LEARN.
ADACORE.COM

AdaCore Technologies for Railway Software

Version 2.1

Jean-Louis Boulanger
and Quentin Ochem

Sep 26, 2026

CONTENTS

1 Introduction **5**
1.1 CENELEC safety-related railway standards . . . 5
1.2 Safety Integrity Levels . . . 6
1.3 AdaCore technologies for railway software . . . 7

2 CENELEC EN 50128 **9**
2.1 Overview . . . 9
2.2 Structure of the standard . . . 10
2.3 Tool qualification . . . 14
2.3.1 Tool classes . . . 14
2.3.2 AdaCore tool qualification support . . . 15

3 AdaCore Tools and Technologies Overview **17**
3.1 Ada . . . 17
3.1.1 Background . . . 17
3.1.2 Language Overview . . . 18
3.1.2.1 Scalar Ranges . . . 18
3.1.2.2 Contract-Based Programming . . . 18
3.1.2.3 Programming in the large . . . 19
3.1.2.4 Generic Templates . . . 19
3.1.2.5 Object-Oriented Programming (OOP) . . . 19
3.1.2.6 Concurrent Programming . . . 20
3.1.2.7 Systems Programming . . . 20
3.1.2.8 Real-Time Programming . . . 20
3.1.2.9 High-Integrity Systems . . . 20
3.1.2.10 Summary . . . 21
3.2 SPARK . . . 21
3.2.1 Flexibility . . . 22
3.2.2 Powerful Static Verification . . . 22
3.2.3 Ease of Adoption . . . 22
3.2.4 Hybrid Verification . . . 22
3.2.5 Reduced Cost and Improved Efficiency of Executable Object Code Verification . . . 22
3.3 GNAT Pro Assurance . . . 23
3.3.1 Sustained Branches . . . 23
3.3.2 Language and Tool Support . . . 23
3.3.3 Configurable Run-Time Libraries . . . 23
3.3.4 Full Implementation of Ada Standards . . . 24
3.3.5 Source to Object Traceability . . . 24
3.3.6 Safety-Critical Support and Expertise . . . 24
3.3.7 Libadalang . . . 25
3.3.8 GNATstack . . . 25
3.4 GNAT Static Analysis Suite (GNAT SAS) . . . 26
3.4.1 Defects and Vulnerability Analyzer . . . 26

3.4.2 GNATmetric . . . 26
3.4.3 GNATcheck . . . 26
3.5 GNAT Dynamic Analysis Suite (GNAT DAS) . . . 27
3.5.1 GNATtest . . . 27
3.5.2 GNATemulator . . . 28
3.5.3 GNATcoverage . . . 28
3.5.4 GNATfuzz . . . 28
3.5.5 TGen . . . 29
3.6 GNAT Pro for Rust . . . 29
3.7 Integrated Development Environments (IDEs) . . . 29
3.7.1 GNAT Studio . . . 29
3.7.2 VS Code Extensions for Ada and SPARK . . . 30
3.7.3 Eclipse Support – GNATbench . . . 30
3.7.4 GNATdashboard . . . 30

4 AdaCore Contributions to the Software Quality Assurance Plan 31
4.1 Table A.3 – Software Architecture (7.3) . . . 31
4.2 Table A.4 – Software Design and Implementation (7.4) . . . 33
4.3 Table A.5 – Verification and Testing (6.2 and 7.3) . . . 35
4.4 Table A.6 – Integration (7.6) . . . 36
4.5 Table A.7 – Overall Software Testing (6.2 and 7.7) . . . 37
4.6 Table A.8 – Software Analysis Techniques (6.3) . . . 37
4.7 Table A.9 – Software Quality Assurance (6.5) . . . 38
4.8 Table A.10 – Software Maintenance (9.2) . . . 39
4.9 Table A.11 – Data Preparation Techniques (8.4) . . . 39
4.10 Table A.12 – Coding Standards . . . 40
4.11 Table A.13 – Dynamic Analysis and Testing . . . 41
4.12 Table A.14 – Functional/Black Box Test . . . 42
4.13 Table A.15 – Textual Programming Language . . . 42
4.14 Table A.17 – Modeling . . . 43
4.15 Table A.18 – Performance Testing . . . 44
4.16 Table A.19 – Static Analysis . . . 44
4.17 Table A.20 – Components . . . 45
4.18 Table A.21 – Test Coverage for Code . . . 46
4.19 Table A.22 – Object Oriented Software Architecture . . . 46
4.20 Table A.23 – Object Oriented Detailed Design . . . 47

5 Technology Usage Guide 49
5.1 Analyzable Programs (D.2) . . . 49
5.2 Boundary Value Analysis (D.4) . . . 50
5.3 Control Flow Analysis (D.8) . . . 50
5.4 Data Flow Analysis (D.10) . . . 51
5.5 Defensive Programming (D.14) . . . 52
5.5.1 Data should be range checked . . . 52
5.5.2 Data should be dimension-checked . . . 53
5.5.3 Read-only and read-write parameters should be separated and their access checked . . . 53
5.6 Functions should treat all parameters as read-only . . . 53
5.6.1 Literals and constants should not be write-accessible . . . 54
5.6.2 Using GNAT SAS and SPARK to drive defensive programming . . . 54
5.7 Coding Standards and Style Guide (D.15) . . . 55
5.8 Equivalence Classes and Input Partition Testing (D.18) . . . 55
5.9 Error Guessing (D.20) . . . 56
5.10 Failure Assertion Programming (D.24) . . . 56
5.11 Formal Methods (D.28) . . . 56
5.12 Impact Analysis (D.32) . . . 57
5.13 Information Encapsulation (D.33) . . . 57
5.14 Interface Testing (D.34) . . . 60

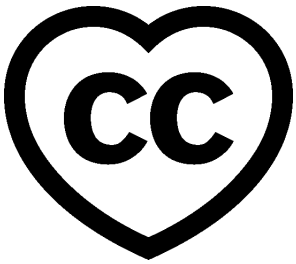
5.15 Language Subset (D.35) . . . 60
5.16 Metrics (D.37) . . . 60
5.17 Modular Approach (D.38) . . . 61
5.17.1 Connections between modules shall be limited and defined, coherence shall be strong . . . 61
5.17.2 Collections of subprograms shall be built providing several level of modules . . . 61
5.17.3 Subprograms shall have a single entry and single exit only . . . 62
5.17.4 Modules shall communicate with other modules via their interface . . . 62
5.17.5 Module interfaces shall be fully documented . . . 62
5.17.6 Interfaces shall contain the minimum number of parameters necessary 62
5.17.7 A suitable restriction of parameter number shall be specified, typically 5 62
5.17.8 Unit Proof and Unit Test . . . 62
5.18 Strongly Typed Programming Languages (D.49) . . . 62
5.19 Structure Based Testing (D.50) . . . 63
5.20 Structured Programming (D.53) . . . 63
5.21 Suitable Programming Languages (D.54) . . . 63
5.22 Object Oriented Programming (D.57) . . . 63
5.23 Procedural Programming (D.60) . . . 64

6 Technology Annex **65**
6.1 Ada Programming Language . . . 65
6.1.1 Qualification . . . 65
6.1.2 Annex D References . . . 65
6.2 GNAT Pro Assurance Toolsuite . . . 66
6.2.1 Qualification . . . 66
6.2.2 Run-Time Certification . . . 66
6.2.3 Annex D References . . . 66
6.3 SPARK Language and Toolsuite . . . 66
6.3.1 Qualification . . . 66
6.3.2 Annex D References . . . 66
6.4 GNAT Static Analysis Suite . . . 67
6.4.1 Defects and Vulnerability Analysis . . . 67
6.4.1.1 Qualification . . . 67
6.4.1.2 Annex D References . . . 67
6.4.2 Basic Static Analysis tools . . . 67
6.4.2.1 Qualification . . . 67
6.4.2.2 Annex D References . . . 68
6.5 GNAT Dynamic Analysis Suite . . . 68
6.5.1 Qualification . . . 68
6.5.2 Annex D References . . . 68

Index **69**

Copyright © 2015 – 2025, AdaCore

This book is published under a CC BY-SA license, which means that you can copy, redistribute, remix, transform, and build upon the content for any purpose, even commercially, as long as you give appropriate credit, provide a link to the license, and indicate if changes were made. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You can find license details on this page[1]



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

About the Authors

Jean-Louis Boulanger

Since the late 1990s Jean-Louis Boulanger has been an independent safety assessor for the CERTIFER certification authority in France, for safety-critical software in railway systems. He is an experienced safety expert in both the railway industries with the CENELEC standard and the automotive domain with ISO 26262. He has published a number of books on the industrial use of formal methods, as well as a book on safety for hardware architectures and a recent book on the application of the CENELEC EN 50128 and IEC 62279 standards. He has also served as a professor and researcher at the University of Technology of Compiègne.

Quentin Ochem

Quentin Ochem is the Chief Product and Revenue Officer at AdaCore, where he oversees marketing, sales, and product management while steering the company's strategic initiatives. He joined AdaCore in 2005 to work on the company's Integrated Development Environments and cross-language bindings. With an extensive background in software engineering in high-integrity domains such as avionics and defense, he has served leading roles in technical sales, customer training, and product development. Notably, he has conducted training on the Ada language, AdaCore tools, and the DO - 178B/ED - 12B and DO - 178C/ED - 12C software certification standards. In 2021 he stepped into his current role, directing the company's strategic initiatives.

Foreword

The guidance in the CENELEC standard EN 50128:2011 helps achieve confidence that railway control and protection software meets its safety requirements. Certifying compliance with this standard is a challenging task, especially for the testing and verification activities, but appropriate usage of qualified tools and specialized run-time libraries can significantly simplify the effort. This document explains how a number of technologies offered by AdaCore — tools, libraries, and supplemental services — can help. The content is based on the authors' many years of practical experience with the certification of railway software and with the Ada and SPARK programming languages.

Jean-Louis Boulanger
October 2015

Quentin Ochem, AdaCore
October 2015

Foreword to V2.1

In the years since the initial version of this document was published, the EN 50128:2011 standard has been amended twice, and AdaCore's products have evolved to meet the growing demands for, and challenges to, high assurance in mission-critical real-time software. This revised edition reflects the current (2025) versions of EN 50128 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 Vasiliy Fofanov (AdaCore) for his detailed and helpful review and suggestions.

For up-to-date information on AdaCore support for developers of rail software, please visit[Page 1, 1].

Ben Brosgol, AdaCore
September 2025

CHAPTER

ONE

INTRODUCTION

1.1 CENELEC safety-related railway standards

Railway projects are subject to a legal framework (laws, decrees, etc.) and also a normative process based on certification standards. In Europe, these standards are issued and maintained by CENELEC (European Committee for Electrotechnical Standardization). This document explains the usage of AdaCore's technologies in conjunction with EN 50128:2011[2] — *Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems* — as modified by amendments EN 50128/A1[3] and EN 50128/A2[4]. (For ease of exposition, the 2011 edition of the standard, as modified by the A1 and A2 amendments, will simply be referred to as EN 50128.)

EN 50128 is concerned with the safety-related aspects of a railway system, down to the hardware and/or software elements used. This document will cover where AdaCore's technologies fit best and how they can best be applied to meet various requirements in this standard.

EN 50128 is based on fundamentals described in other CENELEC railway standards:

- EN 50126-1[5] — *Railway applications - The specification and demonstration of reliability, availability, maintainability and safety (RAMS) - Part 1: Generic RAMS process* (subsequently modified by EN 50126-1/A1[6])
- EN 50126-2[7] — *Railway applications - The specification and demonstration of reliability, availability, maintainability and safety (RAMS): Part 2: systems approach to safety*
- EN 50129[8] — *Railway applications - Communication, signalling and processing systems - Safety related electronic systems for signalling*

As noted in EN 50128, page 7:

> *EN 50126-1 addresses system issues on the widest scale, while EN 50129 addresses the approval process for individual systems which can exist within the overall railway control and protection system. ... [EN 50128] concentrates on the methods which need to be used in order to provide software which meets the demands of safety integrity which are placed upon it by these wider considerations.*

[2] *EN 50128: Railway applications – Communications, signalling and processing systems – Software for railway control and protection systems*. CENELEC, 2011.

[3] *EN 50128/A1: Railway applications – Communications, signalling and processing systems – Software for railway control and protection systems*. CENELEC, February 2020.

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

[5] *EN 50126-1: Railway applications - The specification and demonstration of reliability, availability, maintainability and safety (RAMS)- Part 1: Generic RAMS process*. CENELEC, October 2017.

[6] *EN 50126-1/A1: Railway applications - The specification and demonstration of reliability, availability, maintainability and safety (RAMS) - Part 1: Generic RAMS process*. CENELEC, 2024.

[7] *EN 50126-2: Railway applications - The specification and demonstration of reliability, availability, maintainability and safety (RAMS)- Part 2: systems approach to safety*. CENELEC, October 2017.

[8] *EN 50129: Railway applications - Communication, signalling and processing systems - Safety related electronic systems for signalling*. CENELEC, November 2018.

In addition to EN 50126 and EN 50129, several other CENELEC standards relate to software's role in the safety of a railway system:

- EN 50657:2017[9] as modified by amendment EN 50657/A1[10] — *Railways applications - Rolling stock applications - Software on Board Rolling Stock*

  This standard extends the principles of EN 50128 into the rolling stock domain, focusing on onboard systems such as braking, door control, and driver interfaces.

  It retains RAMS goals but tailors them for embedded systems in motion, where environmental and operational variables are more dynamic.

- EN 50716:2023[11] — *Railway Applications - Requirements for software development*

  This standard is a successor to EN 50128 and EN 50657, providing better alignment with EN 50126-1 and EN 50126-2. As of 2025 it is at the early adoption stage but is intended to supersede both EN 50128 and EN 50657.

Fig. 1 depicts the relationships among the various standards.

