAT Pro for Ada, the GNAT Static and Dynamic Analysis Suites, and the SPARK Pro toolset support the software security management plan.

4.7.2 Software security analysis {§ 5.11.3}

- "The methods to be used for the security analysis shall be identified as part of the planning of the project." {§ 5.11.3b}
  - The Ada and SPARK languages, GNAT Pro for Ada, the GNAT Static and Dynamic Analysis Suites, and the SPARK Pro toolset help meet this requirement. For example:
    - Ada's compile-time checks prevent unsafe practices such as treating an integer value as a pointer.
    - The Defects and Vulnerability Analyzer in the GNAT Static Analysis Suite can detect a number of dangerous software errors in the MITRE Corporation's Common Weakness Enumeration.
    - The SPARK language and SPARK Pro can enforce security-related properties such as correct information flows and absence of run-time errors.
    - GNATfuzz can be used to stress test the software with malformed input values.

4.7.3 Security activities in the software life cycle {§ 5.11.5}

4.7.3.1 Security in the requirements baseline {§ 5.11.5.1}

- "The security assurance requirements shall determine the type and extent of security verification and validation activities, including testing, to be conducted.... Security verification and validation activities can include, for example, fuzzing tests...." {§ 5.11.5.1c}
  - The Ada and SPARK languages, GNAT Pro for Ada, the GNAT Static and Dynamic Analysis Suites, and the SPARK Pro toolset help meet this requirement.

4.7.3.2 Security in the detailed design and implementation engineering {§ 5.11.5.3}

- "The software security analysis shall be used during verification and validation activities to evaluate iteratively residual vulnerabilities and to reassess security risks." {§ 5.11.5.3b}
  - The Ada and SPARK languages, the GNAT Static and Dynamic Analysis Suites, and the SPARK Pro toolset help meet this requirement. For examples, see *Software security analysis {§ 5.11.3}* (page 58).

4.8 Software code verification {Annex U (informative)}

- "The following checks are expected to be performed on the software code:
  1. the code implements numerical protection mechanisms (e.g. against overflow and underflow, division by zero);
  2. the code does not perform out of bounds accesses - e.g. underrun or overrun of buffers, arrays or strings;
  3. the code does not include any infinite loop other than the main loop of the software and the main loops of cyclic tasks;
  4. the code appropriately uses arithmetical and logical operators (e.g. arithmetical OR vs. logical OR);
  5. implicit type conversions do not lead to arithmetical errors;
  6. the lifetime of variables is consistent with their use;
  7. the code makes proper use of static/global functions/variables to enforce the correct level of visibility;
  8. the code makes proper use of volatile variables for all variables that can be modified asynchronously (e.g. hardware access, memory-mapped I/O);
  9. the code does not perform invalid memory accesses (e.g. NULL dereferences);
  10. the code does not access uninitialized variables;
  11. the code does not perform unused assignments, unless this is done to trigger HW side-effects;
  12. there are no memory leaks;
  13. pointer arithmetic is justified and types of operands are consistent;
  14. the code does not lead to race conditions." {§ U.2}

Table 12 shows how AdaCore's technologies help meet these requirements:

Table 12: Verification Support

Verification check #	Technology	Explanation
1	Ada language, GNAT Pro for Ada, GNAT Pro Assurance	Run-time check[38]
	SPARK language, SPARK Pro	Static check
	Defects and Vulnerability Analyzer (in GNAT Static Analysis Suite)	Static check[39]
2	Ada language, GNAT Pro for Ada, GNAT Pro Assurance	Run-time check[38]
	SPARK language, SPARK Pro	Static check
	Defects and Vulnerability Analyzer (in GNAT Static Analysis Suite)	Static check[39]
3	Defects and Vulnerability Analyzer (in GNAT Static Analysis Suite)	Static check[39]
	GNATcheck (in GNAT Static Analysis Suite)	User-defined rule
	SPARK language, SPARK Pro	Static check
4	Ada language, GNAT Pro for Ada, GNAT Pro Assurance SPARK language, SPARK Pro	Static check
	GNATcheck (in GNAT Static Analysis Suite)	User-defined rule
5	Ada language, GNAT Pro for Ada, GNAT Pro Assurance SPARK language, SPARK Pro	Static check
	Defects and Vulnerability Analyzer (in GNAT Static Analysis Suite)	Static check[39]
6	Ada language, GNAT Pro for Ada, GNAT Pro Assurance SPARK language, SPARK Pro	Static check
7	Ada language, GNAT Pro for Ada, GNAT Pro Assurance SPARK language, SPARK Pro	Static check
8	Defects and Vulnerability Analyzer (in GNAT Static Analysis Suite)	Static check[39]
9	Ada language, GNAT Pro for Ada, GNAT Pro Assurance	Run-time check[38]
	SPARK language, SPARK Pro	Static check
	Defects and Vulnerability Analyzer (in GNAT Static Analysis Suite)	Static check[39]
10	SPARK language, SPARK Pro	Static check
	Defects and Vulnerability Analyzer (in GNAT Static Analysis Suite)	Static check[39]
11	Defects and Vulnerability Analyzer (in GNAT Static Analysis Suite)	Static check[39]
12	Defects and Vulnerability Analyzer (in GNAT Static Analysis Suite)	Static check[39]
	Ada Language, GNAT Pro for Ada, GNAT Pro Assurance	Run-time check[40]
	GNATstack (for preventing stack overflow)	Static check
13	Ada language, GNAT Pro for Ada, GNAT Pro Assurance SPARK language, SPARK Pro	Static check
14	Defects and Vulnerability Analyzer (in GNAT Static Analysis Suite)	Static check[39]

[38] Depending on the source code, the GNAT Pro compiler may be able to guarantee that the check will always succeed and can thus avoid generating run-time code to detect violations.

[39] The static checks can result in false alarms (*false positives*), but the user can calibrate the tool to control the

4.9 Compliance Summary

Table 13 shows how AdaCore's technologies support the requirements in ECSS-E-ST-40C:

Table 13: Technology Support for ECSS-E-ST-40C Compliance

Technology	§5.4	§5.5	§5.6	§5.7	§5.8	§5.9	§5.10	§5.11	Annex U
Ada language	✓	✓			✓	✓	✓	✓	✓
SPARK language	✓	✓			✓		✓	✓	✓
GNAT Pro for Ada	✓	✓	✓	✓	✓		✓	✓	✓
SPARK Pro	✓	✓	✓		✓		✓	✓	✓
GNAT Static Analysis Suite		✓	✓		✓		✓	✓	✓
GNAT Dynamic Analysis Suite		✓	✓		✓		✓	✓	
GNAT Pro Assurance	✓	✓	✓	✓	✓	✓	✓	✓	✓

tradeoff between soundness (no false negatives) and precision (minimization of false positives).

[40] Exhausting dynamic memory raises the `Storage_Error` exception at run time. Ada does not require a general garbage collector for storage reclamation, but several techniques can be used to prevent storage leaks:

- Ensure, through analysis or testing, that dynamic allocations occur only during startup (*elaboration time*) and not thereafter.
- Reclaim storage explicitly through Ada's `Unchecked_Deallocation` and ensure, through analysis or testing, that this does not create dangling references.
- Define a memory pool for a pointer type (*access type*) so that allocations of objects for that type only use that storage area, and through analysis or testing demonstrate that the pool is never exhausted. Ada's memory pool facility can be used to implement a reference counting strategy (for non-cyclic data structures) with automatic reclamation.

CHAPTER

FIVE

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 GNATmetric tool (see *GNATmetric* (page 42)) in the GNAT Static Analysis Suite can show quality metrics related to the source code structure.
  - The GNATdashboard IDE tool (see *GNATdashboard* (page 38)) can 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 for Ada environment and any supplemental tools that are selected (e.g., GNAT Static and Dynamic Analysis Suites, SPARK Pro) 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, ... the tools and methods are appropriate to the security sensitivity of the product as determined by the security analysis and as defined in the software security management plan" {§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/ED - 12C[41] and EN 50128[42].

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

[42] *EN 50128/A2: Railway applications – Communications, signalling and processing systems – Software for rail-*

- AdaCore's *sustained branch* service for its GNAT Pro for Ada Assurance product (see *Sustained Branches* (page 34)) can guarantee that the toolchain is maintained throughout the product lifetime.
- AdaCore's SPARK Pro environment (see *Static Verification: SPARK Pro* (page 30)) can be used to demonstrate security properties such as correct information flows and, for software at the highest security levels, compliance with formally specified requirements.

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; 13. the assessment of the product with respect to the security sensitivity level of the products; 14. compliance with appropriate security requirements due to organizational or national security regulations, policies or directives." {§5.6.2.1a}
  - AdaCore tools directly satisfy these criteria. The availability of qualification material for specific tools (GNATcheck, GNATprove, GNATstack) contributes to criterion 6, and the *sustained branch* service for GNAT Pro for Ada Assurance supports criteria 4, 7 and 10. AdaCore tools come with source code and flexible licensing, mitigating the issues noted in criteria 11 and 12. The GNAT Static Analysis Suite, SPARK, and the GNATfuzz tool in the GNAT Dynamic Analysis Suite facilitate demonstrating the relevant security properties (criteria 13 and 14).

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; use of dynamic code verification techniques." {§6.2.3.2a}
  - Ada's pragma `Restrictions` and pragma `Profile`, together with the GNAT Static Analysis Suite tool GNATcheck, can enforce a coding standard for Ada (in effect a 'safe subset'). See *GNAT Pro Enterprise* (page 33) and *GNATcheck* (page 42).
  - The SPARK language serves as a safe subset of full Ada, and a formal design language for formal proof.
  - 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.

way control and protection systems. CENELEC, July 2020.