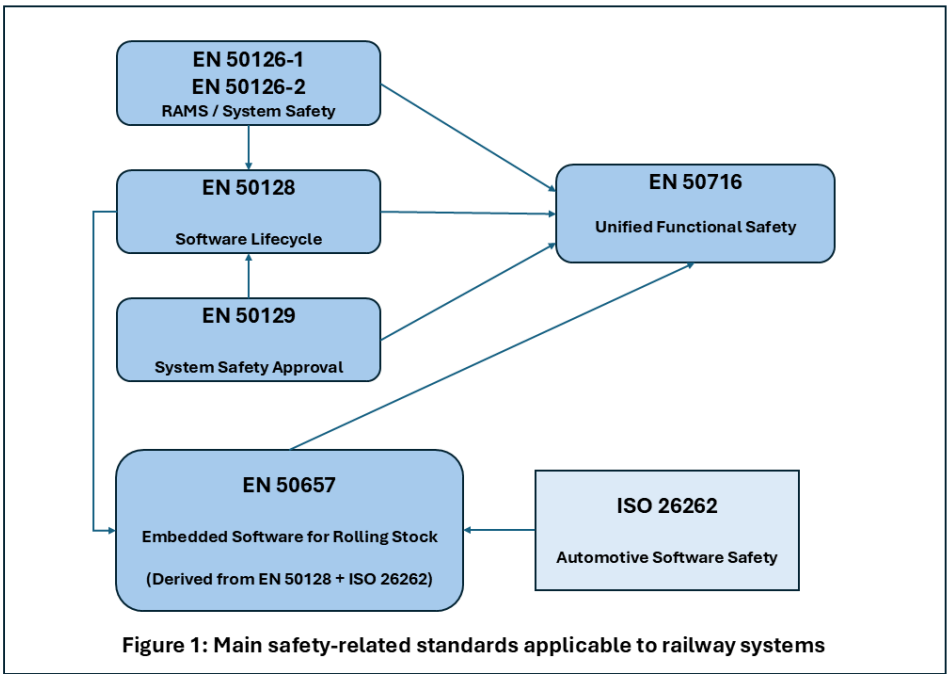


Fig. 1: Relationships among the various standards

1.2 Safety Integrity Levels

A key concept in EN 50128 is the *Safety Integrity Level* (SIL) of a software component, which reflects the risk of a hazard occurring if the software fails. If there is no impact on safety, the level is referred to as "Basic Integrity" (earlier known as SIL 0). Otherwise the level has a value between 1 and 4, where 4 is the most critical; i.e., with the highest risk of a hazard in case of software failure.

EN 50128 defines the techniques/measures that need to be used at various software life cycle stages, based on the applicable SIL.

[9] *EN 50657: Railways applications – Rolling stock applications - Software on Board Rolling Stock*. CENELEC, August 2017.
[10] *EN 50657/A1: Railways applications – Rolling stock applications - Software on Board Rolling Stock*. CENELEC, November 2023.
[11] *EN 50716: Railway applications – Requirements for software development*. CENELEC, November 2023.

1.3 AdaCore technologies for railway software

AdaCore's technologies revolve around programming activities, as well as the closely related design and verification activities. This is the bottom of the "V" cycle as defined in EN 50128, sub-clause 5.3, Figure 4 (see Fig. 2 below). The company's tools exploit the features of the Ada language (highly recommended by table A.15) and its formally verifiable SPARK subset. In particular, the 2012 version of the Ada standard includes some significant capabilities in terms of specification and verification.

AdaCore's technologies bring two main benefits to the software life cycle processes defined by the CENELEC railway standards.

- *Expressing software interface specifications and software component specifications directly in the source code.*

  Interfaces can be precisely expressed through standard syntax for features such as strong typing, parameter constraints, and subprogram contracts. These help to clarify interface documentation, to enforce program constraints and invariants, and to provide an extensive foundation for software component and integration verification.

- *Reducing the verification costs.*

  Bringing additional specification at the language level allows verification activities to run earlier in the software life cycle, during the software component implementation itself. Tools provided by AdaCore support this effort and are designed to be equally usable by both the development team and the verification team. Allowing developers to use verification tools greatly reduces the number of defects found at the verification stage and thus reduces costs related to change requests identified in the ascending stages of the cycle.

AdaCore's technologies can be used at all Safety Integrity Levels, from Basic Integrity to SIL 4. At lower levels, the full Ada language is suitable, independent of platform. At higher levels, specific subsets will be needed, for example the Ravenscar Profile ([12],[13]) for concurrency support with analyzable semantics and a reduced footprint, or the Light Profile[14] for a subset with no run-time library requirements. At the highest level (SIL 4) the SPARK language ([13],[15]) and its verification toolsuite enable mathematical proof of properties including correct information flow, absence of run-time exceptions, and, for the most critical code, correctness of the implementation against a formally defined specification.

The following technologies will be presented:

- *Ada*, a compilable programming language supporting imperative, object-oriented, and functional programming styles and offering strong specification and verification features. Unless otherwise indicated, "Ada" denotes the 2012 version of the Ada language standard.
- *SPARK*, an Ada language subset and toolset supporting formal verification of program properties such as Absence of Run-Time Errors
- *GNAT Pro Assurance*, a specialized edition of AdaCore's GNAT Pro language development environments that is oriented towards projects with long maintenance cycles or certification requirements
- The *GNAT Static Analysis Suite* ("GNAT SAS"), comprising several tools:

[12] Alan Burns, Brian Dobbing and Tullio Vardanega. Guide for the use of the Ada Ravenscar Profile in high integrity systems. *Ada Letters*, June 2004.

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

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

[15] *SPARK Reference Manual, Release 2020*. AdaCore and Altran, 2020. URL: https://www.adacore.com/uploads/techPapers/spark_rm_community_2020.pdf.

  - A "bug finder" engine that identifies potential defects and vulnerabilities in Ada code
  - GNATmetric — a metric computation tool
  - GNATcheck — a coding standard checker
- The *GNAT Dynamic Analysis Suite* ("GNAT DAS"), comprising several tools:
  - GNATtest — a unit testing framework generator
  - GNATemulator — a processor emulator
  - GNATcoverage — a structural code coverage analyzer
  - GNATfuzz — a fuzzing tool that helps uncover potential faults
  - TGen — an experimental run-time library for automating test case generation
- *GNAT Pro for Rust*, a professionally supported complete development environment for the Rust programming language
- Several *Integrated Development Environments* (IDEs):
  - GNAT Studio — a robust, flexible, and extensible IDE
  - VS Code support — extensions for Ada and SPARK
  - GNATbench — an Ada-knowledgeable Eclipse plug-in
  - GNATdashboard — a metric integration and management platform

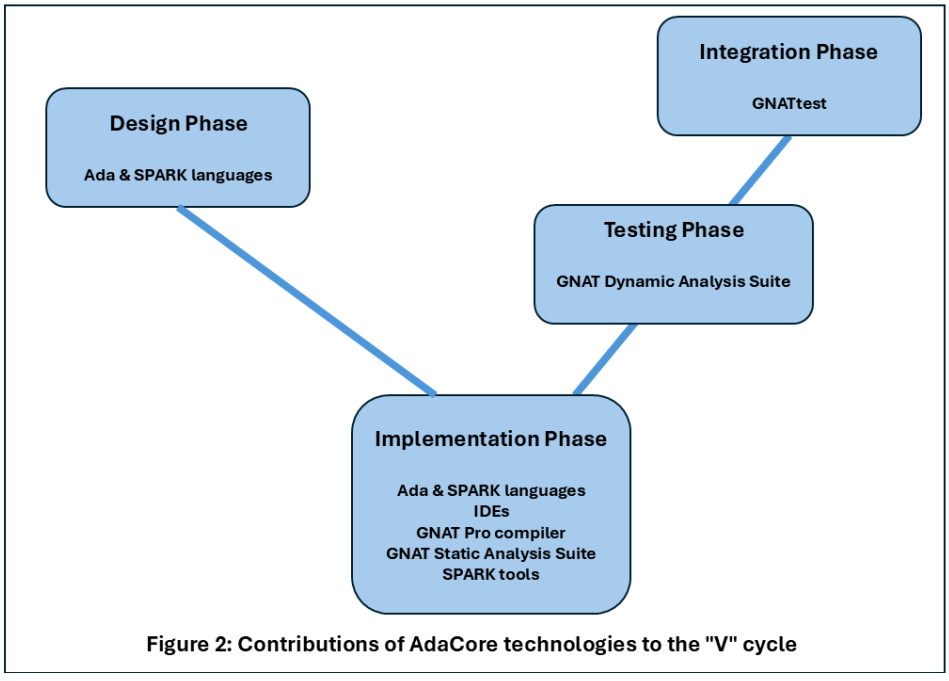


Fig. 2: Contributions of AdaCore technology to the "V" Cycle

CHAPTER

TWO

CENELEC EN 50128

2.1 Overview

EN 50128 governs software used in railway control and protection applications, i.e., systems that ensure the safe and efficient movement of trains. Examples include:

- *Automatic Train Protection (ATP)*, which ensures automatic braking to avoid collisions or overspeed;
- *Interlocking Systems*, which prevent conflicting train movements through tracks, signals, and switches;
- *Train Control Management Systems (TCMS)*, which coordinate control of subsystems (doors, brakes, traction);
- *Level Crossing Protection*, which manages gates and warnings at road-rail intersections; and
- *Centralized Traffic Control (CTC)*, which oversees train routing and dispatch across large regions.

The goal of the standard is to provide confidence that the software functions reliably and safely relative to its SIL. To this end it specifies requirements in areas including the following:

- Software development lifecycle processes;
- Verification and validation;
- Tools, techniques, and documentation;
- Risk mitigation measures; and
- Assessment of compliance with the standard.

More specifically, EN 50128 identifies the procedures and prerequisites (organization, independence and competencies management, quality management, V&V team, etc.) applicable to the development of programmable electronic systems used in railway control and protection applications. The standard therefore may apply to some software applications in the rail sector but not necessarily to all.

EN 50128 is used in both safety-related and non-safety-related applications and applies exclusively to software and its interaction with the whole system. (In light of its role in the certification of non-safety-related software, the standard introduces the Safety Integrity Level "Basic Integrity", which pertains to software that is not safety related.)

Although EN 50128 is targeted to the rail industry, it is not intrinsically domain specific. The standard is basically a specification of sound software engineering practices for long-lived large-scale high-assurance systems in general and could in principle be applied in other domains.

One of the distinctive points of EN 50128 is its requirement to justify the implementation of the resources. For this reason, it is said to be a "resources standard".

2.2 Structure of the standard

Fig. 3 illustrates the structure of EN 50128 (note that chapters in CENELEC standards are referred to as clauses, and individual sections and sub-sections within a chapter are sub-clauses).

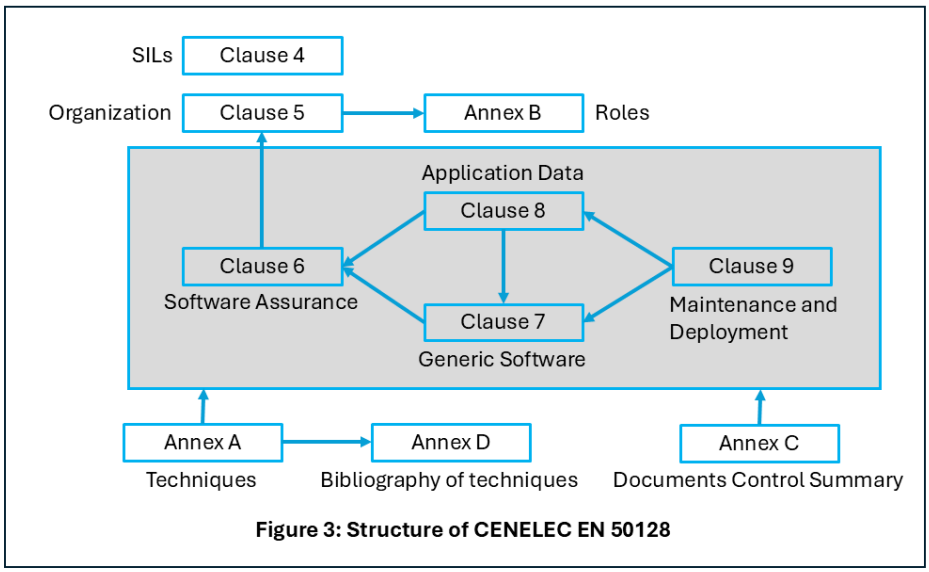


Fig. 3: Structure of CENELEC EN 50128

Clauses 1, 2, and 3 — *Scope*, *Normative references*, and *Terms, definitions and abbreviations*, respectively — provide context and basic information.

Clause 4, *Objectives, conformance and software safety integrity levels*, identifies the five Safety Integrity Levels and states the criterion for conformance to the standard:

> *To conform to this European standard it shall be shown that each of the requirements has been satisfied to the software safety integrity level defined and therefore the objective of the sub-clause in question has been met.*

This clause also specifies the role of normative Annex A in the selection of techniques and measures for satisfying the requirements, and the means for verifying compliance (inspection of the required documents, augmented when appropriate by other evidence such as auditing and the witnessing of tests).

Clauses 5 through 9 form the core of the standard, with sub-clauses providing the following content:

- *Objective*: the purpose of meeting the requirements specified in the sub-clause
- *Input documents* (if applicable)
- *Output documents* (if applicable)
- *Requirements*: Details on what the software supplier needs to do or provide. In some cases the requirements reference the tables in Annex A for specific techniques or measures to be used.

Clause 5, *Software management and organization*, covers three topics:

- Organization, roles and responsibilities (sub-clause 5.1);

- Personnel competence (sub-clause 5.2); and
- Lifecycle-related issues (sub-clause 5.3).

The standard does not dictate a specific lifecycle, but it cites the "V" approach as a recommendation, from the software specification to the overall software testing and integration, and also imposes some requirements. For example, the chosen lifecycle model needs to account for potential iterations between phases, and detailed documentation in the Software Quality Assurance Plan as specified in sub-clause 6.5 has to be supplied.

Clause 6, *Software assurance*, has the goal of achieving a software package with a minimum level of error and involves a variety of activities and technologies:

- Software testing (sub-clause 6.1);
- Software verification (sub-clause 6.2) — defined in sub-clause 3.1.48 as "confirmation, through the provision of objective evidence, that specified requirements have been fulfilled";
- Software validation (sub-clause 6.3) — defined in sub-clause 3.1.46 as "confirmation, through the provision of objective evidence, that the requirements for a specific intended use or application have been fulfilled";
- Software assessment (sub-clause 6.4);
- Software quality assurance (sub-clause 6.5);
- Modification and change control (sub-clause 6.6); and
- Support tools and languages (sub-clause 6.7) — see *Tool qualification* (page 14) below.

As shown in[16], for software applications the assessment process involves demonstrating that the software application achieves its associated safety objectives.

EN 50128 makes a clear separation between the application software, referred to as the *generic software* (Clause 7), and the data or algorithms that are used to configure the generic software (Clause 8).

Clause 7, *Generic software development*, has the following sub-clauses:

- Lifecycle and documentation for generic software (sub-clause 7.1);
- Software requirements (sub-clause 7.2);
- Architecture and Design (sub-clause 7.3);
- Component design (sub-clause 7.4);
- Component implementation and testing (sub-clause 7.5);
- Integration (sub-clause 7.6); and
- Overall Software Testing / Final Validation (sub-clause 7.7).

Clause 8, *Development of application data or algorithms: systems configured by application data or algorithms*, ensures that the configuration parameters are verified and validated with the same degree of assurance, based on the relevant SIL, as is needed for the generic software that they configure.