  - The Defects and Vulnerability Analyzer in the GNAT Static Analysis Suite can detect unreachable code, including deactivated code.
  - The Ada language and the GNAT Pro Ada development environment allow contracts to be verified either statically or with run-time checks and thus facilitate dynamic code verification.

5.2.1.2 Verification {§6.2.6}

- "The completion of actions related to software problem reports generated during verification shall be verified and recorded." {§6.2.6.4a}
  - GNAT Pro for Ada Assurance, and its Sustained Branch service, help support compliance.
- "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." {§6.2.6.5a}
  - The Defects and Vulnerability Analyzer in the GNAT Static Analysis Suite can detect unreachable code, including deactivated code.

5.2.1.3 Software security {§6.2.9}

- "The supplier shall identify the methods and techniques for the software security analysis." {§ 6.2.9.3a}
  - The Ada and SPARK languages, GNAT Pro for Ada, the GNAT Static and Dynamic Analysis Suites, and the SPARK Pro toolset help meet this requirement.
- "Based on the results of the software security analysis, the supplier shall apply engineering measures to reduce the number of security sensitive software components and mitigate the risks associated with security sensitive software." {§ 6.2.9.4a}
  - Code modularization in Ada and SPARK can be used to minimize the number of security sensitive components, and to localize and encapsulate them in modules (packages) with compile-time enforcement of data consistency. Checks performed by the GNAT Pro and SPARK Pro tools help mitigate security risks, while hybrid verification can combine formal analysis (for high security components in SPARK) with traditional testing methods.

5.2.1.4 Handling of security sensitive software {§ 6.2.10}

- "The supplier shall define and implement measures to avoid propagation of failures, including the ones caused by deliberate action, between software components." {§ 6.2.10.1a}
  - The Ada language provides features that support this requirement for components in the same address space. Failures can be modeled by exceptions, and software components that need to interact synchronously or asynchronously can be modeled by tasks. (A failure in one component may be due to a device malfunction and is not necessarily security related, but it needs to be handled in a way that does not compromise security or other mission requirements.) If the failure (exception) occurs in an Ada task, then an appropriate style is for the task to take corrective measures by handling that exception and/or to report the failure so that a separate component can take further actions. If the task terminates as a result of the exception, the effect is localized; the failure that triggered the termination does not propagate to other components.
  - The SPARK language and SPARK Pro can help by ensuring that failures do not occur (proving the Absence of Run-Time Errors).

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 security, 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 *GNATcheck* (page 42)), can define and enforce a coding standard.
  - The SPARK language can serve as 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." {§6.3.4.3a}
  - GNATcheck and SPARK Pro are relevant tools for this activity.
- "The supplier shall define measurements, criteria and tools to ensure that the software code meets the quality and security requirements." {§6.3.4.6a}
  - The GNATmetric tool (see *GNATmetric* (page 42)) can be used to report quality data 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 *GNATdashboard* (page 38)) 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, GNATfuzz) can help meet this requirement for unit testing.
- "Based on the dependability and safety criticality, and the security sensitivity 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, GNATfuzz) 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, GNATfuzz) 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, GNATfuzz) 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, GNATfuzz) 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, GNATfuzz) 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, GNATfuzz) 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 {§6.3.9}

- "The maintenance plans and procedures shall include the following as a minimum: 1. scope of maintenance; 2. identification of the first version of the software product for which maintenance is to be done; 3. support organization; 4. maintenance life cycle; 5. maintenance activities; 6. quality measures to be applied during the maintenance; 7. security measures to be applied during the maintenance. 8. maintenance records and reports." {§6.3.9.4a}
  - 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 ...." {§6.3.9.7a}
  - 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 accommodation 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 integrate the metrics in a meaningful way.
  - Use of the SPARK language, and formal demonstration of absence of runtime errors, support this requirement.

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 in the GNAT Dynamic Analysis Suite 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.
  - The contract facility in Ada and SPARK supports verification by both static analysis and testing.
- "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}

- "Test procedures, data and expected results shall be specified." {§ 7.2.3.4a}
  - The GNATtest and GNATfuzz tools in the GNAT Dynamic Analysis Suite help meet this requirement.
- "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).

5.4 Compliance Summary

Table 14 shows how AdaCore's technologies support the requirements in ECSS-Q-ST-80C:

Table 14: Technology Support for ECSS-Q-ST-80C Compliance

Technology	§5.2	§5.6	§6.2	§6.3	§7.1	§7.2
Ada language			✓	✓		✓
SPARK language		✓	✓		✓	
GNAT Pro for Ada		✓	✓		✓	
SPARK Pro		✓	✓		✓	✓
GNAT Static Analysis Suite		✓	✓	✓	✓	
GNAT Dynamic Analysis Suite		✓	✓	✓	✓	✓
GNAT Pro Assurance		✓	✓	✓	✓	
GNAT Pro IDEs	✓			✓	✓	

CHAPTER

SIX

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
IRTAW	International Real-Time Ada Workshop
ISO	International Organization for Standardization
LSP	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

INDEX

W

Z