An important part of the standard is Clause 9, *Software deployment and maintenance*. As stated in sub-clauses 9.1.1 and 9.2.1, the objectives of this clause are, respectively:

> *To ensure that the software performs as required, preserving the required software integrity level when it is deployed in the final environment of the application.*

and

[16] Jean-Louis Boulanger and Walter Schön. Assessment of Safety Railway Application. In *ESREL 2007 - the 18th European Safety and Reliability Conference*. 2007.

> *To ensure that the software performs as required, preserving the required software integrity level and dependability when making corrections, enhancements or adaptations to the software itself.*

Annex A (normative), *Criteria for the Selection of Techniques and Measures*, contains a set of tables that correlate the artifacts and practices (documentation, techniques, and measures) specified elsewhere in the standard, with an indication of whether, and how strongly, they are recommended based on the software's SIL:

- **M**: Mandatory. Must be used
- **HR**: Highly Recommended. If not used, need to explain rationale for using alternative technique
- **R**: Recommended
- **--**: No recommendation either for or against usage
- **NR**: Not recommended. If used, need to explain rationale for decision

Annex A consists of two sub-clauses:

- *Clauses tables (A.1)*; the table headings identify the sub-clause(s) containing the relevant requirements:
  - Table A.1 – Lifecycle Issues and Documentation (5.3)
  - Table A.2 – Software Requirements Specification (7.2)
  - Table A.3 – Software Architecture (7.3)
  - Table A.4 – Software Design and Implementation (7.4)
  - Table A.5 – Verification and Testing (6.2 and 7.3)
  - Table A.6 – Integration (7.6)
  - Table A.7 – Overall Software Testing (6.2 and 7.7)
  - Table A.8 – Software Analysis Techniques (6.3)
  - Table A.9 – Software Quality Assurance (6.5)
  - Table A.10 – Software Maintenance (9.2)
  - Table A.11 – Data Preparation Techniques (8.4)
- *Detailed tables (A.2)*; these are lower-level tables that expand on certain entries in the Clauses tables:
  - Table A.12 – Coding Standards
  - Table A.13 – Dynamic Analysis and Testing
  - Table A.14 – Functional/Black Box Test
  - Table A.15 – Textual Programming Languages
  - Table A.16 – Diagrammatic Languages for Application Algorithms
  - Table A.17 – Modeling
  - Table A.18 – Performance Testing
  - Table A.19 – Static Analysis
  - Table A.20 – Components
  - Table A.21 – Test Coverage for Code
  - Table A.22 – Object Oriented Software Architecture
  - Table A.23 – Object Oriented Detailed Design

As an example, Table A.4 contains a row for the programming language(s) selection:

Technique/Measure	Ref	Basic Integrity	SIL 1	SIL 2	SIL 3	SIL 4
...	...	...	...	...	...	...
10 Programming Language	Table A.15	R	HR	HR	HR	HR
...	...	...	...	...	...	...

Table A.15 contains a row for Ada:

Technique/Measure	Ref	Basic Integrity	SIL 1	SIL 2	SIL 3	SIL 4
ADA	D.54	R	HR	HR	HR	HR
...	...	...	...	...	...	...

Sub-clause D.54 (*Suitable Programming languages*) notes the features that a suitable language should have (e.g., run-time array bound checking), and features that it should encourage (e.g., definition of variable sub-ranges). On the other side, D.54 also lists features that should be avoided because they complicate verification (e.g., implicit variable initialization).

The entries in Tables A.4 and A.15 show that Ada is a Highly Recommended language at SIL 1 through SIL 4 and a Recommended language at the Basic Integrity level. Features that should be avoided can be detected and prevented by using AdaCore's GNATcheck tool in the GNAT Static Analysis Suite; see *GNAT Static Analysis Suite (GNAT SAS)* (page 26).

Annex B (normative), *Key software roles and responsibilities*, consists of ten tables detailing the responsibilities and key competencies for the various roles specified in the standard: Requirements Manager, Designer, Implementor, Tester, Verifier, Integrator, Validator, Assessor, Project Manager, and Configuration Manager.

Annex C (informative), *Documents Control Summary*, provides a table that lists, for each project phase, its output documents and, for each document, the responsible author and reviewer(s). The lifecycle phases and their associated document count are:

- *Planning:* 5 documents
- *Software requirements:* 3 documents
- *Architecture and design:* 6 documents
- *Component design:* 3 documents
- *Component implementation and testing:* 3 documents
- *Integration:* 3 documents
- *Overall software testing / Final validation:* 4 documents
- *Systems configured by application data/algorithms:* 8 documents
- *Software deployment:* 5 documents
- *Software maintenance:* 4 documents
- *Software assessment:* 2 documents

Annex D (informative), *Bibliography of techniques*, details the aim and description for seventy-one specific software engineering practices. These are applicable at various lifecycle phases; for example:

- Coding Standards and Style Guide (sub-clause D.15) and Language Subset (sub-clause D.35) at the design and implementation phase,

- Formal Methods and Formal Proof (sub-clauses D.28 and D.29) at the implementation and verification phases, and
- Equivalence Classes and Input Partition Testing (sub-clause D.18) at the testing phase.

Annex ZZ (Informative), *Relationship between this European standard and the essential requirements of EU Directive 2016/797/EU [2016 OJ L138] aimed to be covered* was introduced in EN 50128/A1. It contains a table showing the relationship noted in the Annex title.

2.3 Tool qualification

An earlier edition of the standard, EN 50128:2001, introduced a requirement that the compilers employed for a project be purpose-certified, but did not give a clear indication of what precisely was expected. Clause 6.7 in the 2011 revision formalizes this concept, which will be referred to here as "tool qualification", and provides details on what needs to be performed and/or supplied. (The standard does not use a specific term for this process, but the "tool qualification" terminology from the airborne software standards DO - 178C/ED - 12C[17] and DO - 330/ED - 215[18] is appropriate.)

2.3.1 Tool classes

Tool qualification is based on the recognition that different tools need different levels of confidence in their reliability, based on how a tool error affects the application software. This is formalized in the concept of a "tool class". As stated in sub-clause 6.7.1:

> *The objective is to provide evidence that potential failures of tools do not adversely affect the integrated toolset output in a safety related manner that is undetected by technical and/or organisational measures outside the tool. To this end, software tools are categorised into three classes namely, T1, T2 & T3 respectively.*

- **T1** is reserved for tools that affect neither the verification of the software nor the final executable file.
- **T2** applies to tools where a fault could lead to an error in the results of the verification or validation. Examples include tools used for verifying compliance with a coding standard, generating quantified metrics, performing static analysis of the source code, managing and executing tests, etc.
- **T3** applies to tools where a fault could have an impact on (and, for example, introduce errors into) the final executable software. This class includes compilers, code generators, etc.

Sub-clause 6.7 of EN 50128 defines a set of recommendations for each tool class; these affect the content of the tool qualification report. The standard identifies twelve requirements (numbered from 6.7.4.1 to 6.7.4.12) concerning tool qualification. Requirement 6.7.4.12 is a mapping from each tool class to the applicable sub-clauses in the standard. It is shown here in the table below, which augments the version in the standard by also specifying the lifecycle phase that is relevant for each sub-clause. The steps shown indicate the requirements to be met and reflect the additional effort needed as the tool level increases; for further information, please see[19], Chapter 9.

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

[18] *DO-330/ED-215: Software Tool Qualification Considerations*. RTCA and EUROCAE, December 2011.

[19] Jean-Louis Boulanger. *CENELEC 50128 and IEC 62279 standards*. ISTE-Wiley, London, 2015. URL: https://onlinelibrary.wiley.com/doi/book/10.1002/9781119005056.

Tool class	Applicable sub-clause(s)	Lifecycle phase
T1	6.7.4.1	Tool identification
T2	6.7.4.1	Tool identification
	6.7.4.2	Tool justification
	6.7.4.3	Tool specification/manual
	6.7.4.10, 6.7.4.11	Tool version management
T3	6.7.4.1	Tool identification
	6.7.4.2	Tool justification
	6.7.4.3	Tool specification/manual
	(6.7.4.4 and 6.7.4.5) or 6.7.4.6	Tool conformity evidence
	(6.7.4.7 or 6.7.4.8) and 6.7.4.9	Tool requirement fulfillment
	6.7.4.10, 6.7.4.11	Tool version management

2.3.2 AdaCore tool qualification support

As will be explained below, AdaCore supports EN 50128 compliance through tools qualified for several purposes:

- Static and dynamic analysis;
- Code verification including formal proof; and
- Compilation with traceability and reproducibility guarantees.

These capabilities reduce certification risk while improving code quality and lifecycle confidence.

AdaCore's qualification packages contain the information required by EN 50128, such as documentation, history, infrastructure, user references, recommended usage, validation strategy, configuration management and change tracking.

Furthermore, tools can be provided through a subscription service known as "sustained branches" (see *Sustained Branches* (page 23)). In this mode, a specific version of the tools can be put into special maintenance, where AdaCore can investigate known problems and provide repairs or work-arounds for potential issues on these branches without unrelated updates that may risk regressions.

AdaCore's decades-long experience in software certification for embedded and safety-critical domains, including rail and avionics, ensures that customers have access to:

- Qualification material for EN 50128 and/or DO - 330/ED - 215 tool assessment;
- A formally verifiable language (SPARK) for high-integrity use cases; and
- Lifecycle support aligned with the needs of long-term platform deployments

CHAPTER

THREE

ADACORE TOOLS AND TECHNOLOGIES OVERVIEW

3.1 Ada

3.1.1 Background

Ada is a modern programming language designed for large, long-lived applications — and embedded systems in particular — where reliability, maintainability, and efficiency are essential. It 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. The language 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 (ISO) object-oriented language. Under the auspices of ISO, a further (minor) revision was completed as an amendment to the standard; this version of the language is known as Ada 2005. Additional features (including support for contract-based programming in the form of subprogram pre- and postconditions and type invariants) were added in the Ada 2012 version of the language standard, and a number of features to increase the language's expressiveness were introduced in Ada 2022 (see[20],[21],[22],[23] for information about Ada).

The name "Ada" is not an acronym; it was chosen in honor of Augusta Ada Lovelace (1815-1852), a mathematician who is 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 is seeing significant usage worldwide in high-integrity / safety-critical / high-security domains including railway systems, commercial and military aircraft avionics, air traffic control, and medical devices.

With its embodiment of modern software engineering principles, Ada is an excellent teaching language for both introductory and advanced computer science courses, and it has been the subject of significant university research especially in the area of real-time technologies. The so-called Ravenscar Profile — a subset of the language's concurrency features with deterministic semantics — 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. AdaCore's initial GNAT compiler was essential to the growth of Ada 95; it was delivered at the time of the language's standard-

[20] 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.

[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.

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

[23] 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.

ization, thus guaranteeing that users would have a quality implementation for transitioning to Ada 95 from Ada 83 or other languages.

3.1.2 Language Overview

Ada is multi-faceted. From one perspective it is a classical stack-based general-purpose language, not tied to any specific development methodology. It has a simple syntax, structured control statements, flexible data composition facilities, strong type checking, traditional features for code modularization (*subprograms*), and a mechanism for detecting and responding to exceptional run-time conditions (*exception handling*). But it also includes much more:

3.1.2.1 Scalar Ranges

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

Here's an example of an integer scalar range:

```
declare
   Score : Integer range 1..100;
   N     : Integer;
begin
   ... -- Code that assigns a value to N
   Score := N;
   -- A run-time check verifies that N is within the range 1..100
   -- If this check fails, the Constraint_Error exception is raised
end;
```

3.1.2.2 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

   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: out Float);
     with Pre  => Contains(T, Item),
          Post => not Contains(T, Item);

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

(continues on next page)

(continued from previous page)

```
   ...
private
   ... -- Full declaration of type Table
end Table_Pkg;

package body Table_Pkg is
   ... -- Implementation of Insert, Remove, Is_Full, and Contains
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.

3.1.2.3 Programming in the large

The original Ada 83 design introduced the package construct, a feature that supports encapsulation (*information hiding*) and modularization, and which allows the developer to control the namespace that is accessible within a given compilation unit. Ada 95 introduced the concept of *child units*, adding considerable flexibility and easing the design of very large systems. Ada 2005 extended the language's modularization facilities by allowing certain kinds of mutual references between package specifications, thus making it easier to interface with languages such as Java.

3.1.2.4 Generic Templates

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

3.1.2.5 Object-Oriented Programming (OOP)

Ada 83 was object-based, allowing the partitioning of a system into modules 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 GUIs that do not have real-time constraints and that could be most effectively developed using OOP features. In part for this reason, Ada 95 supplies comprehensive support for OOP, through its *tagged type* facility: classes, polymorphism, inheritance, and dynamic binding. Ada 95 does not require automatic garbage collection but rather supplies definitional features allowing the developer to supply type-specific storage reclamation operations (*finalization*). Ada 2005 brought additional OOP features including Java-like interfaces and traditional `obj.op(...)` operation invocation notation.

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 17, 22] or[24] for more details.

[24] *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/.

3.1.2.6 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 multi-core architectures were introduced in Ada 2012.

3.1.2.7 Systems Programming

Both in the *core* language and the Systems Programming Annex, Ada supplies the necessary features for 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 be written in Ada, using the protected type facility.

3.1.2.8 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 task dispatching policy that basically requires tasks to run until blocked or preempted. Subsequent versions of the language standard introduced several other policies, such as Earliest Deadline First.

3.1.2.9 High-Integrity Systems

With its emphasis on sound software engineering principles, Ada supports the development of high-integrity applications, including those that need to be certified against safety standards such EN 50128 for rail systems, DO - 178C/ED - 12C[25] for avionics, and security standards such as the Common Criteria[26]. 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 not be accessed via inappropriate operations; errors such as treating pointers as integers (or vice versa) are prevented.

- *Array bounds checking*

  A run-time check guarantees that an array index is within the bounds of the array. This prevents buffer overflow 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.

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

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

- *Prevention of dangling references*

  A scope accessibility check ensures that a pointer cannot reference an object on the stack after exit/return from the scope (block or subprogram) in which the object is declared. Such checks are generally static, with no run-time overhead.

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

The evolution of Ada has seen the continued increase in support for safety-critical and high-security applications. Ada 2005 standardized the Ravenscar Profile, a collection of concurrency features that are powerful enough for real-time programming but simple enough to make certification and formal analysis 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.

3.1.2.10 Summary

In brief, Ada is an internationally standardized language combining object-oriented programming features, well-engineered concurrency facilities, real-time support, and built-in reliability through both compile-time and run-time checks. As such it is an appropriate language for addressing the real issues facing software developers today. Ada has a long and successful history and is used throughout a number of major industries to design software that protects life and property.

3.2 SPARK

SPARK is a software development technology (programming language and verification toolset) specifically designed for engineering ultra-low defect level applications, for example where safety and/or security are key requirements. SPARK Pro is AdaCore's commercial-grade offering of the SPARK technology. The main component in the toolset is GNATprove, which performs formal verification on SPARK code.

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

The SPARK language has been stable over the years, with periodic enhancements. The 2014 version of SPARK represented a major revision[27],[28]), incorporating contract-based programming syntax from Ada 2012, and subsequent upgrades included support for pointers (access types) based on the Rust ownership model.

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

[28] *SPARK Reference Manual, Release 2020*. AdaCore and Altran, 2020. URL: https://www.adacore.com/uploads/techPapers/spark_rm_community_2020.pdf.

3.2.1 Flexibility

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.

3.2.2 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, the language supports mathematical proof and can thus provide high confidence that the software meets a range of assurance requirements: from the absence of run-time errors (*AORTE*), to the enforcement of safety or security properties, to compliance with a formal specification of the program's required behavior.

3.2.3 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[29] can quickly bring developers up to speed; users are assumed to be experts in their own application domain such as railway software and do not need to be familiar with formal methods or the proof technology implemented by the toolset. In effect, SPARK Pro is an advanced static analysis tool that will detect many logic errors 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.

3.2.4 Hybrid Verification

SPARK supports *hybrid verification*, which combines testing and formal proofs. As an example, an existing project in Ada and C can adopt SPARK to implement new functionality for critical components. The SPARK units can be analyzed statically to achieve the desired level of verification, with testing performed at the interfaces between the SPARK units and the modules in the other languages.

3.2.5 Reduced Cost and Improved Efficiency of Executable Object Code Verification

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, they may fail to detect errors. SPARK addresses this issue by allowing automated proof to be used to demonstrate functional correctness at the subprogram level, either in combination with or as a replacement for unit

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

testing. In the high proportion of cases where proofs can be discharged automatically, the cost of writing unit tests is completely avoided. Moreover, verification by proofs covers all execution conditions and not just a sample.

3.3 GNAT Pro Assurance

3.3.1 Sustained Branches

GNAT Pro Assurance is a specialized version of the GNAT Pro development environment, available for any of the products in the GNAT Pro family: GNAT Pro for Ada, GNAT Pro for C, GNAT Pro for C++, and GNAT Pro for Rust. It is targeted to projects requiring customized support, including bug fixes and *known problems* analyses, on a specific version of the toolchain. This service is especially suitable for applications with long maintenance cycles or certification requirements, since critical updates to the compiler or other product components may become necessary years after the initial release. Such customized maintenance of a specific version of the product is known as a *sustained branch*.

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 Language and Tool Support

GNAT Pro Assurance for Ada supports all versions of the Ada language standard as well as multiple versions of C (C89, C99, and C11). It provides an Integrated Development Environment (see *Integrated Development Environments (IDEs)* (page 29)), a comprehensive toolsuite including a visual debugger, and an extensive set of libraries and bindings. Details on the GNAT Pro for Ada toolchain may be found in[30]. AdaCore's GNAT project facility, based on a multi-language builder for systems organized into subsystems and libraries, is documented in[31].

3.3.3 Configurable Run-Time Libraries

Two specific GNAT-defined run-time libraries have been designed with certification in mind and are known as the Certifiable Profiles (see[32]):

- *Light Profile*
- *Light-Tasking Profile*

The Light Profile provides a flexible Ada subset that is supported by a certifiable Ada run-time library. Depending on application requirements, this profile can be further restricted through the `Restrictions` pragma, with the application only including run-time code that is used by the application.

These run-time libraries can also be customized directly to suit certification requirements: unneeded packages can be removed to allow for self-certification of the runtime, while the `-nostdlib` linker switch can be used to prevent the use of the runtime. Even when the run-time library is suppressed, some run-time sources are still required to provide compile-time definitions. While this code produces no object code, the certification protocol may still require tests to ensure correct access to these definitions.

[30] AdaCore. GNAT User's Guide for Native Platforms. URL: https://docs.adacore.com/live/wave/gnat_ugn/html/gnat_ugn/gnat_ugn.html.
[31] AdaCore. GPRbuild and GPR Companion Tools User's Guide. URL: https://docs.adacore.com/gprbuild-docs/html/gprbuild_ug.html.
[32] AdaCore. GNAT User's Guide Supplement for Cross Platforms. URL: https://docs.adacore.com/live/wave/gnat_ugx/html/gnat_ugx/gnat_ugx.html.

The Light-Tasking Profile expands the Light Profile to include Ravenscar tasking support, allowing developers to use concurrency in their certification applications.

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

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 Assurance 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.

3.3.4 Full Implementation of Ada Standards

GNAT Pro provides a complete implementation of the Ada language from Ada 83 to Ada 2012, and support for selected features from Ada 2022. Developers of safety-critical and high-security systems can thus take advantage of features such as contract-based programming, which effectively embed requirements in the source program text and simplify verification.

3.3.5 Source to Object Traceability

A 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.

3.3.6 Safety-Critical Support and Expertise

At the heart of every AdaCore subscription are the support services that AdaCore provides to its customers. AdaCore staff are recognized experts on the Ada language, software certification standards in several domains, compilation technologies, and static and dynamic verification. They have extensive experience in supporting customers in railway, avionics, space, energy, air traffic management/control, automotive, and military projects. Every AdaCore product comes with front-line support provided directly by these experts, who are also the developers of the technology. This ensures that customers' questions (requests for guidance on feature usage, suggestions for technology enhancements, or defect reports) are handled efficiently and effectively.

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

3.3.7 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

3.3.8 GNATstack

Included with GNAT Pro is GNATstack, a static analysis tool 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.

3.4 GNAT Static Analysis Suite (GNAT SAS)

3.4.1 Defects and Vulnerability Analyzer

GNAT SAS features an Ada source code analyzer that detects run-time and logic errors. It assesses potential bugs and vulnerabilities before program execution, serving as an automated peer reviewer, helping to find errors easily at any stage of the development life-cycle. It helps improve code quality and makes it easier to perform safety and/or security analysis.

The defects and vulnerability analyzer can detect several of the "Top 25 Most Dangerous Software Errors" in the Common Weakness Enumeration. It is a stand-alone tool that runs on Windows and Linux platforms and may be used with any standard Ada compiler or fully integrated into the GNAT Pro development environment.

3.4.2 GNATmetric

The GNATmetric tool analyzes source code to calculate a set of commonly used industry metrics, thus allowing developers to estimate the size and better understand the structure of the source code. This information also facilitates satisfying the requirements of certain software development frameworks.

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, 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[33], which formats Ada source code according to the GNAT coding style[34], 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.

3.5 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 22)): augmenting testing with the use of SPARK to formally prove relevant properties of the private subprogram.

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

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

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 and/or GNATfuzz 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.

3.6 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 railway 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 23)) that strikes the balance between tool stability and project flexibility.

3.7 Integrated Development Environments (IDEs)

3.7.1 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.

Tools

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

Robust, Flexible and Extensible

Especially suited for large, complex systems, GNAT Studio can import existing projects from other Ada implementations while adhering to their file naming conventions and retaining the existing directory organization. Through the 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.

Easy to learn, easy to use

GNAT Studio is intuitive to new users thanks to its menu-driven interface with extensive online help (including documentation 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.

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.

3.7.2 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)

3.7.3 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.

3.7.4 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.

CHAPTER

FOUR

ADACORE CONTRIBUTIONS TO THE SOFTWARE QUALITY ASSURANCE PLAN

This chapter identifies AdaCore's tools and technologies that support the techniques and measures defined in the EN 50128 Annex A tables and that can be cited accordingly in the Software Quality Assurance Plan. The information is presented in the form of annotations on the relevant tables in Annex A. These annotations indicate whether a technique or measure is covered by an AdaCore tool or technology and, if so, a comment on how the tool or technology contributes is provided.

Summary of abbreviations:

- **M** → Mandatory
- **HR** → Highly Recommended
- **R** → Recommended
- **---** → Optional (neither Recommended nor Not Recommended)
- **NR** → Not Recommended

4.1 Table A.3 – Software Architecture (7.3)

The Ada language and AdaCore technology do not provide support for software architecture *per se*, but rather are more targeted towards software component design. However, the presence of some capabilities at the lower level may enable certain design decisions at a higher level. This table offers some guidance on how that can be done.

Technique/Measure	Ref	SIL 2	SIL 3/4	Covered	Comment
Defensive Programming	D.14	HR	HR	Yes	Defensive programming is more a component or a programming activity than an architecture activity per se, but as it is recorded in this table, it's worth mentioning that the Ada language provides several features addressing various objectives of defensive programming techniques (e.g., exception handling). In addition, the GNAT Static Analysis Suite and SPARK tools help identify pieces of code that should be protected by defensive code.
Fault Detection & Diagnosis	D.26	R	R	No	
Error Correcting Codes	D.19	—	—	No	
Error Detecting Codes	D.19	R	HR	No	
Failure Assertion Programming	D.24	R	HR	Yes	The Ada language allows formalizing assertions and contracts in various places in the code.
Safety Bag Techniques	D.47	R	R	No	
Diverse Programming	D.16	R	HR	Yes	Using Ada along with another language and a different code generation technology can be used to contribute to the diverse programming argument.
Recovery Block	D.44	R	R	No	
Backward Recovery	D.5	NR	NR	No	
Forward Recovery	D.30	NR	NR	No	
Retry Fault Recovery Mechanisms	D.46	R	R	No	
Memorising Executed Cases	D.36	R	HR	No	
Artificial Intelligence – Fault Correction	D.1	NR	NR	No	
Dynamic Reconfiguration of software	D.17	NR	NR	No	
Software Error Effect Analysis	D.25	R	HR	No	

continues on next page

Table 1 – continued from previous page

Technique/Measure	Ref	SIL 2	SIL 3/4	Covered	Comment
Graceful Degradation	D.31	R	HR	No	
Information Hiding / Encapsulation	D.33	HR	HR	Yes	The Ada language provides the necessary features to separate the interface of a module from its implementation and enforce this separation.
Fully Defined Interface	D.38	HR	M	Yes	The Ada language provides the necessary features to separate the interface of a module from its implementation and enforce this separation.
Formal Methods	D.28	R	HR	Yes	SPARK can be used to formally define architectural properties, such as data flow, directly in the code and provides the means to verify them.
Modeling	Table A.17	R	HR	Yes	Ada and SPARK allow defining certain modeling properties in the code and provide means to verify them.
Structured Methodology	D.52	HR	HR	Yes	Structured Methodology designs can be implemented with Ada.
Modeling supported by computer aided design and specification tools	Table A.17	R	HR	No	

4.2 Table A.4 – Software Design and Implementation (7.4)

Technique/Measure	Ref	SIL 2	SIL 3/4	Covered	Comment
Formal Methods	D.28	R	HR	Yes	Component requirements and interfaces can be written in the form of formal boolean properties, using the Ada or SPARK languages. These properties are verifiable.
Modeling	Table A.17	HR	HR	Yes	Ada and SPARK allow defining certain modeling properties in the code and provide means to verify them.
Structured methodology	D.52	HR	HR	Yes	Structured Methodology designs can be implemented with Ada.

continues on next page

Table 2 – continued from previous page

Technique/Measure	Ref	SIL 2	SIL 3/4	Covered	Comment
Modular Approach	D.38	M	M	Yes	A module can be represented as an Ada package, with a cohesive and well-defined functionality, a clear external interface in the visible part of the package spec, a private part whose visibility is limited to to its child units, and a body containing the implementation (which is only visible to its subunits).
Components	Table A.20	HR	HR	Yes	A component can be defined as a set of Ada packages, which can clearly define the interface to access the internal data, and the interfaces can be fully and unambiguously defined. This set of packages is typically identified within a project file (GPR file) and can be put into a version control system.
Design and Coding Standards	Table A.12	HR	M	Yes	There are available references for the coding standard. Verification can be automated in different ways: the GNAT compiler can define base coding standard rules to be checked at compile-time, with GNATcheck implementing a wider range of rules.
Analyzable Programs	D.2	HR	HR	Yes	The Ada language provide native features to improve program analysis, such as type ranges, parameter modes, and encapsulation. Tools such as GNATmetric and GNATcheck can help monitor the complexity of the code and prevent the use of overly complex code. GNAT SAS allows making an assessment of program analyzability during its development. For higher SILs, the use of SPARK ensures that the subset of the language used is suitable for most the rigorous analysis.
Strongly Typed Programming Language	D.49	HR	HR	Yes	Ada is a strongly typed language.
Structured Programming	D.53	HR	HR	Yes	Ada supports all the usual paradigms of structured programming. In addition, GNATcheck can control additional design properties, such as explicit control flows, where subprograms have single entry and single exit points, and structural complexity is reduced.

continues on next page

Table 2 – continued from previous page

Technique/Measure	Ref	SIL 2	SIL 3/4	Covered	Comment
Programming Language	Table A.15	HR	HR	Yes	Ada can be used for most of the development, while facilitating interfacing to other languages such as C or assembly.
Language Subset	D.35	—	HR	Yes	Ada is designed to support subsetting, possibly under the control of specific runtimes, GNATcheck, or with SPARK. Another possibility is to follow the recommendations in[35].
Object-Oriented Programming	Table A.22, D.57	R	R	Yes	If needed, Ada supports all the usual paradigms of object-oriented programming, in addition to safety-related features such as the Liskov Substitution Principle.
Procedural Programming	D.60	HR	HR	Yes	Ada supports all the usual paradigms of procedural programming.
Metaprogramming	D.59	R	R	No	

4.3 Table A.5 – Verification and Testing (6.2 and 7.3)

Technique/Measure	Ref	SIL 2	SIL 3/4	Covered	Comment
Formal Proof	D.29	R	HR	Yes	When Ada pre- and post-conditions are used, together with the SPARK subset of the language, formal methods can formally verify compliance of the implementation with these contracts.
Static Analysis	Table A.19	HR	HR	Yes	See Table A.19
Dynamic Analysis and Testing	Table A.13	HR	HR	Yes	See Table A.13

continues on next page

[35] ISO/IEC JTC1/SC22. Working Draft 3.6 - Programming Languages - Guide for the Use of the Ada Programming Language in High Integrity Systems. ISO/IEC PDTR 15942, International Organization for Standardization, August 1998. URL: https://www.open-std.org/JTC1/SC22/WG9/n350.pdf.

Table 3 – continued from previous page

Technique/Measure	Ref	SIL 2	SIL 3/4	Covered	Comment
Metrics	D.37	R	R	Yes	GNATmetric can compute and report metrics, such as code size, comment percentage, cyclomatic complexity, unit nesting, and loop nesting. These can then be compared with standards.
Traceability	D.58	HR	M	No	
Software Error Effect Analysis	D.25	R	HR	Yes	GNAT Studio supports code display and navigation. GNAT SAS can identify likely error locations in the code. This supports potential software error detection and analysis throughout the code.
Test Coverage for code	Table A.21	HR	HR	Yes	See Table A.21
Functional / Black-Box Testing	Table A.14	HR	HR	Yes	See Table A.14
Performance Testing	Table A.18	HR	HR	No	
Interface Testing	D.34	HR	HR	Yes	Ada's strong typing, together with its contract-based programming support, provides increased assurance to demonstrate that the software interfaces are correct. This can help improve software-to-software integration testing.

4.4 Table A.6 – Integration (7.6)

Technique/Measure	Ref	SIL 2	SIL 3/4	Covered	Comment
Functional and Black-box testing	Table A.14	HR	HR	Yes	GNATtest can generate a framework for testing.
Performance Testing	Table A.18	R	HR	Yes	Stack consumption can be statically computed using the GNATstack tool.

4.5 Table A.7 – Overall Software Testing (6.2 and 7.7)

Technique/Measure	Ref	SIL 2	SIL 3/4	Covered	Comment
Performance Testing	Table A.18	HR	M	Yes	Stack consumption can be statically computed using the GNATstack tool.
Functional and Black-box Testing	Table A.14	HR	M	Yes	GNATtest can generate a testing framework for testing.
Modeling	Table A.17	R	R	No	

4.6 Table A.8 – Software Analysis Techniques (6.3)

Technique/Measure	Ref	SIL 2	SIL 3/4	Covered	Comment
Static Software Analysis	D.13, D.37, Table A.19	HR	HR	Yes	See Table A.19.
Dynamic Software Analysis	Tables A.13, A.14	R	HR	Yes	See Tables A.13 and A.14.
Cause Consequence Diagrams	D.6	R	R	No	
Event Tree Analysis	D.22	R	R	No	

continues on next page

Table 6 – continued from previous page

Technique/Measure	Ref	SIL 2	SIL 3/4	Covered	Comment
Software Error Effect Analysis	D.25	R	HR	Yes	GNAT Studio supports code display and navigation. GNAT SAS can identify likely error locations in the code. These tools support both detection of potential software errors and analysis throughout the code.

4.7 Table A.9 – Software Quality Assurance (6.5)

Although AdaCore doesn't directly provide services for ISO 9001 or configuration management, it follows standards to enable tool qualification and/or certification. The following table only lists items that can be useful to third parties.

Technique/Measure	Ref	SIL 2	SIL 3/4	Covered	Comment
Accredited to EN ISO 9001	7.1	HR	HR	No	
Compliant with EN ISO 9001	7.1	M	M	No	
Compliant with ISO/IEC 90003	7.1	R	R	No	
Company Quality System	7.1	M	M	No	
Software Configuration Management	D.48	M	M	No	
Checklists	D.7	HR	M	No	
Traceability	D.58	HR	M	No	
Data Recording and Analysis	D.12	HR	M	Yes	The data produced by tools can be written to files and placed under configuration management.

4.8 Table A.10 – Software Maintenance (9.2)

Technique/Measure	Ref	SIL 2	SIL 3/4	Covered	Comment
Impact Analysis	D.32	HR	M	Yes	GNAT SAS contributes to identifying the impact of a code change between two baselines.
Data Recording and Analysis	D.12	HR	M	Yes	AdaCore tools can be invoked from the command line. They produce result files including the date and version of the tool used.

4.9 Table A.11 – Data Preparation Techniques (8.4)

Technique/Measure	Ref	SIL 2	SIL 3/4	Covered	Comment
Tabular Specification Methods	D.68	R	R	Yes	Tables of data can be expressed using the Ada language, together with type-wide contracts (predicates or invariants).
Application specific language	D.69	R	R	No	
Simulation	D.42	HR	HR	No	
Functional testing	D.42	M	M	No	
Checklists	D.7	HR	M	No	
Fagan inspection	D.23	HR	HR	No	
Formal design reviews	D.56	HR	HR	Yes	GNAT Studio can display code and navigate through the code as a support for walkthrough activities.
Formal proof of correctness	D.29	—	HR	Yes	When contracts are expressed within the SPARK subset, their correctness can be formally verified.
Walkthrough	D.56	R	HR	Yes	GNAT Studio can display code and navigate through the code as a support for walkthrough activities.

4.10 Table A.12 – Coding Standards

There are available references for coding standards. Their verification can be automated through different ways: the GNAT compiler can define base coding standard rules to be checked at compile time, and GNATcheck implements a wider range of rules and is tailorable to support project-specific coding standards.

Technique/Measure	Ref	SIL 2	SIL 3/4	Covered	Comment
Coding Standard	D.15	HR	M	Yes	GNATcheck allows implementing and verifying a coding standard.
Coding Style Guide	D.15	HR	HR	Yes	GNATcheck allows implementing and verifying a coding style guide.
No Dynamic Objects	D.15	R	HR	Yes	GNATcheck can forbid the use of dynamic objects.
No Dynamic Variables	D.15	R	HR	Yes	GNATcheck can forbid the use of dynamic variables.
Limited Use of Pointers	D.15	R	R	Yes	GNATcheck can forbid the use of pointers or force justification of their usage.
Limited Use of Recursion	D.15	R	HR	Yes	GNATcheck can forbid the use of recursion or force justification of their usage.
No Unconditional Jumps	D.15	HR	HR	Yes	GNATcheck can forbid the use of unconditional jumps.
Limited size and complexity of Functions, Subroutines and Methods	D.38	HR	HR	Yes	GNATmetric can compute complexity measures, and GNATcheck can report excessive complexity.
Entry/Exit Point strategy for Functions, Subroutines and Methods	D.38	HR	HR	Yes	GNATcheck can verify rules related to exit points.
Limited number of subroutine parameters	D.38	R	R	Yes	GNATcheck can limit the number of parameters for subroutines and report when that number is exceeded.

continues on next page

Table 10 – continued from previous page

Technique/Measure	Ref	SIL 2	SIL 3/4	Covered	Comment
Limited use of Global Variables	D.38	HR	M	Yes	GNATcheck can flag global variable usage and enforce their justification. SPARK forbids function side effects and enforces documentation and verification of uses of global variables. GNAT Studio allows analyzing usage of global variables.

4.11 Table A.13 – Dynamic Analysis and Testing

Technique/Measure	Ref	SIL 2	SIL 3/4	Covered	Comment
Test Case Execution from Boundary Value Analysis	D.4	HR	HR	Yes	GNATtest can generate and execute a testing framework for tests written by developers from requirements.
Test Case Execution from Error Guessing	D.20	R	HR	Yes	GNAT fuzz can automate the generation of large numbers of test cases at a high frequency.
Test Case Execution from Error Seeding	D.21	R	HR	No	
Performance Modeling	D.39	R	HR	No	
Equivalence Classes and Input Partition Testing	D.18	R	HR	Yes	Ada and SPARK provide specific features for partitioning function input and verifying that this partitioning is well formed (i.e., no overlap and no gaps).
Structure-Base Testing	D.50	R	HR	Yes	See Table A.21

4.12 Table A.14 – Functional/Black Box Test

GNATtest can generate and execute a testing framework, with the actual tests being written by developers from requirements.

Technique/Measure	Ref	SIL 2	SIL 3/4	Covered	Comment
Test Case Execution from Cause Consequence Diagrams	D6	—	R	No	
Prototyping/ Animation	D.43	—	R	No	
Boundary Value Analysis	D.4	R	HR	Yes	GNATtest can be used to implement tests coming from boundary value analysis.
Equivalence Classes and Input Partitioning Testing	D.18	R	HR	Yes	Ada and SPARK provide specific features for partitioning function input and verifying that this partitioning is well formed (i.e., no overlap and no gaps).
Process Simulation	D.42 R	R	No		

4.13 Table A.15 – Textual Programming Language

Technique/Measure	SIL 2	SIL 3/4	Covered	Comment
Ada	HR	HR	Yes	GNAT Pro tools support all versions of the Ada language.
MODULA-2	HR	HR	No	
PASCAL	HR	HR	No	
C or C++	R	R	Yes	GNAT Pro for C and GNAT Pro for C++ support these languages
PL/M	R	NR	No	
BASIC	NR	NR	No	
Assembler	R	R	No	
C#	R	R	No	

continues on next page

Table 13 – continued from previous page

Technique/Measure	SIL 2	SIL 3/4	Covered	Comment
Java	R	R	No	
Statement List	R	R	No	

4.14 Table A.17 – Modeling

Technique/Measure	Ref	SIL 2	SIL 3/4	Covered	Comment
Data Modeling	D.65	R	HR	Yes	Ada allows modeling data constraints, in the form of type predicates.
Data Flow Diagram	D.11	R	HR	Yes	SPARK allows defining data flow dependencies at subprogram specification.
Control Flow Diagram	D.66	R	HR	No	
Finite State Machine or State Transition Programs	D.27	HR	HR	No	
Time Petri Nets	D.55	R	HR	No	
Decision/Truth Tables	D.13	R	HR	No	
Formal Methods	D.28	R	HR	Yes	SPARK allows defining formal properties on the code that can be verified by the SPARK toolset.
Performance Modeling	D.39	R	HR	No	
Prototyping/Animation	D.43	R	R	No	
Structure Diagrams	D.51	R	HR	No	
Sequence Diagrams	D.67	R	HR	No	

4.15 Table A.18 – Performance Testing

Technique/Measure	Ref	SIL 2	SIL 3/4	Covered	Comment
Avalanche/Stress Testing	D.3	R	HR	No	Ada allows modeling data constraints, in the form of type predicates.
Response Timing and Memory Constraints	D.45	HR	HR	Yes	GNATstack can statically analyze stack usage.
Performance Requirements	D.40	HR	HR	No	

4.16 Table A.19 – Static Analysis

Technique/Measure	Ref	SIL 2	SIL 3/4	Covered	Comment
Boundary Value Analysis	D.4	R	HR	Yes	GNAT SAS can compute boundary values for variables and parameters from the source code. GNAT SAS and SPARK can verify various properties by analyzing potential values and boundary values of variables. This includes detecting errors such as dereferencing a pointer that could be null, generating a value outside the bounds of an Ada type or subtype, violating a memory safety constraint (*buffer overrun*), generating a numeric overflow or wraparound, and dividing by zero. GNAT SAS and SPARK also help to confirm expected boundary values of variables and parameters coming from the design.
Checklists	D.7	R	R	No	

continues on next page

Table 16 – continued from previous page

Technique/Measure	Ref	SIL 2	SIL 3/4	Covered	Comment
Control Flow Analysis	D.8	HR	HR	Yes	GNAT SAS and SPARK can detect suspicious and potentially incorrect control flows, such as unreachable code, redundant conditionals, loops that either run forever or fail to terminate normally, and subprograms that never return. GNATstack can analyze control flow and compute the maximum amount of stack memory for each task. More generally, GNAT Studio provides visualization for call graphs and call trees.
Data Flow Analysis	D.10	HR	HR	Yes	GNAT SAS and SPARK can detect suspicious and potentially incorrect data flow, such as variables being read before they are written (uninitialized variables), and values that are written to variables without being read (redundant assignments).
Error Guessing	D.20	R	R	Yes	Although realized through dynamic rather than static analysis, GNAT fuzz can automatically generate test cases to support Error Guessing.
Walkthroughs/Design Reviews	D.56	HR	HR	Yes	GNAT Studio can display and navigate the code, supporting walkthrough activities.

4.17 Table A.20 – Components

Technique/Measure	Ref	SIL 2	SIL 3/4	Covered	Comment
Information Hiding	D.33	—	—	Yes	See Information Encapsulation below.
Information Encapsulation	D.33	HR	HR	Yes	Ada provides the necessary features to separate the interface of a module from its implementation, and enforce this separation.
Parameter Number Limit	D.38	R	R	Yes	GNATcheck can limit the number of parameters for subroutines, and report violations.

continues on next page

Table 17 – continued from previous page

Technique/Measure	Ref	SIL 2	SIL 3/4	Covered	Comment
Fully Defined Interface	D.38	HR	M	Yes	Ada offers many features to support interface definition, including behavior specification through pre- and post-conditions.

4.18 Table A.21 – Test Coverage for Code

Technique/Measure	Ref	SIL 2	SIL 3/4	Covered	Comment
Statement	D.50	HR	HR	Yes	GNATcoverage provides statement-level coverage capabilities.
Branch	D.50	R	HR	Yes	GNATcoverage provides branch-level coverage capabilities.
Compound Condition	D.50	R	HR	Yes	GNATcoverage provides MC/DC (Modified Condition/Decision Coverage) capabilities, which can be used as an efficient alternative to Compound Condition coverage.
Data Flow	D.50	R	HR	No	
Path	D.50	R	NR	No	

4.19 Table A.22 – Object Oriented Software Architecture

Technique/Measure	Ref	SIL 2	SIL 3/4	Covered	Comment
Traceability of the concept of the application domain to the classes of the architecture	—	R	HR	No	
Use of suitable frames, commonly used combinations of classes and design patterns	—	R	HR	Yes	The conventional OO design patterns can be implemented in Ada.
Object Oriented Detailed Design	Table A.23	R	HR	Yes	See Table A.23

4.20 Table A.23 – Object Oriented Detailed Design

Technique/Measure	Ref	SIL 2	SIL 3/4	Covered	Comment
Class should have only one objective	—	R	HR	Yes	It's possible in Ada to write classes with a unique objective.
Inheritance used only if the derived class is a refinement of its basic class	—	HR	HR	Yes	Ada and SPARK can enforce respecting the Liskov Substitution Principle, ensuring inheritance consistency.
Depth of inheritance limited by coding standards	—	R	HR	Yes	GNATcheck can limit inheritance depth.
Overriding of operations (methods) under strict control	—	R	HR	Yes	Ada can enforce explicit syntax for overriding methods.

continues on next page

Table 20 – continued from previous page

Technique/Measure	Ref	SIL 2	SIL 3/4	Covered	Comment
Multiple inheritance used only for interface classes	—	HR	HR	Yes	Ada only allows multiple inheritance from interfaces.
Inheritance from unknown classes	—	—	NR	Yes	Ada only allows inheritance from known classes.

CHAPTER

FIVE

TECHNOLOGY USAGE GUIDE

This chapter explains how AdaCore's tools and technologies support a variety of techniques from Annex D.

5.1 Analyzable Programs (D.2)

The Ada language has been designed to increase program specification expressiveness and verification. Explicit constraints at the code level can be used as the basis of both manual analysis (inspection), such as code reviews, and automatic analysis, ranging from the compiler's semantic consistency checks to the SPARK tools' formal proof of program properties.

Examples of Ada language features supporting analysis include:

- type and subtype ranges and predicates
- parameter modes and subprogram contracts
- packages and private types (encapsulation)
- the Ravenscar concurrency profile
- minimal set of implicit or undefined behaviors

Tools such as GNATmetric and GNATcheck allow monitoring the complexity and quality of the code and identifying potentially problematic constructs. This is accomplished through techniques such as basic code size metrics, cyclomatic complexity computation, and coupling analysis.

GNAT SAS identifies potential run-time errors in the code. The number of false positive results depends on the code complexity. A high number of false positives is often a symptom of overly-complicated code. Using GNAT SAS during development allows finding locations in the code that are overly complex and provides information on what needs to be improved.

The SPARK language is much more ambitious in analyzing programs, at the extreme supporting full correctness proofs against formally specified requirements. It structurally forbids features such as exceptions, which complicate or prevent formal analysis. Code that is hard to analyze is often hard to understand and maintain, and conversely. Using SPARK as part of the development phase thus results in code that is not only maximally analyzable but also clear and readable.

During code review phases, GNAT Studio offers a variety of features that can be used for program analysis, in particular call graphs, reference searches, and other code organization viewers.

5.2 Boundary Value Analysis (D.4)

The objective of this technique is to verify and test the behavior of a subprogram at the limits and boundary values of its parameters. AdaCore's technologies can provide complementary assurance on the quality of this analysis and potentially decrease the number of tests that need to be performed.

Ada's strong typing allows refining types and variables boundaries. For example:

```
type Temperature is new Float range -273.15 .. 1_000;
V : Temperature;
```

Additionally, it's possible to define the specific behavior of values at various locations in the code. For example, it's possible to define relationships between the input and output of a subprogram, in the form of a partitioning of the input domain:

```
function Compute (J : Integer) return Integer
   with Contract_Cases => (J = Integer'First => Compute'Result = -1,
                           J = Integer'Last  => Compute'Result = 1,
                           others            => J - 1);
```

The above shows an input partition of one parameter (but it can also be a combination of several parameters). The behavior on the boundaries of J is specified and can then either be tested (for example, with enabled assertions) or formally proven with SPARK. Further discussion of input partitioning can be found in the context of *Equivalence Classes and Input Partition Testing (D.18)* (page 55).

Another possibility is to use GNAT SAS to identify possible values for variables, and propagate those values from call to call, constructing lists and/or ranges of potential values for each variable at each point of the program. These are used as the input to run-time error analysis. When used in full-soundness mode, GNAT SAS provides guarantees that the locations it reports on the code are the only ones that may have run-time errors, thus allowing a reduction of the scope of testing and review to only these places.

However, it's important to stress that GNAT SAS is only performing this boundary value analysis with respect to potential exceptions and robustness. No information is provided regarding the correctness of the values produced by subprograms.

GNAT SAS also has the capacity to display the possible values of variables and parameters. This can be used as a mechanism to increase confidence that testing has taken into account all possible boundaries for values.

SPARK has the ability to perform similar absence of run-time errors (AORTE) analysis, thus reaching the same objectives. In addition to the above, when requirements can be described in the form of boolean contracts, SPARK can demonstrate correctness of the relation between input and output on the entire range of values.

5.3 Control Flow Analysis (D.8)

Control flow analysis requires identifying poor and incorrect data structures, including unreachable code and useless tests in the code (such as conditions that are always true).

GNAT Studio can display call graphs between subprograms, allowing visualization and analysis of control flow in the application.

GNAT SAS contributes to control flow analysis by identifying unreachable code, as well as conditions being always true or always false. This analysis is partial and needs to be completed with other techniques such as code review or code coverage analysis, which together will allow reaching higher levels of confidence.

GNATmetric can compute coupling metrics between units, helping to identify loosely or tightly coupled units.

GNATstack computes worst-case stack consumption based on the application's call graph. This can help identify poorly structured code which consumes too much memory on some sequences of calls.

5.4 Data Flow Analysis (D.10)

The GNAT Pro toolchain can be configured to detect uninitialized variables at run-time through the use of the pragma `Initialize_Scalars`. With this pragma, all scalars are automatically initialized to either an out-of-range value (if there is one) or to an *unusual* value (either the largest or smallest). This significantly improves detection at test time.

GNAT SAS can detect suspicious and potentially incorrect data flows, such as variables that are read before they are written (uninitialized variables), variables written more than once without being read (redundant assignments), and variables that are written but never read. This analysis is partial and needs to be completed with other techniques such as formal proof, code review or code coverage analysis, which together allow reaching higher levels of confidence.

SPARK performs this analysis and much more, allowing the specification and verification of data flow. This is used in the following activities:

- verification that all inputs and outputs have been specified, including possible side effects
- verification that all dependencies between inputs and outputs are specified
- verification that the implemented dataflow corresponds to the one specified

Here's an example:

```
procedure Compute (A, B, C : Integer; R1, R2 : out Integer)
   with Depends => (R1 => (A, B),
                    R2 => (B, C));

procedure Compute (A, B, C : Integer; R1, R2 : out Integer) is
begin
   R1 := A + B;
   R2 := A + B - C;
end Compute;
```

R1 is required to be computed from A and B, and R2 from B and C. However, in the procedure body, R2 also depends on A. SPARK's formal proof detects this error.

The error is likewise detected in the presence of branches:

```
procedure Compute (A, B, C : Integer; R1, R2 : out Integer) is
begin
   R1 := A + B;
   if A = 0 then
      R2 := B + C;
   else
      R2 := B - C;
   end if;
end Compute;
```

Here R2 depends on the result of the expression `A = 0`, so its value is actually computed from A, B and C, and not just B and C. As in the previous case, SPARK's formal analysis detects the error.

A similar result occurs when the dependence is indirect, through a subprogram call. Here's an example based on a logging procedure that has a global state, Screen, which is written to by the procedure:

```
procedure Log (V : String)
   with Global  => (Output => Screen),
        Depends => (Screen => V)

procedure Compute (A, B, C : Integer; R1, R2 : out Integer)
  with Depends => (R1 => (A, B),
                   R2 => (B, C));

procedure Compute (A, B, C : Integer; R1, R2 : out Integer) is
begin
   R1 := A + B;
   R2 := B + C;

   if A = 0 then
      Log ("A is 0");
   end if;
end Compute;
```

The data flow does not correspond to the specification: Compute should specify that it modifies Screen. So the incorrect code is detected. The error is detected whether or not a branch is present, serving as a useful complement to structural code coverage in many cases.

5.5 Defensive Programming (D.14)

As stated in sub-clause D.14, the goal of defensive programming is to "detect anomalous control flow, data flow, or data values ... and react to them in a predetermined and acceptable manner".

Ada's strong typing will avoid the need for many situations where constraints would be expressed in the form of defensive code. However, in some situations strong typing is not enough. This can be the case, for example, when accessing an element of an array. In this case, Ada allows expressing constraints in the specification, through preconditions, postconditions or predicates.

Beyond this, Ada provides specific support for a subset of what's specified in the D.14 annex. GNAT SAS and SPARK will allow the development of defensive programming in places where it makes the most sense.

Specific defensive code rules can also be defined in the coding standard and their verification can then be automated through code analysis using, for example, GNATcheck.

5.5.1 Data should be range checked

Ada offers types and subtypes that are naturally associated with ranges, e.g.:

```
subtype Percent is Integer range 0 .. 100;
-- Percent is the same type as Integer but with a run-time constraint on its range

X, Y : Integer;
V    : Percent;
...
V := X + Y; -- Raises exception Constraint_Error if X + Y is not in 0 .. 100
...
```

It's then the task of the developer to react to potential exceptions. Alternatively, it's possible to write explicit verification in the code to ensure that the expression is within its boundary:

```
V_Int : Integer;
V_Pct : Percent;
...
V_Int := X+Y;
if V_Int in Percent then
   V_Pct := V_Int;
else
  ... -- Respond to out-of-range result
end if;
```

Another way to proactively ensure the absence of range check failure is to use tools such as GNAT SAS or SPARK, which statically identify the only possible locations in the code where such failures can happen.

Note that run-time checks can be deactivated if needed for performance reasons, for example once thorough testing or formal proof has been performed.

5.5.2 Data should be dimension-checked

The GNAT Pro compiler provides a language extension for dimensional consistency analysis, which ensures that variables are properly typed according to their dimension. The system is implemented based on the seven base dimensions (meter, kilogram, second, ampere, kelvin, mole, candela), and will check that operations between these types are consistent. For example, a type Speed can be defined to represent distance per time. Consistency between these types is checked at compile time so that dimension errors will be reported as errors. For example:

```
D       : Distance := 10;
T       : Time     := 1;
S       : Speed    := D / T; -- OK
My_Time : Time     := 100;
...
Distance_Traveled := S / My_Time;
-- Error, resulting dimension is Distance / Time**2
-- The expression should be S * My_Time
```

5.5.3 Read-only and read-write parameters should be separated and their access checked

In Ada, the parameter mode is specified in parameter specifications and is checked by the compiler. For example, a read-only parameter is passed as mode **in** and may not be modified. A read-write parameter is passed either as mode **in out** or as mode **out** and is modifiable. (The **out** mode is appropriate if the parameter is written before being read). The compiler will produce an error for an attempted modification of **in** parameters and detect when an **in out** or **out** parameter is not modified and so could have been passed as **in**. For example:

```
procedure P (V : in Integer) is
begin
   V := 5; -- ERROR, V is mode "in"
end P;
```

5.6 Functions should treat all parameters as read-only

The original version of Ada required that functions could only have **in** parameters. This restriction was relaxed in a later version of the standard, but the original behavior can be

reverted through a GNATcheck rule. The SPARK Ada subset forbids functions with writable (i.e., `out` or `in out` parameters).

5.6.1 Literals and constants should not be write-accessible

Ada provides many kinds of literals (e.g. numeric, character, enumeration, string) and allows declaring constants of any type but ensures that their values can not be updated.

```
type Color is (Red, Blue, Green);
Answer    : constant Integer := 42;
One_Third : constant         := 1.0 / 3.0;
Greeting  : String           := "Hello";
```

The literals and constants are read-only as per language definition. For example, trying to pass Red or `Answer` to a subprogram as an `out` or `in out` parameter would be illegal. Note that `Greeting` is a variable and can be assigned to, but the literal `"Hello"` is immutable.

5.6.2 Using GNAT SAS and SPARK to drive defensive programming

GNAT SAS and SPARK identify locations where there are potential run-time errors — in other words, places where code is either wrong or where defensive programming should be deployed. This helps guide the writing of defensive code. For example:

```
procedure P (S : String; V : Integer) is
   C : Character;
begin
   ...
   C := S (V);
   ...
end P;
```

In the above code, there's a use of `V` as an index into the String `S`. GNAT SAS and SPARK will detect the potential for a run-time error. Protection of the code to prevent the error can take several forms:

Explicit test

The application code checks that `V` is in range before using its value as an index into the String. If the check fails, the appropriate recovery action can be taken (here the procedure simply returns).

```
procedure P (S : String; V : Integer) is
   C : Character;
begin
   ...
   if V not in S'Range then
      return;
   end if;
   C := S (V)
   ...
end P;
```

Precondition

Here the error is detected at call time. If assertion checking is enabled and the check fails, the `Assertion_Check` exception is raised.

```
procedure P (S : String; V : Integer)
   with Pre => V in S'Range
is
   C : Character;
begin
   ...
   C := S (V);
   ...
end P;
```

The main difference between GNAT SAS and SPARK is that GNAT SAS may miss some potential run-time errors (except when run only on small pieces of code if configured in "sound" mode), while SPARK requires the use of the appropriate Ada subset but is a sound technology (i.e., it will detect all potential run-time errors).

In general, the recommended Ada style is to use contracts instead of defensive code.

5.7 Coding Standards and Style Guide (D.15)

A coding standard can be defined using a combination of predefined rules (using GNAT options and GNATcheck rules) and appropriate arguments to pragma `Restrictions`.

5.8 Equivalence Classes and Input Partition Testing (D.18)

This technique involves partitioning the various potential inputs to subprograms and creating a testing and verification strategy based on this partitioning.

Ada extensions included in GNAT Pro for Ada can support partitioning at the source code level. The partition is a list of conditions for inputs together with their associated expected output, verifying the following criteria:

- The full set of all potential values is covered
- There is no overlap between partitions

These criteria can be verified either dynamically, by verifying at test time that all inputs exercised fall into one and only one partition, or formally by SPARK, proving that the partitions are indeed complete and disjoint.

Here's a simple example of such partitioning with two input variables:

```
function ArcTan (X, Y : Float) return Float with
   Contract_Cases =>
      (X >= 0 and Y >= 0 => ArcTan'Result >= 0        and ArcTan'Result <= PI/2,
       X <  0 and Y >= 0 => ArcTan'Result >= PI/2     and ArcTan'Result <= PI,
       X <  0 and Y < 0  => ArcTan'Result >= PI       and ArcTan'Result <= 3 * PI/
↪2,
       X >= 0 and Y < 0  => ArcTan'Result >= 3 * PI/2 and ArcTan'Result <= 2 * PI);
```

The presence of these contracts enable further verification. At run time, they act as assertions and allow verification that the form of the output indeed corresponds to the expected input. If SPARK is used, it's possible to formally verify the correctness of the relation between the input and properties.

5.9 Error Guessing (D.20)

The GNATfuzz tool for fuzz testing (part of the GNAT Dynamic Analysis Suite) supports the Error Guessing technique and can provide evidence for a system's robustness. GNATfuzz exercises a program with a large number of automatically generated test values, often random or malformed, and checks for crashes, hangs, and other anomalous behavior.

5.10 Failure Assertion Programming (D.24)

Ada offers a large variety of assertions (contracts) that can be defined in the code, either through pragmas or aspects.

- Pragma `Assert`

  This pragma allows verification within a sequence of statements:

```
A := B + C;
pragma Assert (A /= 0);
D := X / A;
```

- Pre- and postcondition contracts

  Pre- and postconditions can be defined as subprogram aspects:

```
procedure Double (X : in out Integer)
   with Pre  => X < 100,
        Post => X = X'Old * 2;
```

- Predicates and invariants

  Predicate and invariant contracts can be defined on types:

```
type Even is new Integer
with Dynamic_Predicate => Even mod 2 = 0;
```

These contracts can be checked dynamically, for example, during testing. The developer has fine control over which contracts can be removed (e.g. for improved performance) and which should remain in the deployed software.

The contracts can be used by the static analysis and formal proof tools. GNAT SAS uses contracts to refine its analysis and exploits them as assertions, even if it may not be able to demonstrate that they are correct. In this manner, contracts provide the tool with additional information on the code behavior. SPARK can go further and either prove their correctness, or else report its inability to do so. (In the latter case, the issue is either that the contract or the code is incorrect, or that the proof engine is not powerful enough to construct a proof.)

5.11 Formal Methods (D.28)

With SPARK, formal methods are used to define and check certain architectural properties, in particular for data coupling specification and verification. For example:

```
G : Integer;

procedure P (X, Y : Integer)
with Global => (Output => G),
     Depends => (G => (X, Y));
```

In the above example, the side effect of the subprogram is fully defined: `P` is modifying `G`. SPARK will check that this side effect, and no other, is present. `G` is specified as depending

on the values of X and Y. Again, SPARK will analyze the code to check that the variable relationships specified are correct.

In this example, an actual variable is used to define data flow. It's also possible to create an *abstract* state, implemented by a set of variables. Generally speaking, although these notations and verifications are quite useful on the lower levels of the architecture, they may not be that pertinent at higher levels. SPARK is flexible with regard to where this should be checked or where it should not.

At the lower level of the design phases, some properties and requirements can be refined or specified in the form of boolean expressions. SPARK will allow expressing these properties, including the formalism of first-order logic (quantifiers). These properties can be expressed in the form of subprogram preconditions, postconditions, type invariants and type predicates. For example:

```
-- P must have an input greater or equal to 10, and then has to modify V.
procedure P (V : in out Integer)
with Pre  => V     >= 10,
     Post => V'Old /= V;

-- Variables of type Even must be even
type Even is new Integer
with Dynamic_Predicate => Even mod 2 = 0;

-- Arrays of this type are always sorted in ascending order
type Sorted_Array is array (Integer range <>) of Integer
with Dynamic_Predicate =>
   Sorted_Array'Size <= 1 or else
   (for all I in Sorted_Array'First .. Sorted_Array'Last - 1 =>
       Sorted_Array (I) <= Sorted_Array (I + 1));
```

These properties can be formally verified through the SPARK toolset, using state of the art theorem proving methodologies. Testing aimed at verifying the correctness of these properties can then be simplified, if not entirely removed.

5.12 Impact Analysis (D.32)

Identifying the effect of a change on an entire software component requires the combination of various techniques, including reviews, testing and static analysis. GNAT SAS has specific features to identify the impact of a change from the perspective of potential runtime errors. It can establish a baseline with regard to potential failure analysis and filter only the potential defects that have been introduced or repaired following a change in the code.

GNAT Studio can provide call graphs and call trees, allowing the developer to see how a function is called in the software. This can be directly used in impact analysis.

5.13 Information Encapsulation (D.33)

Information encapsulation is good software engineering practice, enforcing access to data on a "need to know" basis and preventing hard-to-detect bugs from erroneous updates to global variables. Encapsulation has been intrinsic to the Ada design since the earliest version of the language and is embodied in the syntax and semantics of a variety of language features.

Ada's approach to encapsulation achieves similar methodological benefits to Object-Oriented Programming, but with a different syntax. In most OO languages, a class is both a type (which can be instantiated to produce objects) and a module (which can be separately compiled). Ada separates these concepts, modeling a class by a type (typically a private

type, as will be shown below) defined with a package (the main unit of modularization in Ada).

Separation of specification and body

The various program units in Ada — packages, tasks, subprograms, generic templates — have a structure that supports the separation of the unit's specification (its interface to other units) and its implementation (inaccessible externally). This physical separation not only supports encapsulation but also facilitates independent development of the two parts. For example, a package specification can be produced during the detailed design phase, with the body fleshed out later, perhaps by a different developer, during the implementation phase.

Package structure

A package comprises at least a specification and, if necessary, a body that implements the subprograms and other entities whose specifications are in the package specification. The package specification in general consists of a visible part and a private part. In a typical scenario, the visible part declares a type as **private**, along with subprogram specifications for the operations that are relevant to that type. The type and operations form the interface for that type. The private part of the package specification then provides the full declaration of the type, and the package body supplies the bodies for the subprograms defined for the type.

External to the package, the type name and the operations defined for the type are accessible, but the representational details for the full type declaration are hidden. This allows the designer of the package to modify the representation of the type during development or maintenance, without requiring source code changes to client code. This principle is sometimes referred to as *data abstraction*. Here is an example:

```
package Counters is
   type Counter is private;
   --  We don't want to give access to the representation of the counter here

   procedure Increment (C : in out Counter);
   procedure Print (C : in out Counter);

private

   type Counter is new Integer;
    -- Here, Counter is an Integer, but it could change to something
    -- else if needed without disturbing the interface.
end Counters;

with Ada.Text_IO;
package body Counters is
   procedure Increment (C : in out Counter) is
   begin
      C := C + 1;
   end Increment;

   procedure Print (C : in out Counter) is
   begin
      Ada.Text_IO.Put_Line (C'Img);
   end Print;
end Counters;
```

As a variation on this example, Ada supports encapsulation through *getter* and *setter* subprograms. Rather than directly manipulating a global variable declared in a package specification, the program can be structured to enforce accesses through a procedural interface:

```
package Data is
   function Value return Integer;       -- "Getter" function
   procedure Set (New_Value : Integer); -- "Setter" procedure
end Data;

package body Data is
   Global : Integer := 0;

   function Value return Integer is
   begin
      return Global;
   end Value;

   procedure Set (New_Value : Integer) is
   begin
      Value := New_Value;
   end Set;
end Data;
```

Concurrency

Both of Ada's tasking constructs — the task and the protected object — enforce encapsulation.

- A task object or task type specification defines its interface (its entries, which are used for synchronization and communication), and its body defines the implementation.
- A protected object or protected type specification defines its interface (entries and procedures, which are executed with mutual exclusion), and its body defines the implementation. The private part of a protected object/type specification encapsulates the data that is being protected: it can only be accessed externally through the entries and procedures that it defines.

Representation clauses

As another form of encapsulation, Ada's representation clause facility separates an entity's logical properties — its interface to client code — and its representation. For example:

```
type Alert is (Low, Medium, High);

type Packet is record
  Flag :   Boolean;
  Danger : Alert;
  Data :   Interfaces.Unsigned_8;
end record;

Byte : constant := 8;

for Alert use (Low => 0, Medium => 5, High => 10);
for Alert'Size use 4;

for Packet use record
   Flag   at 0*Byte range 3 .. 3;  -- Bits 1..2 are unused
   Danger at 0*Byre range 4 .. 7;
   Data   at 1*Byte range 0 .. 7;
end Packet;
for Packet'Size use 2*Byte;
```

5.14 Interface Testing (D.34)

Ada allows extending the expressiveness of an interface specification at the code level, allowing constraints such as:

- parameter passing modes
- pre- and postconditions
- input partitioning
- typing

These are each described in other sections of this document. These specifications can help the development of tests around the interface, formalize constraints on how the interface is supposed to be used, and activate additional dynamic checking or formal proofs (through SPARK), all ensuring that users are indeed respecting the expectations of the interface designer.

In addition, GNATtest can generate a testing framework to implement interface testing, and GNATfuzz can help by probing the robustness of the system when interface requirements are violated.

5.15 Language Subset (D.35)

The Ada language has been designed to facilitate subsetting, since its targeted domain — long-lived safety-critical embedded systems — often involves small-footprint applications that need to be certified under demanding software standards. The full language would be inappropriate with such constraints, and Ada provides a general feature — pragma Restrictions — to allow subsetting on a user-selectable basis. For example, with `pragma Restrictions (No_Abort_Statements)` the program will be rejected by the compiler if it contains an `abort` statement.

Going one step further, the language standard has bundled a set of restrictions into a so-called profile — the Ravenscar Profile — that supports common concurrency idioms (e.g. periodic and sporadic tasks) and can make a tasking program deterministic and statically analyzable.

SPARK is a natural Ada language subset, constraining the language so that programs can be subject to formal analysis (e.g., safe pointers, no aliasing, and no exceptions).

Other language subsets can be supplied by the implementation, such as the features implemented by the GNAT Pro Certifiable Profiles. And with GNATcheck the user can in effect define a subset in an *à la carte* fashion, to specify prohibited constructs and verify that they are not present in the code.

5.16 Metrics (D.37)

The GNATmetric tool reports various metrics on the code, from simple structural metrics such as lines of code or number of entities to more complex computations such as cyclomatic complexity or coupling.

Custom metrics can be computed based on these first-level metrics. In particular, the GNATdashboard environment allows gathering all metrics into a database that can then be accessed through Python or SQL.

These metrics can be viewed through various interfaces.

5.17 Modular Approach (D.38)

5.17.1 Connections between modules shall be limited and defined, coherence shall be strong

Ada allows the developer to define group of packages that have different levels of coupling, through the notions of *child packages* and *private packages* as described below. In addition, the GNAT Pro technology provides the notion of a *project*, which defines a group of packages, possibly with a defined interface. These constructs can be used to define a tool-supported notion of *component* or *module* at the software level.

There are three main types of dependence between compilation units:

- Loose coupling through `with` clauses. If unit `Q` `with`s unit P, then `Q` can only access the entities in the visible part of P.
- Medium coupling through public child units. If P is a package, then child `P.Q` has visibility privileges that would not be available to a unit that only `with`s P. More specifically, `P.Q`, which is said to be a *public child*, can access the entities in the visible part of P. However, only the private part and body of `P.Q` can access the entities in the private part of P. And the entities declared in the body of P are only accessible in the package body itself.
- Tight coupling through private child units. As a generalization of public child units, if `P.Q` is declared as a *private* child, then the visible part of `P.Q` can also access the entities in the private part of P. This does not compromise encapsulation; the only units that can `with` a private child are units that otherwise would have access to the entities that the private child can see.

Ada's expressiveness makes it easier to develop large software systems, with precise control over the coupling between modules, and guaranteeing that data are only accessed by the intended units.

A typical example is the implementation of a complex system that needs to be spread across several packages. For example, suppose that packages `Communication` and `Interfaces`, contribute to the implementation of a signaling protocol. In Ada, this design can be implemented in three (or more) distinct files:

```
package Signaling is ...
private package Signaling.Communication is .
private package Signaling.Interfaces is ...
```

The two private packages are defined in separate files. They are private children of `Signaling`, which means they can only be used by the implementation of `Signaling`, and not by any module outside of the hierarchy.

In addition, tools can provide metrics on coupling between packages. GNATmetric has built-in support for retrieving these numbers.

At a coarser granularity, packages can be grouped together into a GNAT Project file (*GPR*), with a clear interface. An application architecture can be defined as a combination of project files.

5.17.2 Collections of subprograms shall be built providing several level of modules

Following the above example, it's possible to create public sub-modules as well, creating a hierarchy of services. Public child units are accessible to client code.

5.17.3 Subprograms shall have a single entry and single exit only

The GNATcheck tool has specific rules to verify this property on any Ada code.

5.17.4 Modules shall communicate with other modules via their interface

This is built-in to the Ada language. It's not possible to circumvent a package's interface. If a module is implemented using a coarser granularity, e.g. as a group of packages or at project level, then the project file description allows identifying those packages that are part of the interface and those packages that are not.

5.17.5 Module interfaces shall be fully documented

Although this is mostly the responsibility of the developer, Ada contracts can be used to formalize part of the documentation associated with a package interface, using a formal notation that can be checked for consistency by the compiler. This addresses the part of the documentation that can be expressed through boolean properties based on the software-visible entities.

5.17.6 Interfaces shall contain the minimum number of parameters necessary

The GNAT Pro compiler will warn about parameters not used by a subprogram implementation.

5.17.7 A suitable restriction of parameter number shall be specified, typically 5

GNATcheck allows specifying a maximum number of parameters per subprogram.

5.17.8 Unit Proof and Unit Test

GNATtest can be used to generate a unit testing framework for Ada applications.

SPARK performs a modular formal verification: it proves the postcondition of a subprogram according to its own precondition and the precondition and postconditions of its callees, whether or not these callees are themselves proven.

For a complete proof, all the subprograms of an application need to be formally proven. Where this is not possible, one subset can be proven and the other can be assumed to be true. These assumptions can then be verified using traditional testing methodology, allowing for a hybrid test / proof verification system.

5.18 Strongly Typed Programming Languages (D.49)

Ada is, from its inception, a strongly typed language, which supports both static and dynamic verification.

From a static verification point of view, each type is associated with a representation and a semantic interpretation. Two types with similar representations but different semantics will still be considered different by the compiler. For example:

```
type Kilometers is new Float;
type Miles is new Float:
```

These are distinct types. The compiler will not allow mixed operations, for example assigning a `Kilometers` value to a `Miles` variable, or adding a `Kilometers` value and a `Miles` value, unless explicit conversions are used. Mixing floating point and integer values is similar: the developer is responsible for deciding where and how conversion should be made.

From a dynamic verification point of view, types can be associated with constraints, such as value ranges or arbitrary boolean predicates. These type ranges and predicates will be verified at specific points in the application, allowing the early detection of inconsistencies.

5.19 Structure Based Testing (D.50)

AdaCore provides three tools to support structure based testing:

- GNATtest is a unit testing framework generator. It will run on Ada specifications, and generate a skeleton for each subprogram. The actual test can then be manually written into that skeleton.
- GNATemulator allows emulating code for a given target (e.g. PowerPC and Leon) on a host platform such as Windows or Linux. It's particularly well suited for running unit tests.
- GNATcoverage performs structural coverage analysis from an instrumented platform (GNATemulator or Valgrind on Linux or directly on a board through a Nexus probe). It supports statement coverage and decision coverage as well as MC/DC. Note that although EN 50128 requires compound condition coverage, Modified Condition/Decision Coverage (MC/DC) is usually accepted as a means of compliance.

5.20 Structured Programming (D.53)

The Ada language supports all the usual paradigms of structured programming. Complexity can be controlled with various tools, see *Analyzable Programs (D.2)* (page 49) for more details.

5.21 Suitable Programming Languages (D.54)

Ada is noted as "Highly Recommended" in the list of programming languages. Some features may, however, not be suitable for the highest SIL. To enforce the detection and rejection of specific features, the developer can specify a language subset; see *Language Subset (D.35)* (page 60).

One of the advantages of the Ada language is that it is precisely defined in an international document, ISO/IEC 8652. This document specifies the required effect as well as any implementation-defined behavior for the core language, the standard Ada libraries (known as the "predefined environment"), and the specialized needs annexes.

5.22 Object Oriented Programming (D.57)

Ada supports the usual constructs for object-oriented programming, but, for reasons of simplicity and reliability, with multiple inheritance limited to `interface` types. In addition, the Liskov Substitution Principle can be verified through class-wide contracts and SPARK formal verification, allowing the verification of class hierarchy consistency and the safety of dispatching operations.

Ada's OOP model is particularly well suited to safety-critical applications, as it allows instantiating objects on the stack. For example:

```
type Base_Class is tagged ...; -- Base_Class is the root of a class hierarchy
procedure P (X : Base_Class);
...
type Subclass is new Base_Class with ...;
overriding procedure P (X : Subclass);

B : Base_Class := ... -- on the stack
S : Subclass   := ... -- on the stack

X : Some_Type'Class := (if .. then B else S):
P (X);  -- Dispatches to appropriate version of P
```

In the above code, X is a polymorphic object that can be initialized with a value from any class in the hierarchy rooted at `Base_Class`; here it will be a value from either `Base_Class` or `Subclass`. Storage for X is reserved on the stack, and the invocation `P (X)` will dispatch to the appropriate version of P.

The booklet[36] provides additional information on how to use object-oriented features in a certified context.

5.23 Procedural Programming (D.60)

Ada implements all the usual features of procedural programming languages, with a general-purpose data type facility and a comprehensive set of control constructs.

[36] *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/.

CHAPTER

SIX

TECHNOLOGY ANNEX

This annex summarizes how AdaCore's tools and technologies support the various techniques and measures defined in Annex D of EN 50128. The qualification status for tools, and certifiability for run-time libraries, are also noted.

6.1 Ada Programming Language

See *Ada* (page 17).

6.1.1 Qualification

Although there is no qualification of a language *per se*, the Ada language is standardized through an official process managed by an ISO committee, IEC/ISO 8652. AdaCore's Ada compilers and tools have reference and user documentation that precisely describes the expected behavior, including the effects of implementation-defined features.

6.1.2 Annex D References

- D.2 Analyzable Programs
- D.4 Boundary Value Analysis
- D.14 Defensive Programming
- D.18 Equivalence Classes and Input Partition Testing
- D.24 Failure Assertion Programming
- D.33 Information Hiding / Encapsulation
- D.34 Interface Testing
- D.35 Language Subset
- D.38 Modular Approach
- D.49 Strongly Typed Programming Languages
- D.53 Structured Programming
- D.54 Suitable Programming Languages
- D.57 Object Oriented Programming
- D.60 Procedural Programming

6.2 GNAT Pro Assurance Toolsuite

6.2.1 Qualification

GNAT Pro compiler family

The GNAT Pro compilers for Ada and for C are qualified at class T3. AdaCore can provide documentation attesting to various aspects such as service history, development standard, and testing results. This documentation has been submitted and accepted in past certification activities. T3 qualification material can also be developed for the GNAT Pro for C++ and GNAT Pro for Rust compilers.

Since compilers are large and complex pieces of software, bugs can be detected (and subsequently corrected) after a particular version has been chosen. Following the requirements stated in 6.7.4.11, however, a corrected version of the compiler cannot be deployed without specific justification. AdaCore offers a dedicated service – GNAT Pro Assurance – on a specified version of the technology, which provides critical problem fixes (or workaround suggestions) as well as detailed descriptions of the changes. Using GNAT Pro Assurance, a customer can integrate a corrected version of a specific compiler release into their development infrastructure without the risk of regressions from unwanted updates.

See *GNAT Pro Assurance* (page 23).

GNATstack

GNATstack can be qualified as a class T2 tool.

6.2.2 Run-Time Certification

Certification material up to SIL 4 can be developed for the Light and Light-Tasking run-time libraries.

See *Configurable Run-Time Libraries* (page 23).

6.2.3 Annex D References

- D.10 Data Flow Analysis
- D.15 Coding Standards and Style Guide
- D.18 Equivalence Classes and Input Partition Testing
- D.35 Language Subset

6.3 SPARK Language and Toolsuite

See *SPARK* (page 21).

6.3.1 Qualification

The SPARK Pro toolsuite can be qualified at class T2.

6.3.2 Annex D References

The SPARK language and toolset can contribute to the deployment or implementation of the following techniques:

- D.2 Analyzable Programs

- D.4 Boundary Value Analysis
- D.10 Data Flow Analysis
- D.14 Defensive Programming
- D.18 Equivalence Classes and Input Partition Testing
- D.24 Failure Assertion Programming
- D.28 Formal Methods
- D.29 Formal Proof
- D.34 Interface Testing
- D.35 Language Subset
- D.38 Modular Approach
- D.57 Object Oriented Programming

6.4 GNAT Static Analysis Suite

See *GNAT Static Analysis Suite (GNAT SAS)* (page 26).

6.4.1 Defects and Vulnerability Analysis

6.4.1.1 Qualification

GNAT SAS's defects and vulnerability analysis tool can be qualified at class T2. It has a long cross-industry track record and has been qualified under other standards in the past, such as DO-178B/C as a verification tool/TQL5.

6.4.1.2 Annex D References

GNAT SAS's defects and vulnerability analysis tool can contribute to the deployment or implementation of the following techniques:

- D.2 Analyzable Programs
- D.4 Boundary Value Analysis
- D.8 Control Flow Analysis
- D.10 Data Flow Analysis
- D.14 Defensive Programming
- D.18 Equivalence Classes and Input Partition Testing
- D.24 Failure Assertion Programming
- D.32 Impact Analysis

6.4.2 Basic Static Analysis tools

The basic tools are GNATcheck and GNATmetric.

6.4.2.1 Qualification

These tools can be qualified at class T2. GNATcheck has been qualified under other standards as well, such as DO-178B/C as a verification tool/TQL5.

6.4.2.2 Annex D References

- D.2 Analyzable Programs
- D.14 Defensive Programming
- D.15 Coding Standard and Style Guide
- D.35 Language Subset
- D.37 Metrics

6.5 GNAT Dynamic Analysis Suite

This suite comprises GNATtest, GNATemulator, GNATcoverage, GNATfuzz, and TGEN.

See *GNAT Dynamic Analysis Suite (GNAT DAS)* (page 27).

6.5.1 Qualification

GNATtest, GNATemulator and GNATcoverage can be qualified at class T2. GNATcoverage has been qualified under other standards as well, such as DO-178B/C as a verification tool/TQL5.

6.5.2 Annex D References

- D.50 Structure Based Testing

INDEX

A

Absence of Run-Time Errors (*AORTE*), 50
Ada language, 7
    Concurrent programming, 19, 59
    Contract-based programming, 18, 21
    Coupling between modules, 61
    Generic templates, 19
    High-integrity systems, 20
    History, 17
    Memory safety, 20
    Object-Oriented Programming (*OOP*), 19, 57
    Package feature, 58
    Parameter checking, 53
    Postconditions, 18
    pragma Restrictions, 21
    Preconditions, 18
    Prevention of buffer overflow, 20
    Prevention of dangling references, 20
    Prevention of null pointer dereferencing, 20
    Prevention of vulnerabilities, 20
    Programming in the large, 19
    Protected object / Protected type, 59
    Real-time programming, 20
    Representation clause, 59
    Scalar ranges, 18
    Strong typing, 20
    Support for Analyzable Programs (*D.2*), 49
    Support for Annex D techniques (*summary*), 65
    Support for Boundary Value Analysis (*D.4*), 50
    Support for Defensive Programming (*D.14*), 52, 53
    Support for Information Encapsulation (*D.33*), 57
    Support for Interface Testing (*D.34*), 59
    Support for Modular Approach (*D.38*), 61, 62
    Support for Object Oriented Programming Languages (*D.57*), 63
    Support for Procedural Programming (*D.60*), 64
    Support for Strongly Typed Programming Languages (*D.49*), 62
    Support for Structured Programming (*D.53*), 63
    Support for Suitable Programming Languages (*D.54*), 63
    Systems programming, 20
    Task object / task type, 59
AdaCore
    Support services, 24
    Training and consulting services, 24
Annex D
    Analyzable Programs (*D.2*), 49
    Boundary Value Analysis (*D.4*), 49
    Coding Standards and Style Guide (*D.15*), 55
    Control Flow Analysis (*D.8*), 50
    Data Flow Analysis (*D.10*), 51
    Defensive Programming (*D.14*), 52
    Error Guessing (*D.20*), 55
    Failure Assertion Programming (*D.24*), 56
    Formal Methods (*D.28*), 56
    Impact Analysis (*D.32*), 57
    Information Encapsulation (*D.33*), 57
    Interface Testing (*D.34*), 59
    Language Subset (*D.35*), 60
    Metrics (*D.37*), 60
    Modular Approach (*D.38*), 60
    Object Oriented Programming (*D.57*), 63
    Procedural Programming (*D.60*), 64
    Structure Based Testing (*D.50*), 63
    Structured Programming (*D.53*), 63
    Suitable Programming Languages (*D.54*), 63

B

Babbage (*Charles*), 17
Buffer overflow, 20
Byron (*Lord George*), 17

C

C language support, 23

C++ language support, 23
CENELEC, 5
Certifiable profile, 23
    Support for Language Subset (*D.35*), 60
Child package
    Support for Modular Approach (*D.38*), 61
Common Criteria security standard, 20
Contract_Cases aspect, 55
Contract-based programming, 21
    Support for Failure Assertion Programming (*D.24*), 56

D

Defects and Vulnerability Analysis
    Qualification, 67
    Support for Annex D techniques (*summary*), 67
Defects and vulnerability analysis (*in GNAT SAS*), 26
Dimension consistency analysis, 53

E

Eclipse IDE, 8, 30
EN 50126, 5
EN 50128
    Annex A (*Criteria for the Selection of Techniques and Measures*), 12
    Annex B (*Key software roles and responsibilities*), 13
    Annex C (*Documents Control Summary*), 13
    Annex D (*Bibliography of techniques*), 13
    Annex ZZ, 14
    Clause 4 (*Objectives, conformance and software safety integrity levels*), 10
    Clause 5 (*Software management and organization*), 10
    Clause 6 (*Software assurance*), 11
    Clause 7 (*Generic software development*), 11
    Clause 8 (*Development of application data or algorithms*), 11
    Clause 9 (*Software deployment and maintenance*), 11
    Structure of the standard, 9
EN 50129, 5
EN 50657, 6
EN 50716, 6

G

GNAT Dynamic Analysis Suite (*GNAT DAS*), 8, 27
    GNATcoverage, 8, 28
    GNATemulator, 8, 27
    GNATfuzz, 8, 28
    GNATtest, 8, 27
    TGen, 8, 28
GNAT Pro Assurance, 7, 23
    Configurable Run-Time Libraries, 23
    GNAT Pro for Ada, 23
    GNAT Pro for C, 23
    GNAT Pro for C++, 23
    GNAT Pro for Rust, 23
    GNATstack, 25
    Libadalang, 24
    Qualification, 66
    Source-to-object traceability, 24
    Support for Annex D techniques (*summary*), 66
    Support for Coding Standards and Style Guide (*D.15*), 55
    Support for Data Flow Analysis (*D.10*), 51
    Support for Defensive Programming (*D.14*), 53
    Sustained branch, 23
GNAT Pro for Rust, 8, 29
GNAT Static Analysis Suite (*GNAT SAS*), 7, 26
    Defects and vulnerability analysis, 7, 26
    GNATcheck, 7, 26
    GNATmetric, 7, 26
    Support for Analyzable Programs (*D.2*), 49
    Support for Boundary Value Analysis (*D.4*), 50
    Support for Control Flow Analysis (*D.8*), 50
    Support for Data Flow Analysis (*D.10*), 51
    Support for Defensive Programming (*D.14*), 53, 54
    Support for Impact Analysis (*D.32*), 57
GNAT Studio IDE, 8, 29
    Support for Analyzable Programs (*D.2*), 49
    Support for Control Flow Analysis (*D.8*), 50
    Support for Impact Analysis (*D.32*), 57
GNATbench IDE, 8, 30
GNATcheck, 7, 26
    Qualification, 67
    Support for Analyzable Programs (*D.2*), 49
    Support for Annex D techniques (*summary*), 67
    Support for Coding Standards and Style Guide (*D.15*), 55
    Support for Defensive Programming (*D.14*), 52
    Support for Language Subset (*D.35*),

60
Support for Modular Approach (*D.38*), 62
GNATcoverage, 8, 28
Qualification, 68
Support for Annex D techniques (*summary*), 68
Support for Structure Based Testing (*D.50*), 63
GNATdashboard IDE, 8, 30
GNATemulator, 8, 27
Qualification, 68
Support for Annex D techniques (*summary*), 68
Support for Structure Based Testing (*D.50*), 63
GNATformat, 27
GNATfuzz, 8, 28, 29
Support for Error Guessing (*D.20*), 55
GNATmetric, 7, 26
Metrics on inter-package coupling, 61
Qualification, 67
Support for Analyzable Programs (*D.2*), 49
Support for Annex D techniques (*summary*), 67
Support for Control Flow Analysis (*D.8*), 50
Support for Metrics (*D.37*), 60
GNATprove, 21
GNATstack, 25
Support for Control Flow Analysis (*D.8*), 51
Tool qualification, 66
GNATtest, 8, 27, 29
Qualification, 68
Support for Annex D techniques (*summary*), 68
Support for Interface Testing (*D.34*), 60
Support for Modular Approach (*D.38*), 62
Support for Structure Based Testing (*D.50*), 63

H

Hybrid verification, 22

I

Ichbiah (*Jean*), 17
Integrated Development Environments (*IDEs*), 8, 29
Eclipse, 30
GNAT Studio, 8, 29
GNATbench, 8, 30
GNATdashboard, 8, 30
VS Code support, 8, 30
Workbench, 30

J

Jorvik profile, 21

L

Libadalang, 24
Light Profile, 7, 23
Certification material, 66
Light-Tasking Profile, 23
Certification material, 66
Liskov Substitution Principle, 35, 47, 63
Lovelace (*Augusta Ada*), 17

M

Memory safety, 20

P

pragma Assert
Support for Failure Assertion Programming (*D.24*), 56
pragma Restrictions
Support for Coding Standards and Style Guide (*D.15*), 55
Support for Language Subset (*D.35*), 60
Private package
Support for Modular Approach (*D.38*), 61
Project (*GNAT Pro*)
GPR files, 61
Support for Modular Approach (*D.38*), 61

R

Range checking, 52
Ravenscar Profile, 7, 17, 21, 23
Support for Language Subset (*D.35*), 60
Rust language support, 23, 29

S

Safety Integrity Level (*SIL*), 6
Software Quality Assurance Plan, 30
SPARK language, 21
Formal verification, 22
Hybrid verification, 22
Reduced cost of verification, 22
Static verification, 22
Support for Language Subset (*D.35*), 60
Usage, 21
SPARK Pro toolsuite, 21
GNATprove, 21
Qualification, 66

SPARK technology, 7
    Absence of Run-Time Errors (*AORTE*), 22, 50
    Support for Analyzable Programs (*D.2*), 49
    Support for Annex D techniques (*summary*), 66
    Support for Boundary Value Analysis (*D.4*), 50
    Support for Coding Standards and Style Guide (*D.15*), 55
    Support for Data Flow Analysis (*D.10*), 51
    Support for Defensive Programming (*D.14*), 53, 54
    Support for Failure Assertion Programming (*D.24*), 56
    Support for Formal Methods (*D.28*), 56
    Support for Modular Approach (*D.38*), 62
Sustained branch, 23

T

T1 tool class, 14
T2 tool class, 14
T3 tool class, 14
Taft (*Tucker*), 17
TGen, 8, 28
Tool classes, 14
Tool qualification, 14
    AdaCore support, 15
    Defects and Vulnerability Analysis, 67
    GNAT Pro Assurance, 66
    GNATcheck, 67
    GNATcoverage, 68
    GNATemulator, 68
    GNATmetric, 67
    GNATstack, 66
    GNATtest, 68
    SPARK Pro toolsuite, 66

V

V software life cycle, 7
VS Code support, 8, 30

W

Workbench IDE (*Wind River*), 30